UniTs - University of Trieste

Faculty of Scientific and Data Intensive Computing

Department of mathematics informatics and geosciences

Advanced High Performance Computing

Lecturer:
Prof. Luca Tornatore

Author:
Andrea Spinelli

May 6, 2026

github.com/Spina02 andreaspinelli2002@gmail.com

Preface

In this course, we will explore a set of fundamental topics essential for high-performance and parallel computing:

- **Vectorization**: methods for processing multiple data elements at once to fully utilize modern CPU architectures.
- **OpenMP**: a popular interface for developing shared-memory parallel programs that allows straightforward multi-threading.
- **GPU**:
  - **cuda**: NVIDIA's platform and API for general-purpose computing on GPUs.
  - **openacc**: a directive-based standard for simplifying the acceleration of code on GPUs and other devices.
- **MPI**:
  - **Data-Types**: constructing and efficiently communicating custom and built-in datatypes in distributed environments.
  - **I/O**: strategies and tools for performing parallel input and output in large-scale applications.
  - **one-sided communication**: advanced features for direct remote memory access and asynchronous messaging.
- **MPI in python**: using the Message Passing Interface within the Python ecosystem for approachable and flexible parallel programming.

Contents

1 Vectorization

1.1 Basic Concepts

Modern high-performance computing systems exploit multiple layers of parallelism to achieve maximum performance. At the largest scale, clusters or supercomputers are made up of many nodes working together. Within each node, there may be several CPUs (sockets), each containing multiple processing cores capable of running tasks concurrently.

A further dimension in modern HPC is ***offloading***, where compute-intensive tasks and associated data are delegated from the CPU to accelerators such as GPUs (Graphics Processing Units). Offloading entails transferring data across the system's architecture, a process whose performance impact depends on the system's memory configuration and interconnect.

- In **NVIDIA-based systems**, CPUs typically access DDR5 RAM, while GPUs benefit from their own high-bandwidth HBM3 memory. Data movement across the CPU-GPU boundary (usually via PCIe or NVLink) can be a bottleneck and must be carefully optimized.
- In contrast, **AMD's recent architectures** (notably with some Instinct GPUs and EPYC CPUs) may feature a unified DDR5 memory pool shared by both CPU and GPU, reducing data transfer overheads and enabling more direct sharing.

In addition to thread and process-level parallelism, modern processors employ a crucial form of parallelism: **vectorization**. Vectorization embodies ***Single Instruction Multiple Data (SIMD)*** execution, a paradigm in which a single instruction operates on multiple data elements simultaneously using specialized hardware known as ***vector registers***.

Vector registers (VREGs) are designed for operations on aggregated data, they are considerably wider than traditional scalar registers, commonly supporting 128, 256, or even 512 bits at a time.

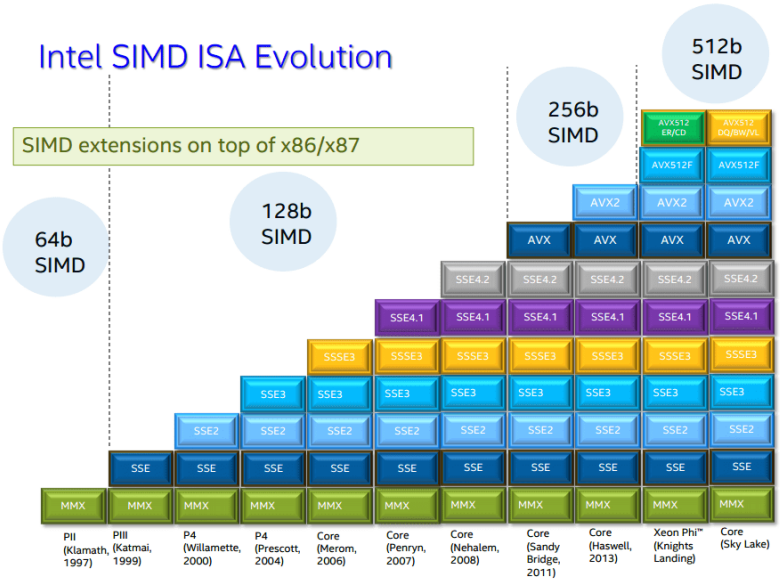


Figure 1.1: Intel SIMD ISA Evolution: timeline of vector instruction set extensions introduced with each CPU generation. The diagram shows how SIMD register width increased from MMX and SSE (64-128 bits) to AVX (256 bits) and AVX-512 (512 bits), with each new standard supporting more data parallelism.

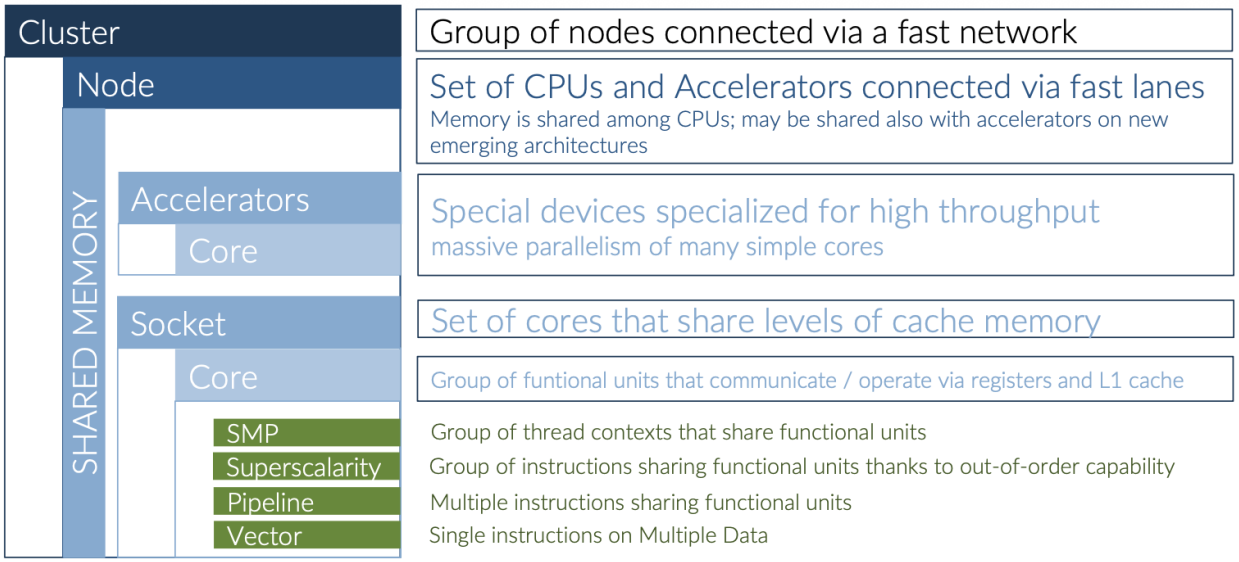


Figure 1.2: The architecture of a modern HPC cluster.

Modern x86 CPUs have evolved both in structure and capacity. The main idea is straightforward: the wider the register, the more data elements you can work on at once with a single instruction. Intel has introduced three generations of SIMD registers:

- **SSE (128-bit):** Each register (`xmm0`) can hold either 2 double-precision (64-bit) values or 4 single-precision (32-bit) floating-point values. This was the first major step for SIMD on x86.
- **AVX (256-bit):** The register width doubles. Now, a `ymm0` register can operate on 4 doubles or 8 floats at once, effectively doubling the amount of data processed per instruction.
- **AVX-512 (512-bit):** The register width doubles yet again. The `zmm0` register can simultaneously process up to 8 doubles or 16 floats.

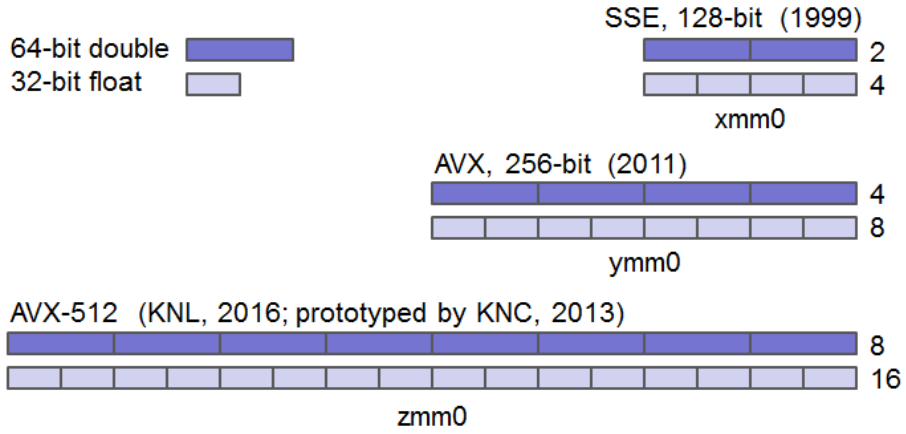


Figure 1.3: SIMD registers: each generation widens the registers, allowing more doubles (dark) or floats (light) to be processed in parallel. [3]

Wider registers over the generations (SSE → AVX → AVX-512) translate directly into the ability to operate on more data elements at once. This increase in register width is key to the dramatic gains in performance for vectorizable workloads, as it enables each instruction to do more useful work in parallel.

How to achieve vectorization?

Vectorization can be approached at various levels, from the most convenient and portable to the most hardware-specific and optimized:

- **Explicit compiler pragmas** (e.g., `#pragma omp simd`, `#pragma vector`): Code annotations that help the compiler vectorize loops or code regions. These are high-level, portable, require few changes, but give only moderate control.
- **Auto-vectorization by the compiler**: Compilers can often detect and apply SIMD to loops with the right options (`-O2`, `-ftree-vectorize`, etc). Convenient, needs no source changes, but gives little control and can miss opportunities.
- **C++ vector libraries**: Libraries such as `std::valarray`, `Vc`, or `Eigen` let you write high-level vector code, balancing portability and some SIMD control.
- **Compiler-specific vector extensions**: Using types like `__m256d` or `float32x4_t` gives direct SIMD register access, improving performance but reducing portability.
- **Intrinsics**: Functions mapping to hardware SIMD instructions (e.g., `_mm256_add_pd`), allow very fine-grained, fast vector code at the cost of readability and portability.
- **Handwritten assembly**: Writing assembly gives maximum control and performance, but is not portable and is complex and error-prone.

Higher-level approaches are simpler and portable; lower-level ones give more control and tuning, but cost portability and add complexity.

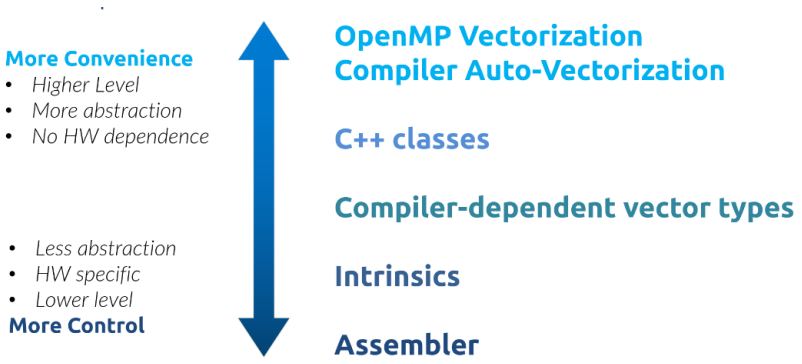


Figure 1.4: Approaches to vectorize, from high-level convenience to low-level control. [1]

Different architectures

Vector ISAs follow two main philosophies. The traditional x86 ISA uses **fixed-width** vectors, while ARM and RISC-V adopt a **scalable** design where the vector length is not fixed in the binary.

Feature	x86 (AVX/AVX-512)	ARM (SVE)	RISC-V RVV
Philosophy	***Fixed-width*** (128/256/512 bit)	***Scalable*** (128–2048 bit)	***Scalable*** (undefined VLEN)
Binary model	***Fragmented*** (compile for each target)	***VLA*** (Vector-Length-Agnostic) write once, run anywhere	***VLA*** write once, run anywhere
Register count	16 (AVX2) 32 (AVX-512)	32	32 + LMUL grouping
Registers flexibility	None (fixed per register/size)	None (set at HW level, 128–2048 bit)	LMUL (register grouping)
Predication	8 mask registers (AVX-512 add-on)	16 predicate registers (core feature)	Built-in mask operand (often `v0`)
Configuration	***Implicit*** (by the called instruction)	***Implicit*** (type chosen; length is loop-invariant)	***Explicit*** via `vsetvl`
ISA complexity	***High fragmentation*** (AVX-512F, CD, VL, ...)	***Low fragmentation*** (NEON, SVE, SVE2)	***Unified*** (RVV 1.0)
Hardware cost	Can cause clock throttling (AVX-512)	Designed for power/performance scaling	—

- **Fixed vs Scalable size**: On x86_64 you must target a specific register width (SSE 128, AVX 256, AVX-512 512) and call the corresponding instructions or binaries. This forces specialized code paths and ***fragmentation***. With scalable architectures (SVE, RVV) you write ***Vector-Length-Agnostic (VLA)*** code: the same loop adapts to the hardware at run time, running efficiently on a 128-bit core or on a server with 1024-bit registers, greatly reducing fragmentation.
- **Native vs add-on predication**: Predication lets a vector instruction conditionally process elements without branching, by creating a boolean *mask* to decide which lanes are active.

```
for (int i = 0; i < N; ++i)
    if (data[i] > 0.0)
        data[i] *= 2.0;
```

If the mask for an 8-lane vector is, for instance:

```
mask: [ 0 0 1 1 1 0 1 1 ]
data: [ u u p p p u p p ]  // u = unchanged, p = processed
```

The inactive lanes (0) are left untouched, while the active lanes (1) are updated. In practice there are two ways to realize this:
– skip the false lanes so they are not executed at all; or
– execute all lanes but ignore the results of the false lanes ("blend" them with the old values).
SVE provides **native predication** via 16 predicate registers and rich lane-wise logic; RVV uses a built-in mask operand (commonly `v0`) as part of the ISA. In x86, predication is an *add-on* of AVX-512 (8 mask registers), while up to AVX2 it is emulated via blending operations.

1.2 Checking for SIMD capabilities

You can check **statically** the value for your cpu, for instance by grepping the output of either `lscpu` or `/proc/cpuinfo`. But what if you want to check the capabilities of the architecture where the code is running? We can use multiple methods to check the capabilities of the architecture:

- **Compiler built-in functions**: The compiler provides built-in functions to check the capabilities of the architecture.

```
#include <cpuid.h>

int main() {
    __builtin_cpu_init();

    if (__builtin_cpu_supports("avx512f"))
        ...
    if (__builtin_cpu_supports("avx2"))
        ...
    if (__builtin_cpu_supports("avx"))
        ...
    if (__builtin_cpu_supports("sse4.2"))
        ...
    if (__builtin_cpu_supports("sse4.1"))
        ...
    if (__builtin_cpu_supports("sse3"))
        ...
```

- **Compiler intrinsics**: The compiler provides intrinsics to check the capabilities of the architecture.

```
#include <immintrin.h>

#ifdef __AVX512__
#define V_DSIZE (sizeof( __m512d ) / sizeof(double) )

#elif defined ( __AVX__ ) || defined ( __AVX2__ )
#define V_DSIZE (sizeof( __m256d ) / sizeof(double) )

#elif defined ( __SSE4__ ) || defined ( __SSE3__ )
#define V_DSIZE (sizeof( __m128d ) / sizeof(double) )

#else
#define V_DSIZE 1UL
#endif
```

Tip: ***-march=native***

By default, the compiler is able to correctly recognize the CPU's capabilities but actually gets a wrong vector size. We need to use the `-march=native` flag, since it will not be able to detect the vector capabilities of the architecture.

In some cases, you might need to explicitly target a particular SIMD extension (for either Intel or AMD architectures). This can be accomplished by specifying the appropriate compiler flags:

- **`-mavx512f`** : Enables AVX-512 instructions.
- **`-mavx2`** : Enables AVX2 instructions.
- **`-mavx`** : Enables AVX instructions.
- **`-msse4.2`** : Enables SSE4.2 instructions.
- **`-msse4.1`** : Enables SSE4.1 instructions.
- **`-msse3`** : Enables SSE3 instructions.

1.3 Vector Types via Intrinsics

Once we include the `<immintrin.h>` header, we get access to the intrinsincs routines and types. **Intrinsics** are functions that are provided by the compiler to allow the programmer to directly use

the vector instructions of the architecture.

INTEGER types

types name	int 8bits	int 16bits	int 32bits	int 64bits
`__m64`	8x	4x	2x	1x
`__m128i`	16x	8x	4x	2x
`__m256i`	32x	16x	8x	4x
`__m512i`	64x	32x	16x	8x

FLOATING-POINT types

types name	single precision	double precision
`__m128`	4x	–
`__m128d`	–	2x
`__m256`	8x	–
`__m256d`	–	4x
`__m512`	16x	–
`__m512d`	–	8x

To make explicit use of vector types while keeping the code portable across machines, it is convenient to introduce a small layer of typedefs and wrappers that adapts at compile time to the ISA detected by the compiler. The names `dvector_t`, `fvector_t` and `ivector_t` will always exist in our code with the correct width, and a couple of macros expose their element count and bit size.

```
// Select vector types based on the available ISA
#ifdef __AVX512__
typedef __m512d dvector_t;
typedef __m512  fvector_t;
typedef __m512i ivector_t;

#elif defined ( __AVX__ ) || defined ( __AVX2__ )
typedef __m256d dvector_t;
typedef __m256  fvector_t;
typedef __m256i ivector_t;

#elif defined ( __SSE4__ ) || defined ( __SSE3__ )
typedef __m128d dvector_t;
typedef __m128  fvector_t;
typedef __m128i ivector_t;

#else
typedef double dvector_t;
typedef float  fvector_t;
typedef int    ivector_t;
#endif

// Convenience macros for sizes (elements per vector and bit width)
#define DV_ELEMENT_SIZE ( sizeof(dvector_t) / sizeof(double) )
#define DV_BIT_SIZE     ( sizeof(dvector_t) * 8 )
#define FV_ELEMENT_SIZE ( sizeof(fvector_t) / sizeof(float) )
#define FV_BIT_SIZE     ( sizeof(fvector_t) * 8 )
#define IV_ELEMENT_SIZE ( sizeof(ivector_t) / sizeof(int) )
#define IV_BIT_SIZE     ( sizeof(ivector_t) * 8 )
```

In real codes one typically hides the operations behind tiny wrappers so that call sites stay uniform: a short `#ifdef` region selects the proper `SSE` / `AVX` / `AVX-512` intrinsic. As an example, below we provide a simple “vector summation” for doubles, floats and integers, via `VDSUM`, `VFSUM` and `VISUM`. We will return to intrinsics usage later in the chapter.

Example: *Vector summation*

Wrappers for a “vector summation” on doubles, floats and ints:

```
#ifdef __AVX512__
#define VDSUM(v1, v2, r) ( (r) = _mm512_add_pd( (v1), (v2) ) )
#define VFSUM(v1, v2, r) ( (r) = _mm512_add_ps( (v1), (v2) ) )
#define VISUM(v1, v2, r) ( (r) = _mm512_add_epi32( (v1), (v2) ) )
#elif defined ( __AVX__ ) || defined ( __AVX2__ )
#define VDSUM(v1, v2, r) ( (r) = _mm256_add_pd( (v1), (v2) ) )
#define VFSUM(v1, v2, r) ( (r) = _mm256_add_ps( (v1), (v2) ) )
#define VISUM(v1, v2, r) ( (r) = _mm256_add_epi32( (v1), (v2) ) )
#elif defined ( __SSE4__ ) || defined ( __SSE3__ )
#define VDSUM(v1, v2, r) ( (r) = _mm_add_pd( (v1), (v2) ) )
#define VFSUM(v1, v2, r) ( (r) = _mm_add_ps( (v1), (v2) ) )
#define VISUM(v1, v2, r) ( (r) = _mm_add_epi32( (v1), (v2) ) )
#else
#define VDSUM(v1, v2, r) ( (r) = (v1) + (v2) )
#define VFSUM(v1, v2, r) ( (r) = (v1) + (v2) )
#define VISUM(v1, v2, r) ( (r) = (v1) + (v2) )
#endif
```

Warning: *Production-ready headers*

The approach we have just discussed is intended for didactical purposes. For comprehensive, production-ready headers, consider for instance:

- Agner Fog's VectorClass Library (VCL)
- Google's Highway

1.4 Loops auto-vectorization

Auto-vectorization refers to the compiler's ability to automatically convert suitable loops and code blocks into vector instructions, allowing data-level parallelism and boosting performance even without manual use of SIMD intrinsics. While this is a powerful tool, it is important to be aware that successful auto-vectorization is not always guaranteed. Various elements can limit or block this process, such as:

- loop-carried data dependencies
- complex control dependencies or conditional branches within the loop
- unaligned or poorly aligned memory accesses
- loop structures that do not match vectorization patterns
- strided or irregular memory access patterns
- calls to functions that cannot be inlined or vectorized
- use of math functions that lack vectorizable implementations
- data types unsupported by the target SIMD ISA
- …

It's important to keep these obstacles in mind when writing code that we want to auto-vectorize.

To ask the compiler to vectorize loops computations, we can use the following flags:

Compiler	Flag	Description
GCC	`-ftree-vectorize`	enables loops vectorization
	`-funroll-loops`	enables the loop unrolling (may or may not issue faster code)
	`-march=native`	specifies the current one as target architecture
	`-mtune=native`	ask maximum code tuning for the host architecture
clang	`-mllvm`	(uses LLVM's internal vectorization)
	`-force-vector-width=4`	enforces vector width (e.g., 4) for SIMD instructions
Intel	`-xHost`	enables maximum vectorization available on the target cpu
	`-axFLAG1 -axFLAG2 ...`	enables code for multiple targets

If we want to ask the compiler to report on optimizations and vectorizations, we can use the following flags:

Compiler	Flag	Description
GCC	`-fopt-info-vec-optimized`	reports on the successful vectorizations
	`-fopt-info-vec-missed`	reports on the reasons for unsuccessful vectorization
	`-fopt-info-vec-all`	reports all vectorization optimizations
clang	`-Rpass=loop-vectorize`	identifies loops that were successfully vectorized
	`-Rpass-missed=loop-vectorize`	identifies loops that were NOT successfully vectorized
	`-Rpass-analysis=loop-vectorize`	identifies statements that caused vectorization to fail (with `-fsave-optimization-record`, detailed causes may be listed)
Intel	`-qopt-report=n`	enables diagnostic output; n=1: vectorized, n=2: non-vectorized+reasons, n=3: dependency info, n=4: only vectorization, n=5: dependency issues
	`-qopt-report-file=`*`name`*	writes the report to the specified file

In addition to enabling vectorization flags, we can provide the compiler with additional directives and hints to facilitate and optimize code vectorization:

Compiler	Pragma	Description
GCC	`#pragma GCC ivdep`	Ignore potential loop-carried dependencies that are not formally proven
	`#pragma GCC unroll n`	Unroll the subsequent loop body `n` times
clang	`#pragma clang ivdep`	Ignore potential loop-carried dependencies
	`#pragma clang vectorize(enable|disable)`	Enable/disable auto-vectorization of the loop
	`#pragma clang vectorize_width(n)`	Specify the vector length (VL)
	`#pragma clang interleave(enable|disable)`	Enable/disable unrolling/interleaving of the loop
	`#pragma clang interleave_count(n)`	Specify how many iterations are to be unrolled/interleaved
Intel	`#pragma ivdep`	Ignore potential loop-carried dependencies
	`#pragma vector always`	Vectorize even if the estimated gain is low/negative
	`#pragma vector aligned`	Assume aligned data
	`#pragma vector unaligned`	Assume unaligned data

Tip: ***Compiler's manual***

Check the compiler's manual for comprehensive and updated descriptions of available vectorization pragmas and options.

Example: *Auto vectorizing a simple loop*

Here we perform a simple reduction of an array, with datatype `dtype` that is determined at compile time (`int` or `float`).

```
dtype sum_loop ( dtype *array, uint N )
{
    dtype sum = 0;
    for ( uint32_t i = 0; i < N; i++ ) {
        sum += array[i];
    }
    return sum;
}
```

Let's compile the code with GCC using different flags:

- To get the generated assembler in intel syntax:
  `-S -fverbose-asm -masm=intel`
- To enable vectorization and optimization:
  `-O3 -march=native -mtune=native -ftree-vectorize -funroll-loops`
- To get a report on vectorization:
  `-fopt-info-vec-optimized -fopt-info-vec-missed -fopt-info-loop`
- To set the data type:
  `-DTYPE=[INTEGER|FPDP]`

Now, inspecting the assembly code for the integer version:

```
.L4:
    vpaddd  ymm0, ymm0, YMMWORD PTR [rax]
    add     rax, 32
    cmp     rdx, rax
    jne     .L4
```

which means take what is inside register `rax` (the square brackets mean dereferencing), sum it with the value in `ymm0` and store the result in `ymm0` (all of 32 bytes, or 256 bits). Then increment `rax` by 32 bytes (the size of the vector) and repeat until `rax` reaches `rdx` (the end of the array). This is a clear example of vectorization: we are summing 8 integers at once, instead of one by one.

Vectorizing loops operating on integers works well, but for floating point numbers, we need to be careful.

```
.L4:
    vaddss  xmm0, xmm0, DWORD PTR [rax]
    add     rax, 32
    vaddss  xmm0, xmm0, DWORD PTR -28[rax]
    vaddss  xmm0, xmm0, DWORD PTR -24[rax]
    vaddss  xmm0, xmm0, DWORD PTR -20[rax]
    vaddss  xmm0, xmm0, DWORD PTR -16[rax]
    vaddss  xmm0, xmm0, DWORD PTR -12[rax]
    vaddss  xmm0, xmm0, DWORD PTR -8[rax]
    vaddss  xmm0, xmm0, DWORD PTR -4[rax]
    cmp     rax, rcx
    jne     .L4
```

Here, we are using the `vaddss` instruction which is actually a scalar instruction, thus we are not actually vectorizing the loop.

What we have just seen should convince you that integers can be easily vectorized by the compiler because there are no constraints on the reshuffling of the operations. Floating points, instead, don't have this advantage: even when some vectorization is possible, the critical paths that we insert through the semantics hinder the expression of data-level parallelism. To further test the result of the compilation, let's profile the codes using `perf` :

	Int	**Float**
Not vectorized	0 int_vec.add_256	0 vector 1,000,000 scalar
Vectorized	0 int_vec.add_256	250,000 vector 1,000,000 scalar

Of the 250,000 int_vec.add_256, 125k are vector operations to initialize the loop and 12k are vector addition in the summation loop. For the float, 125k are to convert vectors of int to floats and 125k are to mov regs in memory. The 1,000,000 scalar are not vectorized at all. Hence, autovectorization works nicely for integers but not at all for the floats. This descends from the compiler not being entitled to reshuffle operations in the summation loop. For as we have written it, $s+ = a_i$, there is a clear critical path on s that must be respected by the compiler.

Observation: *Critical path*

From the basic course of HPC you should remember that when trying to optimize a loop, if we are accumulating on a single variable (S) it may lead to the execution of a lot of bit reshuffling and scalar add, in place of clean vector operations. To fix this, we can use partial accumulators:

```
1      for (int i=0; i < N; i+=VL) {
2          sum0 += array[i];
3          sum1 += array[i+1];
4          sum2 += array[i+2];
5          sum3 += array[i+3];
6          ...
7      }
8      sum = sum0 + sum1 + sum2 + sum3 + ...;
```

This exposes a natural vector pattern.

Now we can rewrite the loop differently, using multiple accumulators to break the critical path:

```
1      dtype sum_loop ( dtype * restrict array, const uint N )
2      {
3          double sum0 = 0;
4          double sum1 = 0;
5          double sum2 = 0;
6          double sum3 = 0;
7          uint N4 = N\%0xFFFFFFFC;
8          for ( uint32_t i = 0; i < N4; i+=4 ) {
```

```
            sum0 += array[i];
            sum1 += array[i+1];
            sum2 += array[i+2];
            sum3 += array[i+3];
        }
        ...
        return sum;
    }
```

This way we obtain:

	unroll vectorized	vectorized associative-math
metrics	500,000 vector 0 scalar	500,000 vector 0 scalar

Hence, re-engineering the code before using the auto-vectorization of the compiler can greatly enhance the achieved vectorization because we expose more parallelism while explicitly relaxing the order of the operations.

Example: *Another example*

Consider this loop:

```
    for (int i = 0; i < N; i++) {
        sum += A[i]*B[i] + C[i];
    }
```

This case has a higher possible parallelism since the reduction comes after the multiply-and-add vector operation. This means that a vector of partial results

```
    D[i:i+n] = A[i:i+n]xB[i:i+n] + C[i:i+n];
```

can be preemptively prepared, waiting for the entries to be summed up serially. The compiler in fact opts for this implementation.

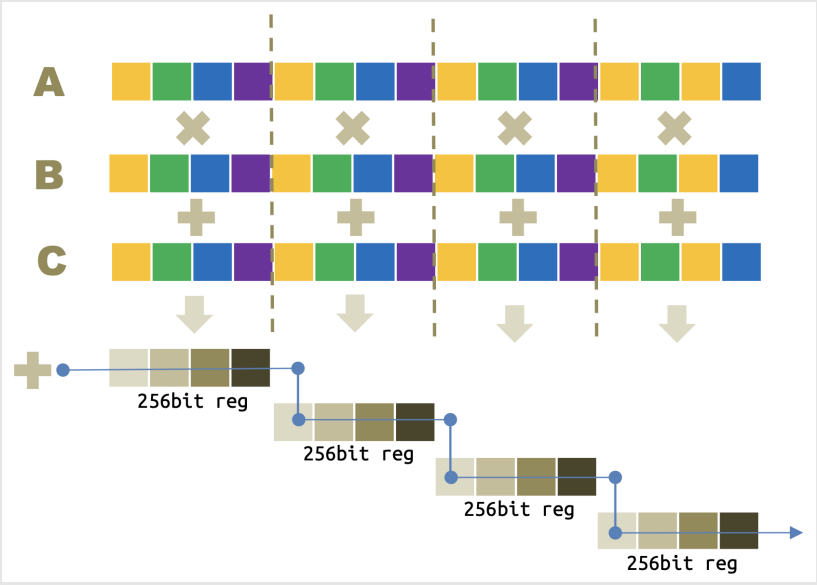


Figure 1.5: Auto-vectorization of a loop with multiply-and-add operation.

Lots of computation are performed via FMA on vector registers, 256b or 512b depending on what is available on the cpu. However, lots of reshuffling is needed and all the reduction summations are serial. Opting either for the usage of the `associative math` flag or for explicit loop unrolling (using separate accumulators) results in a different, fully vectorizable pattern.

So, to express the whole inherent loop-data and instruction-level parallelism is of primary importance to either

- authorizing the compiler to **re-shuffle floating-point** operations or
- **unroll the loops** to expose more instruction-level and data-level parallelism.

Tip:

Professor tip is to manually refactor the code, so that you can test for the correctness of the results while achieving the same vectorization.

The ideal situation is to have ***countable*** loops.

The iteration space must be known at run-time. The end of the loop can not depend on data. Following is an example of a non-countable loop:

```
1 int i = 0;
2 while (i < N) {
3     a[i] = b[i] * c[i];
4     if (a[i] > 0) {
5         break;
6     }
7     i++;
8 }
```

Tip: ***Countable While Loops***

Note that a countable while loop is vectorizable as a for loop is.

Moreover, there should not be branches, since the SIMD instructions are meant to perform exactly the same operation on multiple data and so different iterations can not have different control flows (if statements are admitted if they can result in a mask-like implementation). For the same reason, we should avoid function calls (vectorizable functions are an exception).

Sometimes it is essential to use specific compiler flags to enable or enhance vectorization of floating-point code, especially in the presence of math library calls or exception handling.

- **`-fno-math-errno`** : Tells the compiler to assume that math functions (like `sqrtf`, `logf`, etc.) do not set `errno`, and that your code will not check it. By disabling `errno` handling, the compiler is free to inline math functions or replace them with fast vector library calls, enabling vectorization even across calls to functions such as `libm`. This flag lifts a major legality blocker for SIMD.
- **`-fno-trapping-math`** : Informs the compiler that floating-point operations are not expected to raise traps (hardware FP exceptions) that your code will observe. This allows the compiler to speculate or execute floating-point operations in SIMD lanes that may not be used, enabling masked or predicated vectorization (e.g., by computing both sides of a branch and selecting the result).

By default, compilers like GCC have `-fmath-errno` and `-ftrapping-math` **enabled** (`ON`).

There are many situations where something could go wrong. The first case is when using **non-contiguous memory access**. It refers to loops that access data with a non-unit stride, or even worse with an irregular pattern. They are rarely vectorized, and that is because while consecutive data can be loaded in a single cache line, or in a register, with a single memory operation, sparse memory access need separate loads and data gathering; if the compiler estimates that this overhead is larger than the benefits, the loop is not vectorized. Another case is when there are **data dependencies** between iterations of the loop. If the compiler detects that there is a dependency, it can not vectorize the loop because it can not reshuffle the operations without changing the semantics of the code. For this case, we can identify:

- *Read-After-Write (flow-dependency)* $\rightarrow$ A variable is written in an iteration and read on a subsequent one. This loop can NOT be vectorized without leading to wrong results.
  However, let's consider the following 2D case:

```
1 for (int i = 0; i < N; i++) {
2     A[i][0] = C[i][0];
3     for (int j = 0; j < M; j++) {
4         A[i][j] = A[i][j-1] + C[i][j];
5     }
6 }
```

  the dependency is carried in the inner loop, while the outer loop iterations do not exhibit any dependency and the **outer loop vectorization** can be performed.

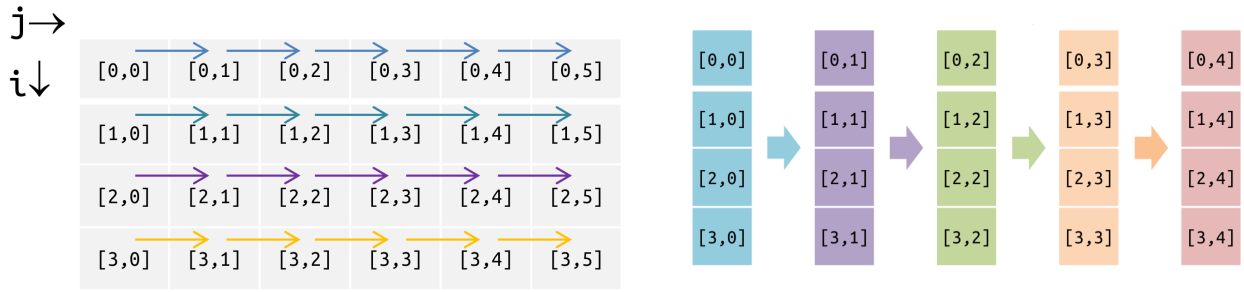


Figure 1.6: RAW data dependency: 4 (or 8) outer iterations can be executed together because there are no vertical dependencies (outer loop vectorization)

Consider the dependency graph of the iteration space. It clearly shows that there are no vertical dependencies. Hence, the outer loop can be vectorized: 4 (or 8) outer iterations can be executed together because there are no vertical dependencies. However, if the elements are stored in memory in row-major order that require a lot of memory gather and scatter operations. If not supported in hw, they must be emulated, which is even more expensive. Thus, we may consider to write the data column-major if possible.

- *Write-After-Read (anti-dependency)* → A variable is read in one iteration and written in a subsequent one. Unsafe for parallelism (no strict order among iterations assigned to tasks), normally safe for vectorization. Consider this example:

```
1 for (int i = 1; i < N; i++) {
2     a[i-1] = a[i] + c[i];
3 }
```

while the following cannot be vectorized because it's unclear if `a[i]` may be overwritten before the summation to `sum` :

```
1 for (int i = 0; i < N; i++) {
2     a[i-1] = a[i] + c[i];
3     sum += a[i];
4 }
```

- *Write-After-Write (output dependency)* → The same variable is written in more than one iteration. Generally unsafe both in parallelization and in vectorization.

```
1 double sum = 0;
2 for (int i = 1; i < N; i++) {
3     a[i-1] = function1(i);
4     a[i] = function2(i);
5 }
```

- *Memory ambiguity* → Typically due to memory aliasing. It may be impossible for the compiler to decide whether, for some value of `i, a[i], b[i]` and `c[i]` will somehow overlap. If that is the case, it will not issue a vector code. When it is possible, the compiler will issue more than one version of the loop and a code that performs an overlap test among the pointers. Then the execution will be routed to the correct version at run-time.

```
1  void function (int *a, int *b, int *c, int N) {
2      for (int i = 0; i < N; i++) {
3          c[i] = a[i] + b[i];
4      }
5  }
```

We can disambiguate using the `restrict` qualifier, which tells the compiler that the pointers do not alias each other. This allows the compiler to vectorize the loop without generating multiple versions and runtime checks. Also the `const` qualifier can help, as it tells the compiler that the data pointed to by the pointer will not be modified, which can also enable vectorization.

A variable is said to be **aligned** in memory when its first byte's address is a multiple of the variable's type size:

- `short int` $\rightarrow$ 2 bytes: 0, 2, 4, 6, 8, ... are aligned placements;
- `int / float` $\rightarrow$ 4 bytes: 0, 4, 8, 12, ... are aligned placements;
- `long long / double` $\rightarrow$ 8 bytes: 0, 8, 16, 24, ... are aligned placements.

Consequently, an aligned variable is also aligned w.r.t. the cache line it belongs to: it means that an aligned variable will never cross 2 cache lines! However, let's imagine that we allocate a bunch of memory and we consider it as an array of elements with byte size s_B. Let's say that:

- its starting byte is aligned at a_B bytes (a_B is a multiple of s_B);
- we access this array by vectors of element-size V_L (meaning that every vector has V_L elements); hence we will access $V_L \times s_B$ bits every time.

Then if

$$(a_B + V_L \times s_B) \% C_L == 0 \quad (C_L \text{ being the cache capacity in terms of elements})$$

all the vectors will stay in the same cache line and will require 1 load; otherwise, a vector will be mapped into two differnt cache lines and will require 2 loads.

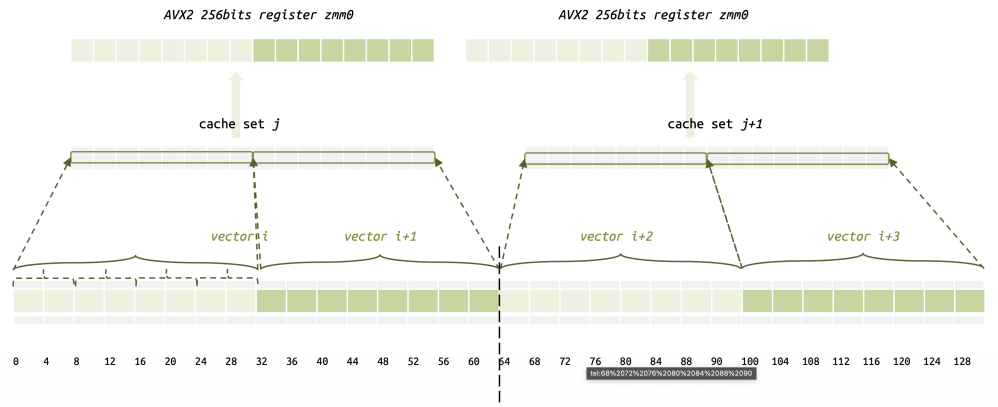


Figure 1.7: Example 1: an array of float aligned at 64 bits accessed as vector of size 4 (cache capacity is 16 with a std cache line of 64B) $a_B = 8B, V_L = 4, s_B = 4, C_L = 16$ every vector will occupy exactly 1 cache line.

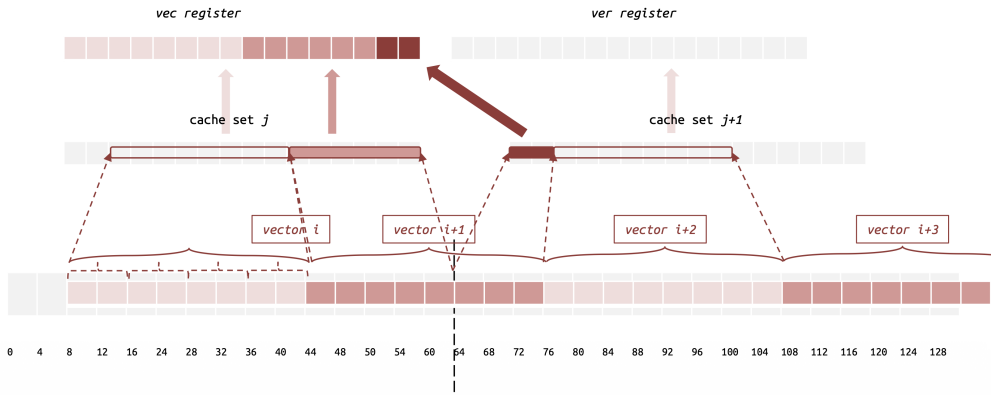


Figure 1.8: Example 2: an array of float aligned at 16 bits accessed as vector of size 4 (cache capacity is 16 with a std cache line of 64B) $a_B = 8B, V_L = 4, s_B = 4, C_L = 16$ every vector will occupy exactly 2 cache lines.

Tip: ***Data structures***

- Keep together data that are used together in vector operations;
- The smallest the data type you use, the larger the number of data that you can fit in the cache and in the vector registers;
- Do not mix same-type different-size data:

```
1 func ( float *A, float *B, double *C, int N ) {
2     for ( int i = 0; i < N; i++ ) {
3         A[i] = B[i] * C[i];
4     }
5 }
```

the type conversion renders things less efficient (`long double` type is not supported on GPUs).

Very commonly the data are multi-dimensional and hence the most natural representation is by a structure that encapsulates all the quantities for a single data point. Every entry of the array that contains all the data points is then a structure:

```
1 data[i].pos[3], data[i].acc[3], data[i].mass, ...
```

This is the **Array of Structures (AoS)** representation. and is a very convenient representation in many respects. However, when the structures collects many and diverse data, the SoA have several inconvenience too when computational performance is considered.

Let's consider the following example that occupies 96 bytes:

```
1 struct particle {
2         double pos[3];
3         double acc[3];
4         double mass;
5         ...
6     };
```

As a first consideration, even a single particle could not fit in a cache line. Second, most probably pos[3] will be used, likely with mass, to get accel[3] and then vel[3] will be updated.

Very likely every of these calculations may have a high degree of vectorization. However, there is no way in which those quantities can be loaded from memory to vector register efficiently without lots of overhead, either using gathering instructions or allocating additional memory and moving there only the variables used for every calculation.

So, a very first step consist in opting for **Structures of Arrays of (small)Structures** (SoAoS):

```
1 struct Ppos       { double pos[3], double mass; };  Ppos *mpositions;
2 struct Paccel     { double accel[3]; };             Paccel *
     accelerations;
3 struct Pvel       { double vel[3]; };               Pvel *velocities;
4 struct Pchemistry {float elements[n_el];};          Pelems *Pchemistry;
5                                                     long long int *PIds;
```

Clearly, that alleviates the aforementioned issues but it is still sub-optimal.

If, instead, we opt for a **Structure of Arrays** (SoA) approach:

```
1 double *x, *y, *z, *mass, *xacc, *yacc, *zacc, ...
2 long long *ids;
```

the possible vectorization is definitely higher. Data rearranged in SoA approach definitely would expose more, and more efficient, vectorization opportunities.

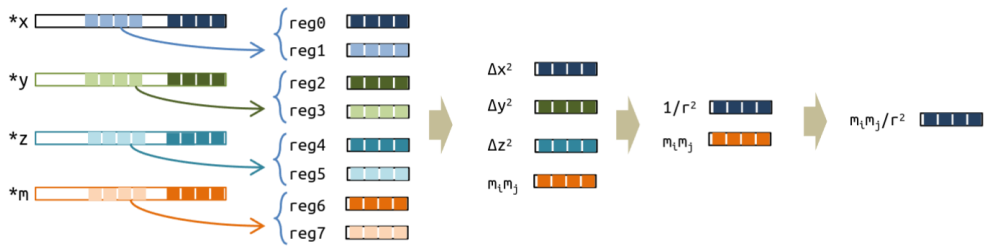


Figure 1.9: AoS vs SoA

Warning:

What is really convenient in every specific case may depend on several additional factors and that, in the context of this lecture, hinders more than just mentioning and explaining the vectorization advantage of SoA.

1.5 Vector Types

It is possible to use a vector type (not present in standard C, but commonly accepted by compilers) to declare a vector of a given type.

```
1 typedef type name __attribute__((vector_size,(bytes)));
```

where:

- **type**: the basic type for every element of the vector.

- **name**: the name with which you will refer to the type in your code.
- **bytes**: the number of bytes of the vector.

```
/* Declare a vector of 4 doubles */
typedef double v4d __attribute__((vector_size,(4*sizeof(double))));
/* Declare a vector of 8 doubles */
typedef double v8d __attribute__((vector_size,(8*sizeof(double))));
/* Declare a vector of 4 integers */
typedef int v4i __attribute__((vector_size,(4*sizeof(int))));
```

It is not possible to access the elements of a vector type directly, in the same way as you cannot access a single byte of a double. However, there is a standard way to do that:

```
typedef union {
    v4d v;
    double d[VD_SIZE];
} v4d_u;
```

1.5.1 Memory Alignment

It is mandatory that memory regions accessed by vector instructions are aligned (32B for AVX/AVX2 and 64B for AVX-512).

Loops translated by the compiler consist of three sections:

1. **Peeling loop** - scalar until aligned addresses are reached (→ avoided by proper alignment);
2. **Main body** - well vectorized from aligned addresses;
3. **Remainder loop** - scalar operations if the number of elements is not a multiple of the VL (→ maybe avoided by hinting the compiler like `__attribute__((__assume(N%4==0)))` or by accounting for the main body and the remainder explicitly).

To allocate memory in an aligned way, we can use C11 and POSIX functions:

- **C11**: `void *aligned_alloc(size_t alignment, size_t size);`
- **POSIX**: `void *posix_memalign(void **memptr, size_t alignment, size_t size);`

It is also possible to use a static allocation (i.e. variables or automatic arrays):

```
__attribute__((aligned(base))) <var>;
```

And it is also possible to assume that the memory is aligned:

```
assume_aligned(<array>, base);
```

Observation: *Unrolling loops by hands*

When unrolling a loop by hands, which you typically do in order to expose more parallelism as we have discussed previously, you also need to separately handle the main loop and the remind loop.

```
for (int i = 0; i < N; i++) ...
```

becomes:

```
int N4 = (N/4)*4;    // --> integer division ensures that N4 is the
                     // largest multiple of 4 smaller than N
for (int i = 0; i < N4; i+=4) ... // main body

for (int i = N4; i < N; i++) ... // remainder loop
```

You can give some additional hint to the compiler so that to avoid the issuing of additional codes to check the peeling. The line

```
int N4 = (N/4)*4;
```

can be replaced by

```
N4 = N&0xFFFFFFFC;  // --> use 0xFFFFFFF8 for 8
                    // --> use 0xFFFFFFFFFFFFFFF<C|8> for 64 bits
                       integers
```

this will hint the compiler that N4 is a multiple of 4, without need of issuing additional code. Alternatively,

```
N4 __attribute((__assume(N4\%4==0)));
```

may also work on some compilers.

Example: *Working with vectors*

Let's re-implement the following loop using the vector types and check what changes in the generated assembler code and in the run-time.

```
double kernel( double *restrict A, double *restrict B, double *
    restrict C, int N ) {
     double sum = 0;
     for ( int i = 0; i < N; i++ )
         sum += A[i]*B[i] + C[i];
     return sum;
}
```

At first we need to define the appropriate vector variables:

```
7  #define VD_SIZE 4
8  typedef double v4df __attribute__((vector_size(VD_SIZE*sizeof(
       double))));
9  typedef union {
10     v4df v;
11     double d[VD_SIZE];
12 } v4df_u;
13
14 v4df *VA = (v4df *) A;
15 v4df *VB = (v4df *) B;
16 v4df *VC = (v4df *) C;
17
18 v4df vsum = {0};
```

Now we can actually implement the sum:

```
19 int N4 = N&0xFFFFFFFC;
20
21 for (int i = 0; i < N4; i++) {
22     vsum += VA[i] * VB[i] + VC[i];
23 }
24
25 v4df_u *vsum_u = (v4df_u *) &vsum;
26 vsum_u->v[0] += vsum_u->v[1] + (vsum_u->v[2] + vsum_u->v[3]);
27
28 for (int i = N4; i < N; i++) {
29     sum += A[i]*B[i] + C[i];
30 }
31
32 sum += vsum_u->v[0];
33
34 return sum;
```

1.5.2 Conditional evaluation

When our code contains conditional statements, we need to pay special attention if we want to use vector types. Vectorization is usually only possible if the condition is simple and the code inside the conditional is not too complex, for example a single operation or a small sequence of straightforward instructions.

```
1 void kernel( double * restrict A, double * restrict B, const int N ) {
2     for ( int i = 0; i < N; i++ )
3         if ( A[i] < B[i] ) swap(A, B);
4     return;
5 }
```

Actually this code can be easily fully vectorized by the compiler itself with the appropriate compilation flags. The idea is to create a mask of the condition and then use it to select the appropriate values.

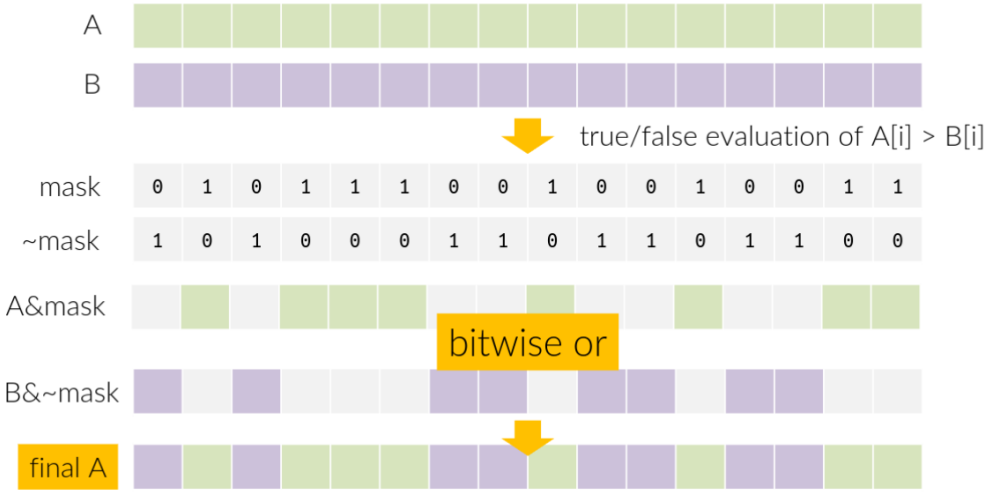


Figure 1.10: Conditional evaluation [1]

Tip:

The concept of vectorized conditional flow can really be thought as ”projecting” througha mask so that the final values that are ”impressed” in the memory positions pass through the ”transparent” entries and are blocked by ”opaque” ones. Many projected values could of course overlap by using multiple masks combined with bit-wise operators.

To efficiently handle conditionals in vectorized code, we first implement the mask logic. The ternary operator is particularly well-suited for expressing such element-wise conditional assignments:

```
1 void kernel( double *A, double *B, int N ) {
2     // element-wise swap of A and B so that
3     // A[i] > B[i] for every i
4
5     for ( int i = 0; i < N; i++ ) {
6         double min = (A[i]>B[i] ? B[i] : A[i]);
7         double max = (A[i]>=B[i] ? A[i] : B[i]);
8         A[i] = max;
```

```
9           B[i] = min;
10      }
11      return;
12  }
```

Now we can vectorize the code using the vector types and the mask logic.

```
1  dvector_t *vA = (dvector_t *)__builtin_assume_aligned(A, VALIGN);
2  dvector_t *vB = (dvector_t *)__builtin_assume_aligned(B, VALIGN);
3
4  int VN = (N/DVSIZE)&0xFFFFFFFC;
5  IVDEP
6  LOOP_VECTORIZE
7  LOOP_UNROLL_N(4)
8  for ( int i = 0; i < VN; i++ ) {
9      dvector_t a = vA[i];
10     dvector_t b = vB[i];
11     llvector_t keep = (vA[i]>=vB[i]);
12     vA[i] = (dvector_t)(((llvector_t)a & keep) |
13             ((llvector_t)b & ~keep));
14     vB[i] = (dvector_t)(((llvector_t)b & keep) |
15             ((llvector_t)a & ~keep));
16 }
17
18 int j = VN*DVSIZE;
19 process ( &A[j], &B[j], N-j+1);
```

TODO: Add second version and possibly comment

1.6 Vectorization by OpenMP SIMD directive

The OpenMP compiler may transform a loop that is marked with the simd construct into a SIMD loop. As a result, multiple iterations of the loop may be executed concurrently by a single thread. A SIMD loop of length n consists of the logical iterations $0, 1, \ldots, n-1$. The numbering denotes the sequential order in which loop iterations are executed.

A SIMD *chunk* refers to the set of iterations that are executed concurrently by a SIMD instruction in a single thread. Within a chunk, each iteration is executed by a SIMD *lane*, which refers to the mechanism that a SIMD instruction uses to process one data element. The number of iterations in a SIMD chunk is the vector length.

The OpenMP specification describes the execution model of the simd construct as follows: ”When an OpenMP thread encounters a `simd` construct, the iterations of the loop asociated with the construct may be executed concurrently using the SIMD lanes that are available to the thread.” Intentionally, this allows for much freedom in the implementation.

```
1  #pragma omp simd [clause [[,] clause] ...]
```

The clauses are:

- `private(list)` and `lastprivate(list)`: they have exatly the same meaning as in “thread-related” OpenMP. The execution instances in this context are the SIMD lanes; hence, an instance of every private variables is created per SIMD lane.

```c
int i;
double tmp, sum=0;

#pragma omp simd private(tmp) reduction(+:sum)
for (int i = 0; i < N; i++) {
    tmp = function(b[i]);
    array[i] = tmp;
    sum += tmp;
}

// NOTE: i is lastprivate
```

- `reduction(identifier: list)` and `collapse(n)`: they also retain their usual meaning and usage.

```c
int i;
double sum=0;

#pragma omp simd reduction(+:sum)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        tmp = function(a[i], b[j]);
        array[i] = tmp;
        sum += tmp;
    }
}
```

- `simdlen(length)`: It has the goal of suggesting the element-size of the vector lane to be adopted, i.e., the count of elements in the vector.

```c
int i;
double tmp;

#pragma omp simd simdlen(32/sizeof(double))
for (int i = 0; i < N; i++) {
    tmp = function(b[i]);
    array[i] = tmp;
}

// NOTE: i is lastprivate
```

- `safelen(length)`: It is used to specify at the compiler the maximum length not having dependences inside the loop. In the following example, we instruct the compiler that 8 is a safe length for vectorization;

```c
#pragma omp simd safelen(8)
for (int i = 0; i < N; i++) {
    a[i] = pow(b[i-k], 3.14); // here the compiler does not know a
       priori
}
```

- `linear(list[: linear-step])`;
- `aligned(list[ : alignment ])`;

1.6.1 The omp SIMD functions

In general, every non-linear element in the execution flow is at high risk a disruption in the vectorization. Function call above all, but also, for instance, conditionals.

OpenMP SIMD offers the possibility of automatically build vector version of code segments through the `declare simd` directive. These functions can then be called from a simd loop.

```
1  #pragma omp declare simd
2  double foo (double x, double y, double z) {
3      double res = x*y + z;
4      return res;
5  }
6
7  void loop (double *a, double *b, double *c, int N){
8      #pragma omp simd
9      for (int i = 0; i < N; i+=4) {
10         a[i] = foo(a[i], b[i], c[i]);
11     }
12 }
```

The `declare simd` directive instructs the compiler to generate at least one additional simd version of the function `foo` because it will be called from a simd loop. That variant will be called from inside the simd loop in the function `loop`.

However, when declaring `foo`, can we give the compiler more details about how x, y and z are changing? In this form, the best assumption is that they must be incremented by 1 in the simd version.

The possible clauses are:

- `uniform`: when listed as uniform, a parameter is intended to be invariant for all the concurrent calls to the function, i.e. all the simd lanes will observe the same value
- `linear`: at odds, being linear means that a parameter changes linearly with the indicated step
- `simdlen`: retains the same meaning as in the simd directive
- `aligned`: as exactly the meaning than in normal code
- `inbranch`: informs the compiler that the function will be called from in a conditional branch.
- `notinbranch`: informs the compiler that the function will not be called from in a conditional branch.

TODO: add examples

1.6.2 The omp SIMD conditionals

OpenMP simd offers two additional clauses that work as assertions: `inbranch` and `notinbranch`. The former informs the compiler that the function will be called from within a conditional branch, whereas the last guarantees the compiler that this will not happen. The compiler can then opt for very different optimizations and how to manage the vectorization.

```
1  #pragma omp declare simd inbranch
2  float do_mult(float x) {
3      return (-2.0*x);
4  }
```

```
5
6  #pragma omp declare simd notinbranch
7  extern float do_pow(float);
8
9  void simd_loop_with_branch(float *a, float *b, int n) {
10     #pragma omp simd
11     for (int i = 0; i < n; i++) {
12         if (a[i] < 0.0) {
13             b[i] = do_mult(a[i]);
14         }
15         b[i] = do_pow(b[i]);
16     } // end simd region
17 }
```

Finally, multiple simd versions of the same function can be generated with multiple declarations:

- `#pragma omp declare simd linear(pixel) uniform(mask) inbranch`;
- `#pragma omp declare simd linear(pixel) notinbranch`;
- `#pragma omp declare simd`;
- `extern void compute_pixel(char *pixel, char mask)`;

2 Concurrency

2.1 Concurrency vs Parallelism

Up to now we have looked inside a single core, and asked the compiler to do as much work in parallel as the vector units allow. Vectorization, however, is just the most local form of parallelism we can express in our code. The next step is to consider what happens when we want *several streams of execution* to make progress on the same machine, possibly on different cores.

Two terms are often used interchangeably here, but they refer to different ideas:

- **Concurrency** is the ability to carry out two or more actions ***in progress at the same time***. It does *not* require multiple cores: a single core can interleave several tasks by repeatedly suspending one and resuming another (***context switching***).
- **Parallelism** is the ability to ***execute*** two or more actions at the very same instant. This requires distinct execution units (cores, hardware threads, vector lanes, GPU SMs, …).

A short slogan captures the difference: *concurrency is about structure, parallelism is about execution*. A concurrent program describes a set of activities that can in principle make progress independently; whether they actually do so simultaneously depends on the hardware that executes the program. The same concurrent program can run serially on a single core (with the OS scheduler interleaving the tasks) or in parallel on multiple cores.

There are two main forms in which concurrency is exposed in a program:

- **Multithreading** (lower level): several ***threads*** are spawned, and they run concurrently (in parallel if possible) on the CPUs at our disposal. Synchronization among threads is usually required to avoid ***data races*** and ***deadlocks***.
- **Asynchronous programming** (higher level): tasks are started without waiting for the previous one to complete. They can run in the background, possibly on other threads, but the user does not need to manage these details. Typical implementations rely on ***callbacks***, ***coroutines***, ***promises*** and ***futures***.

In this chapter we focus on the first form: explicit, low-level multithreading using POSIX threads.

⚠ Warning: ***The sorrows of the young multithreader***

Managing multiple threads requires ***synchronization*** among them. Wrong synchronization can cause race conditions, deadlocks and other undefined behaviors. The unpleasant property of these bugs is that they may manifest *very* infrequently: a logically possible race that occurs only 0.01% of the times will not show up during testing, but will eventually fire in production. The historical example is the Therac-25 radiation therapy machine, where a race condition in the operator's terminal interface caused several patients to receive massive radiation overdoses. The lesson is uncomfortable but firm: if a data race is *logically possible*, even if unlikely, it **must** be avoided by protecting shared resources.

2.2 Processes vs Threads

Both processes and threads can be used to achieve concurrency, but they differ in how they share state.

A **process** is the unit through which the operating system isolates running programs. Each process has its own ***virtual address space***, file descriptors, and accounting. Inside that address space the kernel sets up a layout that, in Linux, looks roughly like:

- the **text** segment (program code);
- the **data** and **bss** segments (initialized and uninitialized globals);
- the **heap**, growing upwards via `brk` / `mmap` ;
- one **stack** per thread, growing downwards;
- shared libraries and shared memory.

A **thread** is an independent unit of execution *inside* a process. All threads of the same process share the text, data, heap, and open files; what is private to a thread are its CPU registers and its own stack. The first thread of a process is the ***main thread***; additional threads can be spawned at runtime.

Observation: ***Why the difference matters***

Sharing the address space makes thread-to-thread communication essentially free: a thread can hand a pointer to another and they both see the same memory. The price is that any unsynchronized access to that shared memory becomes a potential data race. Processes pay the opposite trade-off: they are isolated, so communication needs an explicit ***Inter-Process Communication*** (IPC) mechanism (pipes, sockets, message queues, shared memory regions), but unsynchronized access is impossible by construction.

2.2.1 Forking a process

A new process is created on POSIX systems with the `fork()` system call. The kernel duplicates the calling process; the call returns once in the parent (with the child's PID) and once in the child (with 0). Code accordingly switches on the returned value:

```c
#include <unistd.h>
#include <sys/types.h>

pid_t pid = fork();
switch (pid) {
case -1:                    // something bad happened
    perror("fork");
    return EXIT_FAILURE;
case 0:                     // child process
    printf("CHILD\n\tPID: %ld, PPID: %ld\n",
           (long)getpid(), (long)getppid());
    break;
default:                    // parent process
    printf("PARENT\n\tPID: %ld, PPID: %ld\n",
           (long)getpid(), (long)getppid());
    break;
}
```

Every process has a Process ID (`getpid()`) and a parent PID (`getppid()`). The virtual address

space of the original process is ***copied-on-write***: as long as both parent and child only *read* the same memory pages, no actual copy occurs and they keep referring to the same physical frames. The first time either side writes to a page, the kernel transparently duplicates it so that the modification is visible only in the writer.

The mechanism is general but it is also ***relatively heavy***: there is at least the cost of cloning page tables and accounting structures, and any subsequent communication has to go through one of the IPC primitives. This is why, for fine-grained concurrent computation on a single node, threads are usually preferred to processes.

Example: *Code references*

The `Concurrency/code/1-Processed-and-Threads/` directory contains three minimal programs to compare on a live system:

- `1-Single-Process.c` – one process, one thread.
- `2-Fork.c` – a parent that `fork`s into a child, both calling `sleep`.
- `3-Multiple-Threads.c` – a single process with two threads.

Run `make all` in that directory, launch each binary in the background, and inspect them with `ps -efT`.

Tip: *Listing processes and threads*

The command

```
ps -efT
```

lists every process (PID) together with each of its threads (TID, also reported as SPID). To restrict to a specific process,

```
ps -fT -p <PID>
```

shows the threads of the chosen PID. In `top` and `htop` (with the right toggles) the column labelled “PID” actually shows the *thread* ID, which is sometimes confusing the first time you see it.

2.3 The pthread library

We will use the **Pthread** library (POSIX THREAD), which is the de facto standard threading interface on UNIX-like systems (Linux, BSD, macOS, …). It is *not* available natively on Windows, where the equivalent are the Win32 threading primitives.

For portable code there are two alternatives:

- In C++ (11 and later), threads live in the standard library through the `<thread>` header. The C++ standard also includes higher-level paradigms such as `std::async`, `std::future` and `std::promise`.
- In C (11 and later) the `<threads.h>` header offers a similar but smaller API. It is very close to pthread in spirit, but for a long time it was not implemented on Windows and there is much less documentation about it in the wild.

A few practical remarks on Pthread before we start writing code:

- **Unusual return convention.** A pthread function `pthread_<…>` returns an `int` : 0 on success, a *positive* `errno` -like value on failure. The global `errno` is *not* set, so the usual idiom `if(call() < 0) perror(…)` is wrong here.
- **Compile flag.** Compile with `-pthread` : this links `libpthread` and defines the `_REENTRANT` macro that exposes the reentrant variants of the standard C library functions.

> **Tip**: ***A must-have reference***
>
> For everything that involves the Linux kernel's API surface (system calls, processes, threads, signals, IPC, file I/O, …) the canonical reference is M. Kerrisk, *The Linux Programming Interface*. Whenever you are unsure about the contract of a syscall, `man 2 <name>` or `man 3 <name>` is the second-best option.

2.4 Creating and joining threads

A thread is created with

```c
#include <pthread.h>

int pthread_create( pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start_routine)(void *),
                    void *arg );
```

- `thread` is filled with the thread ID; the type `pthread_t` is implementation-dependent and must be treated as opaque.
- `attr` can be `NULL` to use the default attributes.
- `start_routine` is the entry point of the new thread.
- `arg` is the (single) argument passed to it; `NULL` is allowed.

The call returns *immediately* after enqueuing the thread for execution, so the spawning thread keeps running in parallel with the new one.

```c
void *thread_function(void *unused) {
    (void)unused;
    printf("Child thread\n");
    return NULL;
}

int main(void) {
    printf("Main thread\n");
    pthread_t thread;
    if (pthread_create(&thread, NULL, thread_function, NULL)) {
        fprintf(stderr, "pthread_create failure.\n");
        return EXIT_FAILURE;
    }
    sleep(1);           // poor-man synchronization
    return EXIT_SUCCESS;
}
```

This is the program `Concurrency/code/2-Create-and-Join/1-no-Join.c` . The `sleep(1)` is

a hack: without it, the main thread might `return` from `main` before the child has even printed, terminating the whole process. The proper way to wait for a thread is `pthread_join` :

```
1 int pthread_join(pthread_t thread, void **retval);
```

It blocks the caller until `thread` terminates and, if `retval` is non- `NULL` , stores the value the thread returned. Replacing the `sleep` with `pthread_join(thread, NULL)` (`2-Master-Join.c`) gives a deterministic program with no arbitrary delays.

Observation: *Any thread, any thread*

`pthread_join` is symmetric: any thread of the process can join any other thread, not only the one that created it. The example `3-Multiple-Threads.c` uses this property: thread 1 creates thread 2, and the main thread later joins both of them through their handles.

2.4.1 Detaching a thread

If the result of a thread is not needed and we do not want to bother joining it, we can mark the thread as ***detached***. The system then reclaims its resources automatically once it terminates, and the thread becomes no longer joinable.

```
1 int  pthread_detach(pthread_t thread);
2 pthread_t pthread_self(void);
```

A common idiom is to call `pthread_detach(pthread_self())` as the first line of the thread function (`4-Detach.c`). After that the spawning thread does not need to track the handle.

2.4.2 Passing input and returning output

Both the argument of the thread function and its return value have type `void *` : the library erases the type and we have to put it back. The most common patterns are:

- **Heap-allocated argument.** The spawning thread `malloc` s a struct, fills it, and passes the pointer (cast to `void *`). The thread function casts back to the original type. Memory ownership has to be agreed on: either the spawning thread frees it after the join, or the thread itself frees it before returning.
- **Global variable.** Convenient for a one-off but a poor habit: it ties the thread function to a single global, breaks reentrancy, and quickly turns into a synchronization headache.
- **Type punning.** As long as `sizeof(T) ≤ sizeof(void *)` , one can squeeze a small integer or pointer-sized handle into the `void *` slot. This is widely supported but *not* strictly compliant with ISO C, so avoid it in portable code.

The return value follows symmetric conventions:

- return a pointer to a heap object (the joiner is responsible for freeing it);
- return a small error code, with the same caveats as above;
- return `NULL` and use a different channel to communicate.

A function that terminates the current thread without falling off the end of the entry function is `pthread_exit(void *status)` ; it can be called from any function, not only from the thread's entry point.

Example: *Two flavours of input/output*

The example `Concurrency/code/2-Create-and-Join/5-Input-Output.c` computes the integer division of `numerator` by `denominator`. The thread receives the operands on the heap and *itself allocates* the output, returning a pointer:

```c
struct thread_input  { unsigned int numerator, denominator; };
struct thread_output { unsigned int quotient,  reminder;    };

void *thread_function(void *arg) {
    struct thread_input  *input  = arg;
    struct thread_output *output = malloc(sizeof *output);
    if (!output) return NULL;
    output->quotient = input->numerator / input->denominator;
    output->reminder = input->numerator % input->denominator;
    return output;
}
```

The joiner then has to remember to `free(output)`. A cleaner variant is `6-Input-Output-better.c`: the input struct *contains* a pointer to the output, so the caller owns both allocations from the start, and the thread function only fills the result and returns `NULL`. Ownership is now obvious and there is no risk of a forgotten `free`.

2.5 Synchronization issues and the data race

So far our threads have not touched any shared data. The moment they do, things become subtle. Consider the following innocent-looking program:

```c
int counter = 0;              // global, shared variable

void *threadfunc(void *unused) {
    (void)unused;
    ++counter;
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, threadfunc, NULL);
    pthread_create(&t2, NULL, threadfunc, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("counter = %d\n", counter);
}
```

We naively expect `counter == 2`. In reality the program may also print 1, and reproducing it is hard. To see why, compile the function with `gcc -S -fverbose-asm -g` and look at the body of `++counter`:

```
movl    counter(%rip), %eax        ; load
addl    $1, %eax                   ; increment in register
movl    %eax, counter(%rip)        ; store
```

The single C statement `++counter` is three machine instructions. If the OS preempts thread 1 between `load` and `store`, thread 2 may load the same old value, increment its own register, and store it back; when thread 1 resumes it overwrites the result with a value that is also wrong by one. We have just exhibited a ***data race***: two threads access the same memory location without synchronization, at least one of them writes, and the outcome depends on which goes first.

⚠ Warning: *A dangerous situation*

Even if the timing window for the race is tiny – say 0.01% of the runs – the program is **not** safe. "It works on my machine" is not a property of correctness; it is the result of an arbitrary scheduling that the OS is free to change tomorrow. As soon as a data race is logically possible, the shared resource *must* be protected.

There are two complementary tools to fix this: ***atomic variables***, which make small operations indivisible, and ***mutexes***, which make arbitrary critical sections mutually exclusive.

2.6 Atomic variables

C11 introduced the `_Atomic` type qualifier and the `<stdatomic.h>` header. Declaring a variable as

```
_Atomic int counter;
```

makes every *individual* read, write, increment, decrement, and compound assignment on it atomic with respect to other threads. The compiler emits the appropriate locked instructions (`lock add`, `cmpxchg`, …) so that no thread can ever observe a half-updated value.

A few important caveats:

- Any type other than function and array types can be made `_Atomic`.
- *Not* every `_Atomic` type is guaranteed to be ***lock-free***: for types that the hardware does not support natively (large structs, for instance) the implementation may fall back to an internal mutex.
- Atomic operations introduce ***synchronization*** between threads, with five memory orders defined by the C memory model. The default, `memory_order_seq_cst`, gives sequential consistency: every thread sees a single global order of all sequentially-consistent operations.
- **Combinations of atomic operations are not themselves atomic.** If the critical section consists of more than one read–modify–write – "test and then set", "compare two atomics", "increment if positive" – a single `_Atomic` variable is not enough; you need a mutex or a higher-level primitive.

In our broken counter example, replacing `int counter` with `_Atomic int counter` is enough, because the only operation is `++counter`.

2.7 Mutexes

A **mutex** (***mut***ual ***ex***clusion) is a synchronization object with two states, ***locked*** or ***unlocked***. The contract is:

- Any thread can attempt to lock the mutex *before* touching the resource it protects.
  - If the mutex is unlocked, the call locks it and returns immediately.

  - If the mutex is already locked by another thread, the caller is suspended until the mutex is released by its owner.
- The thread that locked the mutex unlocks it after having finished with the resource. Other threads waiting on the mutex can now acquire it.

Using a mutex is straightforward:

```
1  int counter = 0;
2  pthread_mutex_t counter_mtx = PTHREAD_MUTEX_INITIALIZER;
3
4  void *threadfunc(void *unused) {
5      (void)unused;
6      pthread_mutex_lock(&counter_mtx);
7      ++counter;
8      pthread_mutex_unlock(&counter_mtx);
9      return NULL;
10 }
```

Now if thread 2 reaches `pthread_mutex_lock` while thread 1 holds the mutex, thread 2 is blocked there. When thread 1 calls `pthread_mutex_unlock` , thread 2 acquires the mutex and proceeds. The increment is sequentialised even though the original instruction sequence is not atomic.

Any other use is ***undefined behaviour***: locking the same mutex twice from the same thread, unlocking a mutex one does not own, destroying a locked mutex, … A subset of these mistakes can be checked at runtime by switching to a non-default mutex type (see below). *Recursive* mutexes are an explicit exception: they may be acquired multiple times by the same thread, provided they are released the same number of times.

Tip: ***Protect every access***

A mutex is just a convention. It is *up to you* to wrap *every* access to the resource with the corresponding lock/unlock pair: if you forget a single read site, the data race is back. A common discipline is to keep the mutex and the resource it protects together in the same struct, and to provide accessor functions that are the only legal way to touch the resource.

2.7.1 Variants of the lock call

The pthread library exposes three variants of the lock call:

- **Blocking**: the standard `pthread_mutex_lock` / `_unlock` pair shown above.
- **Time-blocking**: `pthread_mutex_timedlock(mtx, abstime)` blocks until either the lock is acquired or the absolute time `abstime` is reached, in which case it returns `ETIMEDOUT` . It requires `<time.h>` .
- **Non-blocking**: `pthread_mutex_trylock(mtx)` attempts to acquire the lock; if it is already held it returns `EBUSY` immediately, leaving the caller free to do other work or retry later.

2.7.2 Mutex vs spinlock

A natural question is what the CPU does while waiting for a mutex. The answer is reassuring: *nothing*. With `top` you can see that the CPU consumption of a thread blocked on a mutex is negligible. On Linux, mutexes are implemented in terms of ***atomic variables*** and a kernel subsystem called ***futex*** (fast userspace mutex) that handles the suspension and the wake-up of contended

threads via context switching. The fast path – the uncontended case – stays entirely in user space and costs only a few atomic instructions; the slow path crosses into the kernel and pays roughly ~ 100 cycles.

A **spinlock** is semantically similar – it is a binary lock you acquire and release – but the implementation is the opposite: the waiter loops on an atomic test-and-set until the lock becomes available, which is a form of ***busy waiting***. With `top` the thread shows as $\sim 100\%$ CPU. Spinlocks are lighter than mutexes per acquisition (just an atomic in a tight loop, no syscall) but burn a core while waiting; they are only sensible when the critical section is so short that going to sleep would cost more than spinning.

2.7.3 Mutex attributes and dynamic initialization

The default mutex type is `PTHREAD_MUTEX_NORMAL`, which is the fastest and the strictest. There are two other useful types:

- `PTHREAD_MUTEX_ERRORCHECK` adds runtime checks: locking the same mutex twice from the same thread, or unlocking a mutex one does not own, returns an error instead of producing undefined behaviour.
- `PTHREAD_MUTEX_RECURSIVE` allows the same thread to lock the mutex multiple times. The mutex is fully released only after the same number of unlocks.

These attributed mutexes are slower than the default. Whenever a custom attribute is used, or when the mutex is allocated on the heap, the static `PTHREAD_MUTEX_INITIALIZER` cannot be used; we initialize and destroy the mutex explicitly:

```
int pthread_mutex_init   (pthread_mutex_t *mtx, const
    pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mtx);
```

2.8 Deadlocks

Once we have to lock more than one mutex in the same critical section, a new failure mode appears.

```
1 // Thread 1                    // Thread 2
2 pthread_mutex_lock(&first);    pthread_mutex_lock(&second);
3 pthread_mutex_lock(&second);   pthread_mutex_lock(&first);
```

If thread 1 acquires `first` just before thread 2 acquires `second`, both then try to lock the mutex held by the other and block forever. This is a **deadlock**: a circular wait in which no thread can make progress.

Warning: ***Deadlocks are hard to spot***

The same source code may run for a long time without ever deadlocking, because the bug appears only in certain interleavings. The program is still buggy, just lucky. A standard trick to expose the race is to insert a small `sleep` between the two locks and run the program many times: the artificial delay widens the dangerous window and the deadlock becomes essentially deterministic.

2.8.1 How not to fix it

A tempting non-solution is to replace `pthread_mutex_lock` with `pthread_mutex_timedlock`: now no thread is stuck forever. Unfortunately, all that we have done is to convert the deadlock into a ***livelock***: both threads time out, retry, and possibly time out again, burning CPU without making progress. The bug is still there, only better disguised.

2.8.2 Mutex hierarchy

The cleanest fix is to define a ***global ordering*** on the mutexes and to require that, whenever more than one is needed, every thread acquires them in that order:

```
// Both threads acquire `first` before `second`
pthread_mutex_lock(&first);
pthread_mutex_lock(&second);
//      ... critical section ...
pthread_mutex_unlock(&second);
pthread_mutex_unlock(&first);
```

If thread 2 also locks `first` before `second`, the cycle in the wait graph is broken: whichever thread acquires `first` first will then acquire `second` as well, while the other simply waits at `first`. The order has to be the same project-wide; for that reason this strategy is sometimes called the *lock hierarchy*.

2.8.3 Try-and-back-off

A more flexible, slightly less efficient alternative is to use `trylock` to detect a potential deadlock and to back off when one is brewing:

```
// Thread 1
for (;;) {
    pthread_mutex_lock(&first);
    if (pthread_mutex_trylock(&second)) {
        break;
    }
}
```

```
// Thread 2
for (;;) {
    pthread_mutex_lock(&
    if (
        sched_yield();
    } else {
        break;
    }
}
```

Each thread holds its first mutex and *tries* to acquire the second; on failure it releases the first one and calls `sched_yield` to let the OS reschedule. Because no thread holds a lock during the back-off, no other thread can be stuck behind it, and the deadlock dissolves. The code is more involved and incurs more lock/unlock pairs in the contended case, but it does not require a global ordering.

Observation: *A familiar lesson*

The first strategy fixes deadlocks by encoding a global structure in the code; the second one fixes them by reacting at runtime to local information. We will see this same trade-off in OpenMP (static vs dynamic scheduling) and in MPI (collective ordering vs one-sided communication): expressing the structure of the problem to the runtime usually wins on performance, while reacting locally usually wins on flexibility.

2.9 Thread safety and reentrancy

Two related but distinct properties of a function come up constantly in concurrent code.

POSIX defines a hierarchy of ***safety guarantees***:

- **MT-Safe**: the function can be called from multiple threads simultaneously without corrupting its internal state. Examples: `printf` , `malloc` .
- **AS-Safe**: the function can additionally be called from an asynchronous signal handler. Examples: `getpid` , `getppid` .
- **AC-Safe**: the function can be called during asynchronous cancellation.

A **reentrant** function is one that can be safely re-entered while a previous invocation is still in progress. Two conditions are necessary:

- it does not act on global or static state;
- it does not use a mutex (a mutex would deadlock the same thread re-entering through a signal handler).

Every reentrant function is MT-safe, but the converse is false: many functions are made thread-safe by acquiring an internal lock and so are *not* reentrant. The classical example is `strtok` (not reentrant, because it keeps state in a static variable), versus its reentrant counterpart `strtok_r` , which makes the state explicit through a parameter.

2.10 The producer–consumer model

Mutexes and condition variables are the building blocks of the standard **producer–consumer** pattern, which we will use as a running example.

In its simplest form there is a ***producer*** that generates data items and a ***consumer*** that processes them. They communicate through a ***channel***, typically a buffer, and the consumer should pick up data *as soon as* it appears, without leaving the producer waiting indefinitely.

The natural questions are:

- Is there a speed constraint between producer and consumer? In general no: either may be temporarily faster than the other, and the buffer absorbs the difference.
- How is the channel implemented? A first-in-first-out queue, with a fixed capacity, fits both ends naturally.
- How are producer and consumer synchronized? Through a mutex protecting the buffer, plus a way to make the consumer sleep when the buffer is empty (and the producer when it is full).

The model generalizes immediately to multiple producers and multiple consumers all sharing the same buffer. The race conditions multiply – two producers may try to push concurrently, two consumers may try to pop the same item – so the synchronization must be in place from the start.

Example: *Circular buffer*

The canonical bounded channel is a ***circular buffer***: a fixed-size array with two indices, a `head` where the producer writes and a `tail` from which the consumer reads, both wrapping around modulo the capacity. A minimal interface looks like:

```
struct circbuf;

void   cb_init    (struct circbuf *cb, void *buf,
                   size_t el_size, size_t capacity);
void   cb_unset   (struct circbuf *cb);
size_t cb_size    (const struct circbuf *cb);
size_t cb_capacity(const struct circbuf *cb);
bool   cb_push    (struct circbuf *cb, const void *el);
bool   cb_pop     (struct circbuf *cb,       void *el);
```

`cb_push` returns `false` when the buffer is full and `cb_pop` returns `false` when it is empty. With this skeleton in mind, the next section discusses how the consumer should wait until `cb_pop` can succeed.

2.11 Condition variables

Suppose the consumer wants to wait until the buffer is non-empty. Using only mutexes, the natural attempt is:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

while (!done) {
    pthread_mutex_lock(&mtx);
    // ... try to do the job, give up if nothing to do ...
    pthread_mutex_unlock(&mtx);
}
```

This is ***busy waiting***: the consumer wakes up, locks, checks, unlocks, and tries again, burning a CPU core just to find out that nothing has changed. The point of a condition variable is to put the consumer to sleep *efficiently* until the producer says "check again".

2.11.1 Pause and notify

A **condition variable** is a synchronization object that always comes paired with a mutex. It exposes two families of operations:

- **Pause** (`pthread_cond_wait` , `pthread_cond_timedwait`): atomically releases the associated mutex and puts the calling thread to sleep until it is notified. When the thread wakes up, the mutex has been re-acquired before the function returns.
- **Notify** (`pthread_cond_signal` , `pthread_cond_broadcast`): wakes up one (`signal`) or all (`broadcast`) of the threads waiting on the condition variable.

A first attempt at the producer/consumer synchronization could be:

```
// Consumer                                  // Producer
pthread_mutex_lock(&mtx);

                                             // ... produce ...
pthread_cond_wait(&cv, &mtx);                pthread_cond_signal(&cv);
// ... use the data ...
pthread_mutex_unlock(&mtx);
```

This is the right shape, but it is broken in two subtle ways.

Warning: *Spurious and lost wakeups*

A condition variable is allowed to wake a thread up *even when no other thread has signaled it*. This is called a ***spurious wakeup***. It is permitted by the standard for performance reasons, and you must not rely on its absence.
A worse failure happens if the producer signals the condition variable *before* the consumer reaches `pthread_cond_wait`. The signal targets only currently-waiting threads, so it is silently dropped: the consumer enters `cond_wait` just after, finds nobody to wake it up, and the program deadlocks. This is a ***lost wakeup***.

2.11.2 The predicate idiom

Both problems are fixed by the same pattern: protect a ***predicate*** with the same mutex used by the condition variable, and test it in a `while` loop around `cond_wait`:

```
// Consumer                                  // Producer
pthread_mutex_lock(&mtx);                    pthread_mutex_lock(&mtx);
while (!predicate)                           predicate = true;
    pthread_cond_wait(&cv, &mtx);            pthread_mutex_unlock(&mtx);
// ... use the data ...                      pthread_cond_signal(&cv);
pthread_mutex_unlock(&mtx);
```

This is correct against both pitfalls:

- If the producer sets `predicate = true` *before* the consumer enters the `while`, the loop test fails and the consumer never even calls `cond_wait`: no wakeup can be lost.
- If a spurious wakeup occurs and the predicate is still false, the loop simply re-enters `cond_wait`.

Tip: *Always the same mutex*

The predicate *must* be protected by the same mutex that is paired with the condition variable. Mixing two different mutexes (one for the predicate, one for `cond_wait`) reintroduces a window in which the producer's update and the consumer's test can interleave incorrectly, and lost/spurious wakeups come back.

2.11.3 Condition variable API

The complete set of calls is small:

```
// Pause
int pthread_cond_wait     (pthread_cond_t *cond, pthread_mutex_t *mtx);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mtx,
```

```
                                const struct timespec *abstime);
// Notify
int pthread_cond_signal   (pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Static initialization uses `PTHREAD_COND_INITIALIZER`; for dynamic allocation or custom attributes, use `pthread_cond_init` and `pthread_cond_destroy`.

2.11.4 Signal or broadcast?

The two notification primitives differ in how many waiting threads they wake up.

- `signal` wakes up *at least one* of the threads waiting on the condition variable. It is faster, but if the awoken thread's predicate is not satisfied (e.g., the data was meant for somebody else) we have a deadlock: that thread goes back to sleep and the others were never notified. `signal` is the right choice when all the waiting threads share an *equivalent* predicate, so that any of them can make progress.
- `broadcast` wakes up *all* the threads waiting on the condition variable. It is slower – every thread re-acquires the mutex one at a time – but it guarantees that every predicate is rechecked, which is what you want when the waiters might be looking for different things in the same shared state.

A useful default rule: when in doubt, use `broadcast`. Switching to `signal` is an optimization that should be made consciously, after checking that all waiters are interchangeable from the predicate's point of view.

3 OpenMP

3.1 Tasks

A **task** in OpenMP is an independent unit of work, comprising a well-defined set of instructions that can be executed asynchronously, enabling the decomposition of complex computations into smaller pieces that can be scheduled and executed in parallel by different threads. This **task abstraction** allows for flexible parallelism, supporting dynamic workloads with unpredictable execution flows.

Data Abstraction refers to the encapsulation of logically uniform pieces of information (arrays, structures, ...) that may be accessed concurrently by multiple threads. Proper management of data abstraction is essential to avoid race conditions and ensure consistency, especially when multiple tasks access shared data out-of-order. Mechanisms such as data dependencies, synchronization constructs, and memory models are employed to manage safe and efficient concurrent access.

Tasks often interact through shared data, giving rise to dependencies. They form a **task dependency graph**, where nodes represent tasks and directed edges represent data dependencies (i.e., a task must wait for another due to shared data). It is crucial that this graph is ***acyclic***, ensuring there are no circular wait conditions or deadlocks. It is sometimes possible to parallelize a workflow that is irregular or depends on runtime conditions using ***sections***. However, these solutions often become difficult to manage, and the intrinsic rigidity of the sections construct makes it difficult to express dynamic patterns of parallelism.

Since version 3.0, OpenMP introduced the **task** construct, which provides a flexible way to handle irregular execution flow and dynamic task creation. When defining a task, OpenMP internally creates a ”unit of work”, bundling together all necessary code, data, and local variables. This work is then scheduled for execution at some future point. Behind the scenes, OpenMP employs a queuing system to orchestrate the assignment of these tasks to the available threads, balancing the workload and enabling dynamic parallelism. A main thread generates tasks and a pool of threads executes them. As soon as they are free, the threads pick up one queued task each and execute it. When the main thread finishes creating tasks, it picks up a queued task and executes it as well.

Observation: ***Tasks vs Threads***

Creating tasks does not mean creating threads. The tasks are “units of work” that are assigned to the running threads. The pool of threads is unaltered (unless nested parallelism is involved) and mapped onto the underlying physical cores.

Task management in OpenMP revolves around three key concepts to take into account for writing correct and efficient parallel code with OpenMP tasks:

- **Data environment**: how data is assigned to each task, distinguishing shared and private variables.
- **Creation and execution**: when and how many tasks are generated, who executes them, and whether prioritization applies.
- **Synchronization and dependence**: how tasks are coordinated, scheduled, and whether they depend on other tasks’ completion.

3.1.1 Creating tasks

Let's start with a simple example, where we have a single region with a single thread that creates two tasks and then waits for all the other threads to finish. Once the tasks are created, they become available to be executed by the other available threads.

```
#pragma omp parallel
{
    #pragma omp single // a single thread creates the tasks
    {
        printf( "Yuk yuk, here is thread %d from "
            "within the single region\n", omp_get_thread_num() );
        #pragma omp task
        {
            printf( "\tHi, here is thread %d "
                "running task A\n", omp_get_thread_num() );
        }
        #pragma omp task
        {
            printf( "\tHi, here is thread %d "
                "running task B\n", omp_get_thread_num() );
        }
    } // all the other threads waits here
    printf(" :Here is thread %d after the single region, I was"
        "waiting for all the others\n", omp_get_thread_num());
}   // end of the parallel regions  \
```

Running this multiple times shows that a random thread enters the single region while others wait at the barrier. Some waiting threads will pick up the generated tasks. Once all tasks are completed, every thread proceeds.

```
   Yuk yuk, here is thread 5 from within the single region
      Hi, here is thread 2 running task A
      Hi, here is thread 1 running task B
:Hi, here is thread 5 after the single region, I was waiting for all the others
:Hi, here is thread 1 after the single region, I was waiting for all the others
:Hi, here is thread 0 after the single region, I was waiting for all the others
:Hi, here is thread 6 after the single region, I was waiting for all the others
:Hi, here is thread 4 after the single region, I was waiting for all the others
:Hi, here is thread 3 after the single region, I was waiting for all the others
:Hi, here is thread 7 after the single region, I was waiting for all the others
:Hi, here is thread 2 after the single region, I was waiting for all the others
```

Nowait

If we add a `nowait` clause to the single directive, we will get that the greeting message from all the other threads will almost certainly arrive before the greetings from the tasks execution.

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
    ...
```

In fact, the `nowait` lets the threads skip the single region entirely, and execute what is next, until the first scheduling point; which now is the very end of the parallel region.

Taskwait

Adding a `taskwait` directive at the end of the single region introduces an explicit synchronization point: the thread executing the single region will suspend at that point until all tasks it directly created have completed. Only then proceeds to subsequent statements within the single block.

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
        #pragma omp task
        { /* Tasks A, B, ... */ }

        #pragma omp taskwait  // wait for A and B to complete
        printf("All tasks have been executed\n");
    }
    printf(" :Here is thread %d after the single region\n",
        omp_get_thread_num());
}
```

The print following `taskwait` executes only after all tasks have finished. Meanwhile, the `nowait` clause allows other threads to proceed and print their messages independently of task completion.

```
 :Here is thread 3 after the single region
 :Here is thread 4 after the single region
 :Here is thread 2 after the single region
 :Here is thread 6 after the single region
 :Here is thread 7 after the single region
 :Here is thread 0 after the single region
 :Here is thread 1 after the single region
    Hi, here is thread 7 running task A
    Hi, here is thread 4 running task B
All tasks have been executed
 :Here is thread 5 after the single region
```

Warning: *Shallow synchronization*

The `taskwait` directive waits only for tasks directly created within the enclosing task region. If Task A itself creates Task A1, the `taskwait` will return as soon as A completes, even if A1 is still running. For deep synchronization that includes all descendants, use `taskgroup`.

Task scheduling points

A **task scheduling point** is when the runtime may suspend the current task and switch to another; the choice among available tasks is implementation-defined. Scheduling points occur during and after explicit task generation, after task completion, within `taskyield` regions, and inside synchronization constructs:

- **barrier** (implicit or explicit): All tasks created by any thread in the current `parallel` region are guaranteed to be complete when the barrier exits. Implicit barriers occur at the end of `parallel`, `for`, `sections`, and `single` (without `nowait`).
- **taskwait**: The encountering task suspends until all its *direct child* tasks have completed; it does *not* wait for descendants.
- **taskgroup**: All tasks generated within the region, *including their descendants*, are guaranteed to complete before the region exits.

3.1.2 Scope of variables

A key consideration when working with tasks is that a task may be executed at an arbitrary time after its creation, potentially by a different thread. The data environment of the task (which variables it can access and how) must therefore be determined at the moment of task creation, not execution.

Data-sharing clauses

- `shared()`: All accesses inside the task refer to the *same* storage as the original variable in the enclosing context. No copy is made; changes are visible to other tasks/threads.
- `private()`: Each task gets its own *uninitialized* copy of the variable. The original is unaffected.
- `firstprivate()`: Each task gets its own private copy, *initialized* with the value the variable had at task creation time. This is the safest choice for local variables whose value the task needs.

Warning: *Common pitfall*

If a task uses a `shared` reference to a local variable, and it goes out of scope before the task executes, the behavior is undefined. Use `firstprivate` to capture the value safely.

Implicit data-sharing rules for tasks

The rule-of-thumb is: *data, unless otherwise stated, are copied into local copies to preserve the data context at the moment of task creation*. The effect is the same as `firstprivate`.

- If a variable is **shared** in all enclosing constructs up to and including the innermost enclosing `parallel` region, then it is **shared** in the task.
- Otherwise, the variable is implicitly **firstprivate**.
- Variables with automatic storage declared inside the task region are **private**.
- Global and static variables are **shared** (predetermined).
- Dynamically allocated memory (heap) is **shared**.

Tasks need to capture the values of local variables at creation time (since may go out of scope or change before the task runs), while truly shared data should remain accessible to all tasks.

```
1  double pi = 3.1415;
2  double a = 0.0;
3
4  #pragma omp parallel
5  {
6      // pi, a : shared
7      #pragma omp single private (a)
8      {
9          int i = 1;
10         // a : private but not initialized
11         // i : private because it's a local variable
12         #pragma omp task
13         {
14             int j = 0;
15             // i, a : firstprivate by default (value is undefined)
16             // j : is a task's private variable
17             // pi : shared
18         }
19     }
20 }
```

Tied and untied tasks

By default, tasks are **tied**: once a thread begins executing a tied task, only that same thread can resume it if the task is suspended. This simplifies reasoning about thread-local state.

With the `untied` clause, a suspended task may be resumed by *any* available thread in the team:

```
1 #pragma omp task untied
2 { // This task can migrate between threads }
```

Untied tasks offer more flexibility and can improve load balancing, but with some caveats:

- `threadprivate` variables must not be used (the executing thread may change).
- The value of `omp_get_thread_num()` may change after a scheduling point within the task.
- Thread-local resources (e.g., thread-specific allocators) need to be handled carefully.

3.1.3 Tasks synchronization

A fundamental consideration in asynchronous task execution is determining *when* tasks actually run and how to properly coordinate them. Upon creation, a task may either be **deferred** (scheduled for later execution while the generating task region is suspended) or executed immediately. This flexibility introduces synchronization challenges: not all techniques that work well for thread-based parallelism apply directly to task-based execution.

In particular, **mutual exclusion** mechanisms (critical sections, atomic operations, locks) remain valid for tasks, since they protect shared data regardless of which thread or task accesses it. However, **event synchronization** constructs (barriers, busy-wait loops, boolean flags) are problematic: because tasks may be deferred or migrated, waiting for an event that depends on another task's progress can easily lead to deadlock if the waiting thread is the one that would otherwise execute the awaited task.

Synchronization constructs

We have already seen that constructs such as `taskwait`, `taskgroup` and `barrier` can be used to synchronize tasks.

- `barrier`: A barrier synchronizes all threads in the current team. All threads must reach the barrier before any can proceed. Note that barriers apply to threads, not tasks: using a barrier inside a task can lead to deadlock if not all threads reach it.
- `nowait`: This clause removes the implicit barrier at the end of worksharing constructs (such as `single`, `for`, `sections`). When applied to a `single` construct that generates tasks, other threads skip the single region and continue execution. Note that `nowait` does *not* wait for child tasks to complete; it only removes the implicit barrier for the worksharing construct itself.
- `taskwait`: This directive causes the encountering task to wait until all its *direct* child tasks have completed. It binds to the current task region, and the set of binding threads is the current team. When a thread encounters a `taskwait` construct, the current task region is suspended until all child tasks generated before the `taskwait` complete execution. Note that `taskwait` does *not* wait for descendants of child tasks (i.e., grandchildren).
- `taskgroup`: This construct provides more sophisticated synchronization by waiting for all *descendant* tasks (children, grandchildren, etc.) created within the taskgroup region to complete. This is useful for complex workflows with nested task generation.

Task reductions

Reductions on a `parallel` or `for` construct combine private partial results when the worksharing region ends. Tasks complicate this picture: tasks may be deferred, picked up by any thread in the team, and they do not participate in the enclosing `reduction( ... )` clause. A naive "parallel reduction" on a variable updated from inside tasks therefore loses the contribution of every task.

A first attempt is to fall back to an `atomic update` on a shared accumulator:

```
double result = 0;
#pragma omp parallel reduction(+:result)
{
    #pragma omp single nowait
    while (more_work()) {
        #pragma omp task firstprivate(first, last) shared(result) untied
        {
            double myresult = 0;
            for (int ii = first; ii < last; ii++)
                myresult += heavy_work(array[ii]);
            #pragma omp atomic update
            result += myresult;
        }
    }
}
```

This is correct, but every task contends on the same memory location at the end of its work, and the enclosing `reduction(+:result)` is misleading because the actual combination happens inside the tasks, not at the end of the parallel region.

OpenMP 5.0 introduced a dedicated mechanism: a `taskgroup` can declare a `task_reduction(op:var)`, and individual tasks join the reduction with the `in_reduction(op:var)` clause. Each participating task receives a private copy of `var`; partial values are combined when the taskgroup ends.

```
double result = 0;
#pragma omp parallel
{
    #pragma omp single nowait
    {
        #pragma omp taskgroup task_reduction(+:result)
        {
            while (more_work()) {
                #pragma omp task in_reduction(+:result) \
                    firstprivate(first, last) untied
                {
                    double myresult = 0;
                    for (int ii = first; ii < last; ii++)
                        myresult += heavy_work(array[ii]);
                    result += myresult;   // local private copy
                }
            }
        } // results combined here
    }
}
```

Inside the task, `result` refers to a thread-private copy associated with the reduction; no atomic

update is needed. The OpenMP runtime can also combine partial results in a tree, so even with many participating tasks the final reduction does not serialize on a single cache line.

Tip: *`atomic update` vs `task_reduction`*

Use `task_reduction` / `in_reduction` whenever many tasks contribute to the same accumulator: it removes contention on the shared variable and expresses the intent at the construct level. Reserve `atomic update` for cases where only a few tasks update a shared counter, or where the accumulator is structurally not a reduction (e.g. a histogram bin chosen at runtime).

3.1.4 Memory consistency model

Sequential Consistency

Consider the following scenario: we have two shared variables, $A = 0$ and $B = 0$, and two threads:

Thread a

```
[a1] A = 1;
[a2] print(B);
```

Thread b

```
[b1] B = 1;
[b2] print(A);
```

One may ask: what values can be printed for A and B by the respective threads? To answer this, let's introduce **Sequential Consistency**, a memory model defined by Leslie Lamport in 1979.

Sequential consistency means that the results of execution are the same as if all operations from all threads were executed in some sequential order, and each thread's operations appear in this sequence in the order written in the code. Put simply, every individual thread issues memory operations in program order, and all such operations appear as though they are executed one at a time in a global sequence.

In our example, under sequential consistency, the possible outputs printed by each thread for A and B are restricted by the orderings of the two stores and loads. For instance, it won't be possible for both threads to print 0, since that would imply that both the write to A and the write to B happened after both reads, which contradicts a single global ordering. Instead, possible outputs include (A=0, B=1), (A=1, B=0), or (A=1, B=1), depending on how the stores and loads are interleaved. Sequential consistency guarantees that the memory operations will be observed in an order that makes intuitive sense, as if there were some global sequence in which all operations happened.

Relaxed Consistency

Let's consider now the following scenario:

Thread a

```
[a1] A = 1
```

Thread b

```
[b1] A = 2
```

Thread c

```
[c1] print A
```

Even in the presence of local memory, which is permitted by OpenMP's memory model, the cache coherency protocol guarantees that writes to the same memory location are observed by all threads in the same order, regardless of what that order is. Under sequential consistency, either [a1] or [b1] must appear first in the global memory order, and all threads will observe that same ordering.

The drawback of sequential consistency is performance: it effectively serializes memory operations, preventing many hardware optimizations. For this reason, modern systems implement **relaxed memory models** that are better suited to out-of-order CPUs with deep cache hierarchies.

Under a relaxed memory model, threads may buffer writes to local storage (such as cache memory) before propagating them to main memory. Consider again our first example with threads *a* and *b*: thread *a* writes to *A* and reads *B*, while thread *b* writes to *B* and reads *A*. The key insight is that there is no ordering constraint between thread *a*'s write to *A* and thread *b*'s read of *A*, nor between thread *b*'s write to *B* and thread *a*'s read of *B*.

This means both threads could buffer their writes locally and issue their reads before those writes become visible to other threads. Without explicit synchronization, each thread reads from main memory (or its private cache), where both *A* and *B* are still 0. The result: both threads print 0, an outcome that sequential consistency would forbid.

OpenMP memory model

To understand how OpenMP handles these complexities, its memory model is defined in terms of two key concepts:

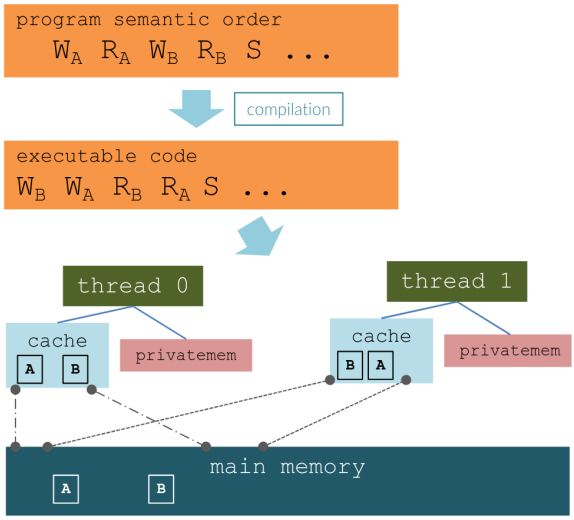
- **Coherence**: the prescribed behavior of the memory system when multiple accesses to the same memory location are performed by different threads.
- **Consistency**: the order in which memory accesses by different threads become visible to one another.

These concepts work together to specify when and how memory updates propagate between threads.

The OpenMP memory model is a relaxed-consistency, shared-memory model. All threads share access to a common memory space where variables can be stored and retrieved. However, each thread is also permitted to maintain its own **temporary view** of memory, which may reside in registers, cache, or other local storage. This temporary view allows threads to avoid accessing main memory on every variable reference, improving performance. Importantly, a thread's temporary view can become inconsistent with main memory or with other threads' views until an explicit synchronization occurs, typically via `flush`, barriers, or other synchronization constructs that force the views to reconcile. In addition to shared memory, each thread has access to **threadprivate memory**, a private storage area that cannot be accessed by other threads.

In practice, the relaxed memory model introduces several layers of reordering between what the programmer writes and what actually happens in memory:

1. **Compiler reordering**:
   The compiler may rearrange the source code's operations into any semantically equivalent order that it deems more efficient.
2. **Hardware reordering**:
   The processor may further reorder instructions at runtime, so the actual memory commit order can differ from the compiled code order.
3. **Temporary views**:
   Each thread's temporary view of memory may diverge from main memory (and from other threads' views) until synchronization occurs.



The consistency model specifies which orderings between reads (R), writes (W), and synchronizations (S) are guaranteed. OpenMP enforces that all S operations (barriers, flushes, locks, etc.) appear in the same order across threads, giving a sequentially consistent view for synchronization

events. Within each thread, the following orderings are preserved: $R \to R$, $W \to W$, $R \to W$, $S \to R$, $S \to W$, $R \to S$, $W \to S$, and $S \to S$. Notably, $W \to R$ is *not* guaranteed, meaning a read may be reordered before a preceding write, so only synchronization points prevent stale data from being observed.

Recall that under sequential consistency, all memory operations appear to execute in some global order that respects each thread's program order, and all threads observe the same ordering. In this case, the program order, code order, and commit order are effectively equivalent. Relaxed consistency lifts some of these constraints: threads may observe memory updates in different orders, and only synchronization points enforce a consistent view across threads to ensure safe, predictable parallel execution.

3.1.5 NUMA, affinity and the first-touch policy

The relaxed consistency model tells us *when* a memory update becomes visible; on a multi-socket machine we also have to worry about *where* a memory page physically lives. Modern HPC nodes are **NUMA** (Non-Uniform Memory Access) systems: each socket has its own memory controller, and a thread accessing a page on a remote socket pays a latency and bandwidth penalty compared to a local access. Thread placement and data placement therefore become first-class performance concerns alongside synchronization.

The first-touch rule

Linux (and most modern operating systems) allocates virtual address space lazily: `malloc` only reserves the range, while a physical page is actually mapped on the first write. The kernel then places the page on the NUMA node of the thread that performed that first write. This is the **first-touch policy**, and it has a sharp consequence for OpenMP: the thread that initialises the data effectively decides which socket owns it for the rest of the program.

Touch by one — the wrong default

The simplest pattern (initialise serially, then process in parallel) places *all* of the data on the master thread's NUMA node:

```
1 double *array = malloc(N * sizeof(double));
2
3 // Serial init: master thread first-touches every page
4 for (int ii = 0; ii < N; ii++)
5     array[ii] = (double)ii;
6
7 #pragma omp parallel for reduction(+:S)
8 for (int ii = 0; ii < N; ii++)
9     S += array[ii];
```

Every worker thread that does not live on the master's socket has to fetch the data over the inter-socket link. The reduction loop is now bandwidth-bound on a single memory controller, no matter how many cores participate.

Touch by all — first-touch via parallel init

The fix is to initialise the array in a `parallel for` with the same schedule that will be used to process it. Each thread first-touches the slice it will later read:

```c
double *array = malloc(N * sizeof(double));

#pragma omp parallel for schedule(static)
for (int ii = 0; ii < N; ii++)
    array[ii] = (double)ii;

#pragma omp parallel reduction(+:S)
{
    #pragma omp for schedule(static) reduction(+:S)
    for (int ii = 0; ii < N; ii++)
        S += array[ii];
}
```

The two loops use the same `schedule(static)` chunking, so iteration *ii* is touched and later read by the same thread, on the same socket. The data is now spread across all NUMA nodes, and aggregate memory bandwidth scales with the number of sockets.

Warning: *Schedules must match*

First-touch only pays off if the init schedule matches the use schedule. A `schedule(dynamic)` use loop after a `schedule(static)` init loop scrambles the mapping and reintroduces remote accesses.

threadprivate arrays — explicit local allocation

For maximum locality, each thread can allocate its own array via `threadprivate`. The array variable itself is replicated per thread, and each thread calls `calloc` (or `malloc` + write) inside the parallel region:

```c
double                 *array  = NULL;
unsigned long long int  mywork = 0, first_iteration = 0;
#pragma omp threadprivate(array, mywork, first_iteration)

#pragma omp parallel
{
    int nthreads = omp_get_num_threads();
    mywork = N / nthreads + 1;
    array  = calloc(mywork, sizeof(double));   // first-touch is local

    #pragma omp for schedule(static)
    for (unsigned long long ii = 0; ii < N; ii++)
        array[ii - first_iteration] = (double)ii;

    #pragma omp for schedule(static) reduction(+:S)
    for (unsigned long long ii = 0; ii < N; ii++)
        S += array[ii - first_iteration];
}
```

Each thread now owns a contiguous, NUMA-local buffer. The trade-off is that `array` is no longer a single contiguous region visible from outside the parallel region: code that needs to operate on the whole array as one object (gather, MPI send, file I/O, …) has to reassemble the partitions or use a different layout.

Tip: *Pinning threads with* `proc_bind` *and* `OMP_PLACES`

First-touch only works if threads stay on the core that did the first write. Use `OMP_PROC_BIND=true` (or a placement policy: `close`, `spread`, `master`) and `OMP_PLACES=cores` (or `sockets`, `threads`) at runtime, and the `proc_bind(close|spread|master)` clause on a `parallel` region for finer control. Without binding, the kernel scheduler is free to migrate a thread to another socket and the locality argument collapses.

3.1.6 FLUSH operations

OpenMP `flush` operations are used to enforce the consistency between a thread's temporary view of the main memory and the main memory itself, or between multiple threads' temporary views.

Strong FLUSH

A strong flush operation enforces consistency between a thread's temporary view and main memory. It is applied to a specified set of variables called the **flush-set**. When a strong flush is executed, the implementation must not reorder memory operations for variables in the flush-set with respect to the flush itself; this restriction applies both to the code generated for memory operations and to the code for the flush operation on those variables.

The primary purpose of a strong flush is to ***guarantee that a value written to a variable by one thread may be read by a second thread***. To achieve this, the programmer must ensure that the second thread has not written to the variable since its last strong flush, and that the following sequence of events occurs in this specific order:

1. Thread 1 writes the value to the variable.
2. Thread 1 performs a strong flush of the variable.
3. Thread 2 performs a strong flush of the variable.
4. Thread 2 reads the value from the variable.

This protocol ensures that the write by thread 1 is committed to main memory before thread 2 attempts to read it, thereby preventing stale or inconsistent values from being observed.

Release or acquire FLUSH

Release and acquire flushes enforce consistency between the memory views of two synchronizing threads, rather than between a thread and main memory directly.

A **release flush** guarantees that any prior operation that writes or reads a shared variable will appear to complete before any operation that writes or reads the same shared variable and follows an acquire flush with which the release flush synchronizes. Concretely, a release flush propagates the values of all shared variables in the thread's temporary view to memory prior to performing any subsequent atomic operation that may establish a synchronization.

An **acquire flush** discards any value of a shared variable in the thread's temporary view that has not been written since the last release flush. It then loads any value of a shared variable propagated by a release flush that synchronizes with it into the temporary view, making that value available for subsequent reads.

Release and acquire flushes may also be used to guarantee that a value written to a variable by one thread can be read by a second thread. To accomplish this, the programmer must ensure that

the second thread has not written to the variable since its last acquire flush, and that the following sequence of events occurs in this specific order:

1. The value is written to the variable by thread 1.
2. Thread 1 performs a release flush.
3. Thread 2 performs an acquire flush.
4. The value is read from the variable by thread 2.

This protocol establishes a synchronization edge between the release and acquire operations, ensuring that writes made visible by the release are observable after the acquire.

Example: *FLUSH example*

Consider the following snippet where multiple threads increment a shared counter:

```
1 #pragma omp parallel
2 {
3     ...
4     #pragma omp atomic update
5     counter++;
6     ...
7     double z = f(counter, ...);
8 }
```

how can you ensure that all the threads have updated `counter` before any thread calls `f` ?

Approach 1: The most straightforward solution is to insert a **barrier** before line 8. A barrier implies a strong flush, guaranteeing that all updates are visible to all threads. However, barriers are expensive because they force all threads to synchronize, which is inherently serializing and can severely limit parallel performance.

Approach 2: A more targeted solution uses an **explicit flush** directive to force the thread to read from main memory rather than its local cache:

```
1  #pragma omp parallel
2  {
3      ...
4      #pragma omp atomic update
5      counter++;
6      ...
7      #pragma omp flush(counter) acquire
8      double z = f(counter, ...);
9      ...
10 }
```

The `acquire` flush ensures that the thread discards any stale cached value of `counter` and reads the most recent value from memory. This approach is lighter than a full barrier.

Approach 3: For protecting more complex operations involving shared variables, **named critical regions** provide both mutual exclusion and implicit strong flushes:

```
#pragma omp parallel
{
    ...
    #pragma omp critical (update_counter)
    counter++;
    ...
    #pragma omp critical (read_counter)
    double z = f(counter, ...);
    ...
}
```

Named critical regions ensure that only one thread at a time executes each named section, and the implicit flush guarantees memory consistency. Different names allow unrelated critical sections to execute concurrently, reducing contention compared to unnamed critical regions.

Implicit flushes

OpenMP provides implicit flushes at specific synchronization points, eliminating the need for explicit `flush` directives in most cases:

```
#pragma omp barrier     // Implicit flush of all variables
#pragma omp critical    // Flush on entry and exit
#pragma omp ordered     // Flush on entry and exit
#pragma omp parallel    // Flush on entry (fork) and exit (join)
#pragma omp for         // Flush at implied barrier (if present)
#pragma omp single      // Flush at implied barrier (if present)
```

These implicit flushes apply to *all* shared variables (the so-called "strong flush"), not merely those modified within the region. Additionally, **lock operations** (`omp_set_lock`, `omp_unset_lock`) carry implicit flushes, as discussed in the following section.

⚠ Warning: *No implicit flush for* `master`

The `#pragma omp master` directive does **not** include an implicit flush. If memory consistency is required after master-only code, an explicit flush or barrier must be inserted.

Memory ordering semantics

OpenMP 5.x adopted the C++11/C11 memory model, exposing fine-grained control over memory ordering. The available orderings, from strongest to weakest, are:

```
seq_cst   // sequential consistency (default)
acq_rel   // acquire-release
release   // release only
acquire   // acquire only
relaxed   // no synchronization guarantees
```

1. `seq_cst`: is the default memory order. It is the most strict memory order and it is the most expensive. A single thread must execute all operations in the order they are written in the code.
2. `acq_rel`: is the acquire-release memory order. It combines acquire semantics on reads and release semantics on writes. It ensures that all memory operations before a release are visible to

any thread that performs an acquire on the same synchronization variable. It is less strict than `seq_cst` (no global total order) but sufficient for most producer-consumer patterns.

3. `release`: it is like doing a write back of whatever is in the local cache to the main memory.
4. `acquire`: it is like doing a read from the main memory to the local cache.
5. `relaxed`: is the least strict memory order and it is the cheapest. It is used when you do not need to ensure that all operations are executed in the order they are written in the code.

A `release` operation on a variable ensures that ***all preceding*** memory operations (reads and writes) by the executing thread become visible to any other thread that subsequently performs an `acquire` operation on the same synchronization variable.

```
1 x = 1;
2 y = 2;
3 #pragma omp atomic write release
4 z = 3; // Release on z synchronizes x, y as well
```

Locks

OpenMP provides explicit lock routines for mutual exclusion when `critical` sections are too coarse-grained or when finer control over synchronization is needed. Unlike named critical regions (which serialize all threads entering a region with the same name), locks can be associated with specific data structures, enabling concurrent access to different parts of a shared resource.

A lock is represented by the `omp_lock_t` type. For scenarios where a thread needs to acquire the same lock multiple times (e.g., recursively), OpenMP provides **nested locks** via `omp_nest_lock_t`. It can be acquired multiple times by the same thread that originally acquired it; each acquisition increments an internal counter, and the lock is only released when this reaches zero.

⚠ Warning: *Memory consistency*

Lock operations imply an implicit **flush** of all the thread's visible variables. Specifically, `omp_set_lock` has **acquire** semantics and `omp_unset_lock` has **release** semantics, establishing a happens-before relationship between the lock release and subsequent acquisitions.

Function	Description
`omp_init_lock(omp_lock_t *)`	Initialize a lock (must be called before first use).
`omp_destroy_lock(omp_lock_t *)`	Destroy a lock (must not be locked when destroyed).
`omp_set_lock(omp_lock_t *)`	Acquire a lock; **blocks** until the lock is available.
`omp_unset_lock(omp_lock_t *)`	Release a lock (must be called by the owning thread).
`omp_test_lock(omp_lock_t *)`	**Non-blocking** attempt to acquire the lock; returns nonzero on success (lock acquired), zero on failure.

Nested lock variants (`omp_init_nest_lock`, `omp_set_nest_lock`, etc.) follow the same pattern.

The function `omp_set_lock` blocks until the lock becomes available, which can lead to deadlock if threads hold locks while waiting for others. The non-blocking `omp_test_lock` returns immediately, allowing the caller to take alternative action if the lock is unavailable:

```c
if (omp_test_lock(&lock)) {
    // Lock acquired: perform critical operation
    omp_unset_lock(&lock);
} else {
    // Lock unavailable: retry later or take alternative action
    #pragma omp taskyield  // Yield to allow other tasks to progress
}
```

This pattern is essential for implementing deadlock-free algorithms: if a thread cannot acquire all required locks, it releases those already held and retries, preventing circular wait conditions.

When acquiring multiple locks, always follow these principles:

- **Consistent ordering**: acquire locks in a fixed global order (e.g., by memory address).
- **Try-lock with backoff**: use `omp_test_lock` ; if it fails, release all held locks and retry.
- **Never block while holding**: avoid `omp_set_lock` when already holding another lock

Advanced Concept: *Lock hints*

OpenMP 5.0 introduced lock hints to guide runtime optimizations:

```c
omp_init_lock_with_hint(omp_lock_t *lock, omp_sync_hint_t hint);
omp_init_nest_lock_with_hint(omp_nest_lock_t *lock, omp_sync_hint_t hint);

// Available hints (can be combined with bitwise OR):
omp_sync_hint_none           // No hint (default)
omp_sync_hint_uncontended    // Lock is usually uncontended
omp_sync_hint_contended      // Lock is often contended
omp_sync_hint_nonspeculative // Prefer non-speculative implementation
omp_sync_hint_speculative    // Allow speculative (e.g., HTM-based) implementation
```

These hints allow the runtime to select appropriate implementations: an uncontended lock might use a simple spinlock, while a contended lock might use a queue-based algorithm to reduce cache-line bouncing. Note that as of 2025, compiler support for these hints remains limited; the runtime may ignore unsupported hints.

Tip: *Memory ordering decisions*

Do you need atomicity of an operation?
- YES: Use atomic
  - Single variable access?
    - YES: atomic read/write with acquire/release
    - NO: atomic capture or use lock
- NO: Do you need memory ordering?
  - YES: Can you use implicit synchronization (barrier, lock)?
    - YES: Use barrier/lock (preferred)
    - NO: Consider atomic release/acquire pattern
  - NO: Use reduction or thread-private variables

Case study: from critical to locks

To see why the lock zoo exists, consider a recurrent pattern from N-body tree codes: each thread iterates over a list of *active* particles, and before computing a force on a particle it must first *drift* that particle to the current time. Many threads may need the same particle, so the drift must be

protected from data races, but only *distinct* particles really conflict — two threads updating two different particles should run in parallel.

Stage 1 — coarse-grained `critical`. The minimal protection is a critical section around the update:

```
if (P[idx].time < update_time) {
    #pragma omp critical
    {
        if (P[idx].time < update_time)        // double-check
            update_particle(&P[idx], update_time);
    }
}
```

Entry to the critical region performs an implicit acquire flush; exit performs a release flush, so the new `time` is visible to subsequent readers. The cost is that *every* update is serialised, even between threads working on different particles: the critical region behaves as a global lock.

Stage 2 — per-particle `omp_lock_t`. Move the lock from the algorithm to the data: give every particle its own lock, and use `omp_test_lock` so a thread that finds the particle already busy can do something else instead of blocking.

```
if (omp_test_lock(&P[idx].lock)) {
    update_particle(&P[idx], update_time);
    omp_unset_lock(&P[idx].lock);
} else {
    // Another thread is updating this particle.
    // Don't acquire the lock; just wait for the work to be done.
    while (P[idx].time < update_time)
        nanosleep(...);
}
```

This unlocks parallelism between disjoint particles, but `omp_lock_t` is an opaque, runtime-managed structure: it is large (taking memory away from useful state) and every `set` / `unset` is a function call into the OpenMP runtime.

Stage 3 — custom atomic spinlock. If the contention is low and the runtime overhead dominates, the lock can be replaced by an `int` field updated through atomic operations, with explicit acquire/release semantics:

```
int my_lock;
#pragma omp atomic capture
{ my_lock = P[idx].lock; P[idx].lock++; }   // fetch-and-add

if (my_lock == 0) {                         // we got the lock
    update_particle(&P[idx], update_time);

    #pragma omp atomic write release        // publish the update
    P[idx].lock = 0;
} else {
    #pragma omp atomic read acquire         // see the latest time
```

```
12      double t = P[idx].time;
13      while (t < update_time) { /* spin */ }
14  }
```

The `release` on the unlock guarantees that everything written inside `update_particle` is visible to any thread that subsequently performs the `acquire` read.

Stage	Complexity	Memory	Concurrency	Latency
`critical`	low	none	very low (global)	high
`omp_lock_t`	medium	high (opaque)	high (per-particle)	medium (runtime)
atomic spinlock	high	low (`int`)	high (per-particle)	low (CPU-level)

Observation: *False sharing on tightly packed structures*

If `P[idx]` is small (say 16 bytes), several particles share a 64-byte cache line. Locking `P[0]` invalidates that line on every other socket, and a thread reading `P[1]` or `P[2]` suffers a cache miss even though the data it wants is logically untouched. Padding each entry to a full cache line removes the false sharing at the cost of memory footprint — a trade-off worth measuring on the target machine.

Case study: optimistic concurrency on a sorted linked list

Lock granularity also shapes the design of concurrent data structures. Consider a sorted, doubly linked list into which many threads insert in parallel. A single global lock around `find_and_insert` would serialise all insertions; per-node locks open the door to real concurrency, but the algorithm becomes subtle.

Optimistic search, then validate. The standard pattern is: search without holding any lock, then lock the two neighbours of the candidate insertion point and *validate* that the structure has not changed under us:

```
1  llnode_t *prev = NULL, *next = NULL;
2  find(start, value, &prev, &next);              // optimistic, lock-free
3
4  // Acquire prev then next, in a fixed order, with non-blocking try-lock
5  while (!try_lock_pair(prev, next))
6      #pragma omp taskyield;
7
8  // Validate: did somebody insert between prev and next while we waited?
9  if (prev->next != next || next->prev != prev) {
10     unlock_pair(prev, next);
11     continue;                                   // restart from head
12 }
13 insert_between(prev, value, next);
14 unlock_pair(prev, next);
```

The validation step is what makes the algorithm correct: between the lock-free `find` and the moment we own the locks, another thread may have inserted a new node exactly where we wanted to insert ours.

Pitfall 1 — blocking lock during repair walk. A natural optimisation is, on a failed validation, to walk locally from `prev` to find the new `next` instead of restarting from the head. The temptation is to grab each successive lock with the blocking `omp_set_lock`, since we already hold one lock and "only" need to acquire one more. This reintroduces the classic deadlock:

Warning: *Never block while holding a lock*

Two threads walking in opposite directions can each block on the lock the other one holds, forming a circular wait. Inside a repair walk, use `omp_test_lock` with a bounded number of attempts; if the limit is reached, release everything and restart. Blocking acquisitions are only safe when no other lock is currently held.

Pitfall 2 — accidentally shared cursor. A subtler bug shows up in the surrounding task code. The repair walk needs a per-task cursor, e.g. `start`, that each task updates as it advances:

```
1 llnode_t *start = head;          // declared OUTSIDE the task
2
3 #pragma omp task                 // BUG: start is implicitly shared
4 {
5     for (int b = 0; b < batch; b++)
6         find_and_insert(start, new_value, &start);
7 }
```

By the implicit data-sharing rules introduced earlier in this chapter, a variable that is shared in the enclosing parallel region remains shared in the task. Every task therefore reads and writes the same `start`, racing on the cursor. The fix is to declare `firstprivate(start)` (or `firstprivate(head)` and let each task maintain its own cursor) so that each task gets an independent copy.

Putting it together. The correct version combines (i) optimistic search, (ii) deadlock-free local repair walk via bounded `omp_test_lock` attempts with fall-back to a full restart, (iii) a second validation after the repair walk in case the structure changed again while we were re-acquiring locks, and (iv) `firstprivate` on the per-task cursor. Once the locks provide acquire/release fences at every `set`/`unset`, the underlying atomic loads on `prev`/`next` pointers can use `memory_order_relaxed` — the lock pair is doing the heavy synchronization.

3.1.7 Controlling the task creation - Clauses

The task creation construct adtmits a number of clauses that allow to control the creation of tasks:

```
1 #pragma omp task clause, clause, ...
2 {
3     /* code */
4 }
```

Clause	Description
`if (expr)`	When `expr` evaluates `false`, the encountering thread does not create a new task in the task queue; instead, the execution of the current task is suspended and the newly created task is *undeferred* and started. The task suspended is resumed afterwards.
`final (expr)`	The `final(expr)` clause can be used to suppress the task creation and then to control the tasking overhead, especially in recursive task creation. If `expr` evaluates `true`, no more tasks are generated and the code is executed immediately. That is propagated to all the children tasks. That is called an *included* task and is always undeferred. Use `omp_in_final()` to check if the task is a final one.
`mergeable`	This clause avoids a separated data environment to be created if a task is undeferred or included.
`tied, untied`	This clause lets a suspended task be resumed by a different thread than the one that started it. The default option is `tied`.
`depend, priority`	*We will cover this afterwards*

Task creation in loops

Many times happens that you need to create tasks in a loop (for instance, a task for every entry, or sections, of an array). The taskloop construct has been conceived to ease this cases, combining the for loops and the tasks natively.

```
1 #pragma omp taskloop [clause [[,] clause] ...]
2 {
3     for-loops // (perfectly nested)
4 }
```

Clauses are very similar to both the usual for and task constructs: `private`, `firstprivate`, `lastprivate`, `shared`, `default`, `if`, `final`, `priority`, `untied`, `mergeable`

There are 3 specific clauses for `taskloop`:

- `grainsize(arg)`: `arg` is a positive integer. This clause regulates the granularity of the work assignment, ensuring that the amount of work per task is not too small. The number of loop iterations assigned to each task is at least the grainsize and at most $2\times$ grainsize, but not more than the number of iterations left.
- `num_tasks(arg)`: `arg` is a positive integer, indicating the maximum number of tasks that can be generated at runtime. This is useful to limit the task creation overhead.
- `nogroup`: Specifies that the tasking construct is not embedded in an otherwise implied `taskgroup` construct.

3.1.8 Task priority and dependencies

Task priority

Even when tasks can execute concurrently, it is sometimes beneficial for certain tasks to run earlier than others. For instance, tasks that receive data should typically execute before tasks that post-process the results. The `priority(p)` clause allows the programmer to suggest a scheduling preference to the OpenMP runtime: higher values of `p` indicate higher priority.

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task priority(100)
    read_data(...);
    #pragma omp task priority(50)
    process_and_save_data(...);
    #pragma omp task priority(10)
    postprocess_and_send_data(...);
}
```

Note that priority is a *hint* to the scheduler, not a strict ordering guarantee. The runtime may ignore priorities under certain conditions (e.g., when work-stealing from another thread's queue).

Data races and dependencies

A ***data race*** leads to undefined behavior; it occurs when three conditions are simultaneously met:

i two or more threads access the same memory location,
ii at least one access is a write
iii the accesses are not ordered by synchronization.

Tip: ***Debugging data races***

The compiler flag `-fsanitize=thread` enables runtime detection of data races:

```
gcc -fopenmp -fsanitize=thread -g -o program program.c
```

In task-based parallelism, data dependencies frequently arise: one task may need the results produced by another, or must wait for another task's operations to complete before proceeding. OpenMP provides the `depend` clause to express these relationships explicitly, allowing the runtime to schedule tasks in a valid order while maximizing parallelism.

OpenMP recognizes three primary **dependence types**:

- `in`: the task depends on any *previously generated* task that has an `out`, `inout`, or `mutexinoutset` dependence on the same memory region.
- `out` (and `inout`): the task depends on any previously generated task with an `in`, `out`, `inout`, or `mutexinoutset` dependence on the same memory region.
- `mutexinoutset`: similar to `out`, but tasks with `mutexinoutset` on the same variable are *mutually exclusive*: they may execute in any order, but never concurrently.

Observation: ***`inout` is equivalent to `out`***

In OpenMP 5.x, `inout` is treated as an alias for `out`. The distinction is purely semantic (indicating that the task both reads and writes a variable); the scheduling behavior is identical.

The dependence types above correspond to well-known data hazards from instruction parallelism:

- **Read-after-Write (RaW)**
  also called a *true dependency* or *flow dependency*, occurs when a task must read data produced by a previous task. This is the fundamental producer-consumer relationship:

```
#pragma omp task depend(out:result)
compute_result(&result);

#pragma omp task depend(in:result)
use_result(&result);
```

- **Write-after-Read (WaR)**
  also known as an *anti-dependency*, occurs when a task overwrites data that a previous task is still reading. The write must wait until all prior reads complete:

```
#pragma omp task depend(in:buffer)
read_buffer(&buffer);

#pragma omp task depend(out:buffer)
overwrite_buffer(&buffer);
```

- **Write-after-Write (WaW)**
  also known as *output dependency*, occurs when two tasks write to the same location. The final value must come from the logically later task. This is sometimes called a *false dependency* or *name dependency*, since renaming the variable could eliminate it:

```
#pragma omp task depend(out:x)
initialize_x(&x);

#pragma omp task depend(out:x)
reinitialize_x(&x);
```

- **Read-after-Read (RaR)**
  involves two tasks reading the same memory region. Since neither modifies the data, there is no actual dependency; both tasks can execute concurrently:

```
#pragma omp task depend(in:data)
analyze_data(&data);

#pragma omp task depend(in:data)
visualize_data(&data);
```

Tip: ***OpenMP 5.x synchronization best practices***

When choosing synchronization mechanisms, prefer higher-level constructs that are easier to reason about:

1. **First choice**: High-level constructs (`barrier` , `critical` , `atomic`)
2. **Second choice**: Atomics with explicit memory order (`acquire` / `release`)
3. **Last resort**: Explicit `flush` (only when synchronizing multiple variables at once)
4. **Avoid**: `volatile` provides no synchronization guarantees in OpenMP

4 MPI

Advanced usage of some MPI features on topology awareness, shared memory and one-sided communications.

- Basics of the one-sided communication
- Building a hierarchy of Communicators that reflects the topology
- Exchanging data in shared memory windows among MPI processes

and we will see briefly other advanced features of MPI:

- Derived Data-Types
- More details on Groups and Communicators
- Virtual topologies (cartesian and graph communicators) and neighborhood collectives
- Non-blocking collectives

4.1 Two-Sided Communications

By its very nature, the message-passing paradigm is designed around the cooperative exchange of information among two or more processes whose address spaces are isolated and not directly accessible by other processes. In the two-sided model (`MPI_Send` / `MPI_Recv` , `MPI_Isend` / `MPI_Irecv` , collectives) every transfer requires the active participation of *both* endpoints: the sender posts a send, the receiver posts a matching receive, and the MPI library is responsible for matching the two. This is a clean abstraction, but it tightly couples *data movement* with *synchronization*: the receiver's progress is tied to its willingness to enter the library and look for a matching message.

4.2 One-Sided Communication

In one-sided communication a process can read, write or update memory in a remote process *without* that process making a matching MPI call. The sender (here called the **origin**) specifies both buffers — its own and the remote one — and issues a `MPI_Put` , `MPI_Get` or `MPI_Accumulate` into a region of memory that the remote process (the **target**) has previously *exposed* as a window. The target need not stop its computation to handle the request.

This decoupling of data movement from process synchronization is the conceptual gain: the application controls when synchronization happens (via fences, PSCW or locks) independently of when data is transferred. On hardware that supports it (RDMA-capable NICs, on-node shared memory, GPUs with peer access), one-sided operations can map directly onto remote DMA, side-stepping the CPU on the target side entirely.

The price is that correctness becomes much subtler than in two-sided code: visibility of remote updates is governed by an explicit memory model (Section 4.2.2), and the absence of a matching call on the target means there is no automatic ordering. We will return to these caveats throughout the chapter.

4.2.1 Remote Memory Access - RMA

Memory Windows

To allow remote processes to access part of its memory, a process must explicitly *expose* that region as a `MPI_Win` (a window). The window is the unit of RMA: every `MPI_Put` / `MPI_Get` / `MPI_Accumulate` addresses a (rank, displacement) pair within a specific window. The MPI standard offers four routines to create one, differing in who allocates the memory and how it is laid out across the communicator.

```
// (a) wrap memory the user has already allocated
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
                   MPI_Info info, MPI_Comm comm, MPI_Win *win);

// (b) let MPI allocate (often pinned) memory and expose it
int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
                     MPI_Comm comm, void *baseptr, MPI_Win *win);

// (c) like (b) but allocate in a shared-memory segment so processes
//     on the same node can also load/store the window directly
int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,
                            MPI_Comm comm, void *baseptr, MPI_Win *win);

// (d) create an empty window; attach/detach memory dynamically
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win);
```

All four are collective on `comm`. They differ in the following way:

- `MPI_Win_create` is the most flexible: you bring your own buffer (typically from `malloc`). The library may need to register that buffer with the NIC at first use, and on some implementations cannot perform zero-copy RDMA into a region the OS may page out. It is the right choice when the memory must be allocated by the application (e.g. a third-party library returns it).
- `MPI_Win_allocate` lets MPI allocate the memory and the window in one shot. Implementations are free to use pinned/registered memory, symmetric heaps or other tricks, so on RDMA hardware this is usually the highest-performance option.
- `MPI_Win_allocate_shared` requires `comm` to be a shared-memory communicator (see Section 4.2.2 and the Shared Memory section); the resulting window is directly addressable via plain C loads and stores from any rank in `comm`, in addition to the usual RMA operations.
- `MPI_Win_create_dynamic` creates a window with no associated memory; processes then `MPI_Win_attach` / `MPI_Win_detach` buffers as the application allocates/frees them. Useful for irregular data structures that grow and shrink at runtime, at the cost of additional bookkeeping (origins must learn the address of attached regions out-of-band).

The `size` argument is in bytes; `disp_unit` is the unit (in bytes) used to interpret displacements supplied in subsequent RMA calls. Setting `disp_unit = sizeof(T)` lets you address window slots as if they were a typed array.

Tip: ***Don't churn windows***

Window creation can be expensive: it may register memory with the NIC and synchronize all ranks in `comm`. As a rule of thumb, *create windows once at the start of the program (or of a phase) and keep them alive*; do not create and free a window per message.

Warning: `disp_unit`

The `disp_unit` parameter is often misunderstood. Setting it to anything other than 1 (byte) can cause subtle bugs when mixing datatypes. The displacement calculation becomes:

```
target_addr = base + (target_disp * disp_unit)
```

Data Movement

Once a window exists, the origin can issue three families of RMA operations against it. They are all *non-blocking*: the call returns when the request has been queued, not when the data has reached its destination — completion is the job of a synchronization call (Section on Ordering and Synchronization below).

```
// remote write: origin -> target
int MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype
    origin_datatype,
            int target_rank, MPI_Aint target_disp,
            int target_count, MPI_Datatype target_datatype, MPI_Win win)
                ;

// remote read: target -> origin
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
    origin_datatype,
            int target_rank, MPI_Aint target_disp,
            int target_count, MPI_Datatype target_datatype, MPI_Win win)
                ;

// remote update: target = target  op  origin
int MPI_Accumulate(const void *origin_addr, int origin_count,
    MPI_Datatype origin_datatype,
                   int target_rank, MPI_Aint target_disp,
                   int target_count, MPI_Datatype target_datatype,
                   MPI_Op op, MPI_Win win);
```

The signature is symmetric: the origin describes its own buffer (`origin_addr`, `origin_count`, `origin_datatype`) and the target slice in the window (`target_rank`, `target_disp`, `target_count`, `target_datatype`). The displacement is in units of `disp_unit` as specified at window creation.

`MPI_Accumulate` composes a remote read-modify-write into a single message: it applies an `MPI_Op` (`MPI_SUM`, `MPI_MAX`, `MPI_REPLACE`, …) to the target memory. Two atomic RMW primitives complete the picture:

```
// fetch-and-op: returns the previous value while applying op
int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
                     MPI_Datatype datatype, int target_rank,
                     MPI_Aint target_disp, MPI_Op op, MPI_Win win);

// compare-and-swap: atomic CAS on a single element
int MPI_Compare_and_swap(const void *origin_addr, const void *
    compare_addr,
```

```
8                           void *result_addr, MPI_Datatype datatype,
9                           int target_rank, MPI_Aint target_disp, MPI_Win
                               win);
```

`MPI_Fetch_and_op` is the canonical building block for distributed counters and queue indices — a single-element `MPI_Accumulate` that also returns the prior value. The Mandelbrot example in `MPI/codes/Mandelbrot/backend_lock.c` uses it exactly that way: workers atomically increment the queue tail with `MPI_Fetch_and_op(&one, &tail, MPI_INT64_T, PRODUCER_RANK, OFFSET_TAIL, MPI_SUM, win)` to claim a slot, then `MPI_Accumulate` the new entry into it.

Warning: *Atomicity is per element*

Plain `MPI_Put` and `MPI_Get` are *not* atomic with respect to overlapping operations from other origins. Atomicity is provided only by `MPI_Accumulate`, `MPI_Fetch_and_op` and `MPI_Compare_and_swap`, and even then only on individual elements (and on predefined operations). Two concurrent `MPI_Put`s on overlapping byte ranges yield undefined data.

Ordering and Synchronization

A non-blocking RMA call only *schedules* a transfer; the application is responsible for declaring when transfers are allowed and when they must have completed. MPI defines two coarse styles:

- **Active target**: the target is aware of the epoch and participates in opening or closing it. Two flavors exist: collective *fence* epochs and finer-grained *Post–Start–Complete–Wait* (PSCW) epochs over arbitrary process subgroups.
- **Passive target**: the target makes no MPI call at all. The origin opens the epoch with `MPI_Win_lock` (or `MPI_Win_lock_all`) and closes it with `MPI_Win_unlock`; the local progress engine on the target services the requests.

In all cases the rule is the same: every RMA call must lie inside an *access epoch* on the origin, and every byte being read or written must lie inside an *exposure epoch* on the target.

Fences are the simplest mechanism. A single collective call opens and closes epochs at all ranks of the window's communicator:

```
1  int MPI_Win_fence(int assert, MPI_Win win);
```

Between two fences, every rank may simultaneously be origin and target for any other rank: it is a Bulk-Synchronous-Parallel pattern. The first fence opens the epoch on all processes, RMA operations are issued by whichever ranks need to communicate, and the second fence guarantees both *remote completion* of every queued RMA and that no new RMA can start until the next fence. Conceptually, if $E_i^{(n)}$ is the n-th epoch on process i, a fence enforces a happens-before edge $\forall i, j : E_i^{(n)} \to E_j^{(n+1)}$.

This is the right tool for stencil and ghost-cell exchanges, where every process talks to its neighbors at the same algorithmic step. Its weakness is also its strength: synchronization is collective even when the actual communication pattern is sparse, so fences cost $O(\log p)$ in latency regardless of the message graph and stragglers slow everyone down.

Post-Start-Complete-Wait (PSCW) avoids that global cost by letting only the involved processes synchronize. It introduces two disjoint groups: an *exposure group* (the targets) and an *access group* (the origins).

```
// On a target: open exposure epoch for group of origins
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win);
// On an origin: open access epoch toward group of targets
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win);
// On an origin: close the access epoch, completing all RMA
int MPI_Win_complete(MPI_Win win);
// On a target: wait until every origin has called Win_complete
int MPI_Win_wait(MPI_Win win);
```

The protocol is: targets call `Win_post`, origins call `Win_start` (these can be matched without a barrier — typically via the `MPI_MODE_NOCHECK` assert when the application already guarantees the order), the origins issue their RMA operations, and finally `Win_complete` / `Win_wait` terminate the access and exposure epochs respectively. PSCW is the right tool for irregular but *statically known* communication patterns: the application builds the groups once and reuses them. Building and freeing `MPI_Group` objects in the inner loop is a known performance trap.

Locks (passive target). Locks are the fully one-sided mechanism: the target makes no MPI call to open or close the epoch.

```
MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win);

MPI_Win_unlock(int rank, MPI_Win win);

MPI_Win_flush/flush_local(int rank, MPI_Win win);
```

where `lock_type` is one of:

- `MPI_LOCK_EXCLUSIVE`: only one origin can hold the lock on `rank` at a time — serialized read–modify–write access.
- `MPI_LOCK_SHARED`: multiple origins can hold the lock concurrently — safe for parallel reads, or for non-conflicting accumulates.

`MPI_Win_lock_all(assert, win)` opens a passive epoch toward *every* rank in the communicator at once, which is convenient for global hash tables or work-stealing queues. Be aware, however, that some implementations track $O(p)$ state per origin under `lock_all`, which can hurt scalability on very large jobs.

While the lock is held, the origin can issue RMA operations and wait for them to complete *without* releasing the lock by calling:

- `MPI_Win_flush(rank, win)`: completes all outstanding RMA from this origin to `rank`, both at the origin (buffers reusable) and at the target (data has landed).
- `MPI_Win_flush_local(rank, win)`: completes them only at the origin — the target may not yet have seen the data.

Warning: *Misconception about locks*

Despite the unfortunate naming, MPI locks are not mutex-like. They do not provide mutual exclusion against the target's own local loads and stores: the target may freely modify its window memory while an origin holds the lock, which can race with the RMA operations. "Lock" here just delimits an RMA epoch with a particular concurrency mode (shared or exclusive) for other *origins*.

Flush vs. sync. Two different completion calls exist; they look superficially similar and are routinely confused.

Observation: `MPI_Win_flush` *vs.* `MPI_Win_sync`

- `MPI_Win_flush` : forces completion of RMA operations from the origin to a (or every) target. It is about **network completion** — "did the data arrive at the target's public copy?"
- `MPI_Win_sync` : synchronizes the calling process's public and private window copies (Section 4.2.2). It is about **memory consistency** — "is the data visible to local CPU loads?". It does not complete any pending RMA.

The two calls operate on different sides of the same window and serve different purposes. As a rule of thumb: standard origin-side RMA only needs `MPI_Win_flush`; you need `MPI_Win_sync` when you mix RMA traffic with direct loads/stores on the same window (typically with shared-memory windows).

Synchronization Asserts

Each synchronization routine takes an `int assert` parameter. Setting it to `0` is always correct; non-zero values are *contractual promises* the application makes to the implementation, which the latter is then free to use to skip otherwise-mandatory work (barriers, lock checks, memory fences). Violating an assert is undefined behavior — there is no error, just silent corruption — so *add asserts only when you have proven they hold*.

The five flags defined by the standard, and the calls that accept them:

- `MPI_MODE_NOPRECEDE` — accepted by `MPI_Win_fence` : "no RMA from this process precedes this fence in the current epoch." Eliminates the initial barrier (typical use: the first fence after window creation).
- `MPI_MODE_NOSUCCEED` — accepted by `MPI_Win_fence` : "no RMA from this process will follow this fence in the current epoch." Eliminates the final barrier (typical use: the last fence before `MPI_Win_free`).
- `MPI_MODE_NOSTORE` — accepted by `MPI_Win_fence` , `MPI_Win_post` , `MPI_Win_complete` : "the local window memory was not modified by local stores since the last sync." Lets the implementation skip cache flushes / private-to-public sync.
- `MPI_MODE_NOPUT` — accepted by `MPI_Win_fence` , `MPI_Win_post` , `MPI_Win_complete` : "the local window will not be the target of a Put or Accumulate during this epoch." Lets the implementation skip incoming-buffer processing.
- `MPI_MODE_NOCHECK` — accepted by `MPI_Win_start` , `MPI_Win_post` , `MPI_Win_lock` , `MPI_Win_lock_all` : "the matching synchronization on the other side is already in place / no conflicting locks exist." Eliminates the matching/conflict-detection handshake.

The first four are valid only on `Win_fence` / `Win_post` / `Win_complete` ; `NOCHECK` is valid only on `Win_start` / `Win_post` / `Win_lock` / `Win_lock_all` . They can be combined with bitwise OR.

Example: *Common assert combinations*

```
// First fence after window creation: no RMA preceded, nothing
   stored yet.
MPI_Win_fence(MPI_MODE_NOPRECEDE | MPI_MODE_NOSTORE, win);

// Read-only epoch: nobody puts, nobody stores.
MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT, win);

// Last fence before Win_free.
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);

// Application-level coordination guarantees no conflicting locks.
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, target, MPI_MODE_NOCHECK, win);
```

Tip: *Develop with* `assert = 0`

Always make the program correct with `assert = 0` first, then add asserts incrementally and benchmark each step. Asserts are not cross-implementation portable in their performance impact, and getting them wrong is invisible in the output until the day you change MPI vendor.

4.2.2 RMA Memory Model

So far we have spoken loosely of "the window memory." MPI is more careful: each window is associated with a memory model that prescribes how RMA traffic and local CPU loads/stores see each other. There are two such models, exposed via the window attribute `MPI_WIN_MODEL` :

```
int model, flag;
MPI_Win_get_attr(win, MPI_WIN_MODEL, &model, &flag);
// model is MPI_WIN_UNIFIED  or  MPI_WIN_SEPARATE
```

The model is chosen by the implementation, not by the user, and depends on the underlying hardware. Conceptually, MPI describes every window location as having two copies:

- a **private** copy — what the local process sees through ordinary loads, stores, sends and receives;
- a **public** copy — what remote RMA operations target.

Unified model (`MPI_WIN_UNIFIED`). Public and private copies are physically the same memory. This is what you get on cache-coherent CPUs (essentially every modern HPC node) and on shared-memory windows. A local store eventually becomes visible to remote `MPI_Get` / `MPI_Accumulate` , and conversely, without extra MPI calls — but only *eventually*, and only at the synchronization points (epoch boundaries, flushes). Within an unbounded epoch and without a flush, you can still observe "torn" or stale values.

Separate model (`MPI_WIN_SEPARATE`). Public and private copies are logically distinct — the model designed for non-coherent NICs, accelerators on the PCIe path and other exotic memory.

A local store updates only the private copy; an incoming `MPI_Put` updates only the public copy. They are reconciled by synchronization calls. In particular, `MPI_Win_sync` acts as the bidirectional memory fence: it pushes recent local stores into the public copy and pulls recent remote updates into the private copy on the calling process.

Warning: `MPI_Barrier` *is not a memory fence*

The standard explicitly states that `MPI_Barrier` provides *process* synchronization, not memory synchronization. Any pattern of the form "I did Puts; everyone calls `MPI_Barrier`; now everyone reads the new values via local loads" is non-portable: it works on x86 by accident of strong cache coherence and may break on weaker architectures, on GPU memory or under the separate model. Use proper RMA synchronization (`MPI_Win_flush`, `MPI_Win_sync`, fence/PSCW/unlock).

For most clusters made of cache-coherent CPUs the unified model is in force and this discussion may seem pedantic; the reason it matters is portability. As soon as a window backs onto GPU memory across PCIe, or you run on an architecture with weaker ordering, the separate-model rules become real. The portable idiom for "producer writes via local stores, consumers read via RMA" is therefore: producer issues local stores then `MPI_Win_sync` (private → public), consumers see the data via RMA after synchronization.

4.3 Shared Memory

An HPC machine is made of several nodes that are connected by a top-level network We call these nodes "hosts".

Currently every host contains a NUMA region which consists in a number of sockets. Each of which is physically connected to RAM banks. In turn, the sockets are interconnected so that a process running on a given core can access a memory bank not physically attached to its socket.

4.3.1 Getting Shared Memory Pools

General and specific routines

MPI provides a method to know about that and to distinguish the tasks depending some of their charateristics:

- a `general routine` to split a communicator into sub-communicators:

```
1 MPI_Comm_split (
2     MPI_Comm comm,      // the communicator to be splitted
3     int color,          // discriminate task
4     int key,            // hint for new ranks
5     MPI_Comm *newcomm   // the new communicator
6 );
```

- a `specific routine` to split a communicator into sub-communicators following the hardware hierarchy (i.e. NUMA region, socket, core, etc.):

```c
MPI_Comm_split_type (
    MPI_Comm comm,      // the communicator to be splitted
    int split_type,     // the splitting property
    int key,            // hint for new ranks
    MPI_Info info,      // helper for the splitting property
    MPI_Comm *newcomm   // the new communicator
);
```

The `split_type` parameter is used to specify the splitting property. It can be one of the following values:

- `MPI_COMM_TYPE_SHARED` : split the communicator based on the shared memory pools.
- `MPI_COMM_TYPE_HW_GUIDED` : split the communicator based on the hardware guided property.
- `MPI_COMM_TYPE_HW_UNGUIDED` : split the communicator based on the hardware unguided property.

The `key` parameter is used to hint for the new ranks. It is used to sort the tasks depending on their charateristics.

The `info` parameter is used to provide helper for the splitting property. It is used to provide additional information about the splitting property.

Placement

```c
MPI_Win_allocate_shared (
    MPI_Aint size,        // the size of the window
    int disp_unit,        // the displacement unit
    MPI_Info info,        // the information to be used
    MPI_Comm comm,        // the communicator
    void *baseptr,        // the base pointer
    MPI_Win *win          // the window
);
```

As with other window-creation routines, the memory allocation for `MPI_Win_allocate_shared` does not require the allocated size to be uniform: each task may allocate a different amount. There is no strict specification as to where the memory should be physically placed: an MPI implementation will typically attempt to maximize data locality and affinity.

By default, the allocation is **contiguous**. However, this contiguity does not only refer to the virtual address space: ideally, it also implies allocation in the same physical memory bank. On many systems, though, it is not possible to remap memory at arbitrary sizes to guarantee this property. Specifying a non-contiguous allocation (`MPI_Info_set(info, "alloc_shared_noncontig", "true")`) can allow the MPI implementation to more flexibly optimize the placement, possibly enhancing the affinity for each individual task.

A few caveats are worth remembering:

- **First-touch policy**: on Linux, a virtual page is only bound to a physical NUMA node when it is first written. With contiguous shared allocation, the rank whose process first touches a given page wins it — if rank 0 zero-initializes the entire shared array, every page ends up on rank 0's NUMA node, regardless of which rank will actually work on it later. The portable workaround is to have each rank initialize its own slice.
- **Contiguity does not imply locality**: even with the default contiguous allocation, the virtual

address range can span pages on different NUMA nodes. The address arithmetic works (it is one big array) but the latency does not (some loads cross sockets). For NUMA-sensitive kernels, request `alloc_shared_noncontig` so the implementation can place each rank's slice on the rank's local node.

- **Non-uniform sizes**: each rank may pass a different `size` (including `0`). This is useful, for instance, when only the host master allocates a control block, but it complicates pointer arithmetic — which is why `MPI_Win_shared_query` (below) exists.
- **Hugepages and stride**: on systems where the implementation backs the shared region with hugepages, the per-rank slice may be padded up to a hugepage boundary. Pointer-arithmetic that assumes `baseptr + rank*size` is therefore not portable; always use `MPI_Win_shared_query`.

A useful routine related to window allocation and memory sharing is `MPI_Win_shared_query`, which allows processes to retrieve a pointer to shared memory in a window, for either direct access or load/store operations. Its prototype is:

```
MPI_Win_shared_query(
    MPI_Win   win,          // the window object
    int       rank,         // rank whose segment to query, or
       MPI_PROC_NULL
    MPI_Aint *size,         // returns segment size
    int      *disp_unit,    // returns displacement unit
    void    **baseptr       // returns base pointer in shared memory
);
```

This function enables a process to obtain the base address and size of the memory segment exposed by any rank in a shared-memory communicator window. Commonly, it is called with `rank = MPI_PROC_NULL`, which gives a pointer to the start of the entire shared memory region (regardless of which process allocated it), and returns:

- the address of the first byte in the shared memory window (for contiguous allocation), or
- the base address of the first process that specified a non-zero `size` (for non-contiguous allocation).

The figure above illustrates the difference between default contiguous allocation and non-contiguous allocation across multiple processes. With contiguous allocation, addresses increase sequentially across processes, so the pointer returned can be used by all processes to access the underlying shared array as a unified space. In non-contiguous mode, each process's memory may be mapped separately, and thus, only the base address of the first valid allocation is returned.

This routine is fundamental for writing portable and generic code utilizing shared memory windows, especially when contiguity cannot be assumed.

4.3.2 Shared Memory Access

A shared-memory window is unusual in that it can be accessed in two ways: through standard RMA calls (`MPI_Put` / `Get` / `Accumulate`) *and* through plain C loads and stores via the pointer returned by `MPI_Win_shared_query`. The MPI standard explicitly defines load/store semantics for shared windows only under the unified memory model (Section 4.2.2), but in practice all major implementations on cache-coherent x86 / aarch64 nodes do provide this.

The freedom to do plain loads and stores is also the source of the only real complication: ordering

and visibility between processes are no longer the MPI library's job, they are the application's. There are two clean access patterns.

Pattern A: epoch-based. Wrap the load/store phases in passive epochs and use `MPI_Win_sync` as the bidirectional memory fence:

```
// once at start: open a long-lived passive epoch toward every rank
MPI_Win_lock_all(MPI_MODE_NOCHECK, win);

// writer phase
shared_ptr[my_offset] = new_value;          // plain store
MPI_Win_sync(win);                          // private -> public
MPI_Barrier(shared_comm);                   // make all writers reach
    this point

// reader phase
MPI_Win_sync(win);                          // public -> private
int v = shared_ptr[some_offset];            // plain load

// at shutdown
MPI_Win_unlock_all(win);
```

The `MPI_Win_sync` is what flushes the local CPU's view of the shared region; `MPI_Barrier` provides the temporal synchronization between writer and reader phases (recall it is *not* a memory fence, only a process barrier — but combined with `Win_sync` on each side it gives the full ordering edge).

Pattern B: atomics. When writers and readers operate concurrently rather than in disjoint phases, fall back to atomics on the shared memory — either MPI's atomic RMA primitives (`MPI_Fetch_and_op` , `MPI_Compare_and_swap` , `MPI_Accumulate` with `MPI_REPLACE`) or, if you are committed to a unified-model platform, C11 atomics on the shared addresses. The Mandelbrot `shmem` backend (`MPI/codes/Mandelbrot/backend_shmem.c`) is the canonical example: it allocates a queue with `MPI_Win_allocate_shared` , queries the base pointer with `MPI_Win_shared_query` , and uses C11 `atomic_fetch_add` on the queue head/tail so workers can claim work items without any MPI call on the hot path.

Warning: `MPI_Barrier` *is still not a memory fence*

The same warning as in Section 4.2.2 applies, with extra force here: when reading data written by another rank via plain loads, you need an MPI window-synchronization call (`MPI_Win_sync` , end-of-fence, `Win_unlock` , …) on each side. A bare `MPI_Barrier` between writer and reader is not portable. On x86 with hardware coherence it appears to work because cache snooping silently does the right thing.

4.4 Cartesian Communication

Many scientific and engineering problems operate on regular, structured grids: finite difference methods for PDEs, stencil computations, image processing, and particle-mesh methods in astrophysics. When parallelizing these applications via domain decomposition, we partition the computational domain into sub-domains, each handled by a distinct MPI process.

The fundamental challenge is neighbor communication: processes must exchange boundary data (ghost zones, halo cells) with their neighbors. While we could manually compute neighbor ranks using arithmetic on process coordinates, this approach is error-prone, non-portable, and fails to leverage potential hardware optimizations.

Why not Manual Rank Arithmetic

Consider a 2D grid decomposition with dimensions ($P_x \times P_y$). Given a process with coordinates (i, j), its neighbors are:

```
// Manual neighbor calculation (fragile approach)
int rank_left  = (i > 0)     ? i-1 + j*Px   : MPI_PROC_NULL;
int rank_right = (i < Px-1)  ? i+1 + j*Px   : MPI_PROC_NULL;
int rank_down  = (j > 0)     ? i + (j-1)*Px : MPI_PROC_NULL;
int rank_up    = (j < Py-1)  ? i + (j+1)*Px : MPI_PROC_NULL;
```

The possible issues, or inconveniencies, with this approach are:

- **Boundary handling**: Explicit conditionals for edges; periodic boundaries add complexity
- **Dimension ordering**: Row-major vs column-major rank assignment affects all calculations
- **Scalability**: 3D or higher dimensions require nested conditionals
- **Optimization**: MPI implementations cannot optimize message routing without knowing the topology

Cartesian Communication Model

MPI provides virtual topologies to express logical process arrangements. A **Cartesian topology** maps processes onto a regular n-dimensional grid, and MPI provides a suite of routines for creating and querying such topologies:

- **Automatic coordinate ↔ rank translation**: Convert between process ranks and their multidimensional coordinates in the grid.
- **Built-in periodic boundary support**: Specify whether each grid dimension is periodic (wrap-around), e.g., for toroidal grids.
- **Neighbor lookup via relative displacement**: Functions for obtaining neighbor ranks given directional offsets ("shifts")—no manual arithmetic required.
- **Hints for process placement optimization**: Inform the MPI implementation of communication patterns, potentially improving performance by optimizing process placement (implementation-dependent).

Example: *Non-Periodic 3x2 Grid*

```
        j=0    j=1
      +------+------+
i=0   |  R0  |  R1  |   Rank = i + j*3
      +------+------+
i=1   |  R2  |  R3  |   R0:(0,0) R1:(1,0)
      +------+------+   R2:(0,1) R3:(1,1)
i=2   |  R4  |  R5  |   R4:(0,2) R5:(1,2)
      +------+------+
```

4.4.1 Core API Functions

MPI_Dims_create

Computes a "balanced" distribution of processes across dimensions:

```
int MPI_Dims_create (int nnodes,    // Total number of processes
                     int ndims,     // Number of dimensions
                     int dims[]);   // IN/OUT: dimension sizes
```

Key behavior: pre-set dimensions (`dims[i] > 0`) are preserved; zero entries are computed by MPI. The algorithm tries to minimize surface-to-volume ratio for communication efficiency.

Example: *2D grid, 12 processes*

```
// unconstrained: let MPI choose
int dims[2] = {0, 0};
MPI_Dims_create(12, 2, dims); // Result: dims = {4, 3} or {3, 4}

// constrained: force 4 columns
int dims2[2] = {0, 4};
MPI_Dims_create(12, 2, dims2); // Result: dims2 = {3, 4}
```

Warning: *`MPI_Dims_create` failure*

`MPI_Dims_create` fails if `nnodes` is not divisible by the product of pre-set dimensions. Always verify `dims[i] > 0` for all `i` after the call.

MPI_Cart_create

Creates a new communicator with Cartesian topology:

```
int MPI_Cart_create(
    MPI_Comm comm_old,      // Input communicator
    int ndims,              // Number of dimensions
    const int dims[],       // Size of each dimension
    const int periods[],    // Periodicity flags (0 or 1)
    int reorder,            // Allow rank reordering?
    MPI_Comm *comm_cart     // Output: new Cartesian comm
);
```

where:

- `periods[i]` : If non-zero, dimension `i` wraps around (periodic boundary)
- `reorder` : If non-zero, MPI may reassign ranks to optimize for hardware topology
- If size(`comm_old`) > product(`dims`), excess processes receive `MPI_COMM_NULL`

Tip: *Best practices*

Set reorder=1 in production code. This allows MPI implementations (especially on clusters with complex interconnects like fat-trees or torus networks) to map logically adjacent processes to physically adjacent nodes, reducing communication latency.

MPI_Cart_coords and MPI_Cart_rank

Once a Cartesian communicator exists, two routines translate between linear ranks and grid coordinates:

```
// rank --> coords
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[]);

// coords --> rank
int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank);
```

`MPI_Cart_coords` is normally the very first call after `MPI_Cart_create`: each process asks for its own coordinates and uses them to figure out which slice of the global domain it owns. `MPI_Cart_rank` goes the other way and is the right tool when you need to address a specific position in the grid (e.g. "the process at coordinates $(0,0)$ collects the global result"). Both functions handle periodic dimensions transparently: a coordinate that falls outside $[0, \texttt{dims[i]} - 1]$ in a periodic dimension wraps to a valid one; in a non-periodic dimension it is invalid.

MPI_Cart_shift

For the central use case — finding nearest neighbors along an axis — the dedicated routine is:

```
int MPI_Cart_shift(MPI_Comm comm,
                   int direction,    // dimension index (0, 1, ...)
                   int disp,         // displacement (typically +1 or
                      -1)
                   int *rank_source, // rank that would shift INTO me
                   int *rank_dest);  // rank I would shift TO
```

The naming is from the point of view of a directional shift: "if everybody shifts by `disp` along `direction`, who would I receive from and who would I send to?" For a 2D grid, calling it once with `direction = 0, disp = +1` gives the left/right neighbors of the current process, and once with `direction = 1, disp = +1` gives the down/up neighbors. The example in `MPI/codes/example_of_cartesian_communicator/basic.c` does exactly this:

```
int up, down, left, right;
MPI_Cart_shift(cart_comm, 0, 1, &up,   &down);
MPI_Cart_shift(cart_comm, 1, 1, &left, &right);
```

When the shift would cross a non-periodic boundary, `MPI_Cart_shift` returns `MPI_PROC_NULL` for that endpoint. `MPI_PROC_NULL` is safe to pass to `MPI_Send`, `MPI_Recv`, `MPI_Isend`, `MPI_Irecv`, etc. — those calls become no-ops — so a single code path works for interior *and* boundary processes without explicit conditionals.

Example: *Halo exchange on a 2D Cartesian grid*

The canonical pattern: post all receives non-blocking, then all sends, then wait. `MPI_PROC_NULL` returned by `MPI_Cart_shift` on the boundary makes the corresponding send/recv a no-op, so no special-case code is needed.

```
int dims[2] = {0, 0}, periods[2] = {0, 0}, coords[2];
int up, down, left, right;
MPI_Comm cart_comm;

MPI_Dims_create(world_size, 2, dims);
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, /*reorder=*/1, &
    cart_comm);
MPI_Cart_coords(cart_comm, my_cart_rank, 2, coords);

MPI_Cart_shift(cart_comm, 0, 1, &up,   &down);
MPI_Cart_shift(cart_comm, 1, 1, &left, &right);

MPI_Request reqs[8];
int n = 0;

// post the four receives first
MPI_Irecv(halo_up,    H, MPI_DOUBLE, up,    0, cart_comm, &reqs[n
    ++]);
MPI_Irecv(halo_down,  H, MPI_DOUBLE, down,  1, cart_comm, &reqs[n
    ++]);
MPI_Irecv(halo_left,  H, MPI_DOUBLE, left,  2, cart_comm, &reqs[n
    ++]);
MPI_Irecv(halo_right, H, MPI_DOUBLE, right, 3, cart_comm, &reqs[n
    ++]);

// then the four sends
MPI_Isend(edge_up,    H, MPI_DOUBLE, up,    1, cart_comm, &reqs[n
    ++]);
MPI_Isend(edge_down,  H, MPI_DOUBLE, down,  0, cart_comm, &reqs[n
    ++]);
MPI_Isend(edge_left,  H, MPI_DOUBLE, left,  3, cart_comm, &reqs[n
    ++]);
MPI_Isend(edge_right, H, MPI_DOUBLE, right, 2, cart_comm, &reqs[n
    ++]);

MPI_Waitall(n, reqs, MPI_STATUSES_IGNORE);
```

The matching tag on each pair (`0` / `1` for the vertical axis, `2` / `3` for the horizontal one) is what makes the receives match the corresponding sends from the neighbor on the other side. See `MPI/codes/example_of_cartesian_communicator/basic.c` for a complete runnable version, and `heat2d_cartesian.c` in the same directory for a full 2D heat-diffusion solver built on this pattern (using `MPI_Sendrecv` plus a derived `MPI_Type_vector` for column halos).

5 Supercomputers

Modern scientific discovery increasingly relies on high-performance computing infrastructure capable of handling massive computational workloads. Supercomputers represent the pinnacle of this infrastructure, combining thousands of processors, accelerators, and specialized networking hardware to tackle problems that would be intractable on conventional systems. These range from climate modeling and molecular dynamics simulations to machine learning training and quantum mechanics calculations. We will focus on Leonardo, one of Europe's flagship supercomputing systems. We'll examine its modular design, computing capabilities, and the practical aspects of accessing and utilizing it.

5.1 Leonardo Supercomputer

Leonardo is a pre-exascale supercomputer hosted at CINECA in Bologna, Italy, and represents one of the cornerstones of the European High Performance Computing Joint Undertaking (EuroHPC JU). Leonardo was designed to support both traditional HPC workloads and emerging AI applications.

The system exemplifies the trend toward heterogeneous computing in modern HPC, integrating both CPU-centric compute nodes and GPU-accelerated nodes optimized for highly parallel workloads.

5.1.1 System Architecture and Modules

Leonardo adopts a modular architecture consisting of several distinct partitions, each tailored to specific classes of workloads. The two primary modules are the **Booster module** and the **Data-Centric (DHPC) module**, supplemented by additional service and hybrid partitions.

Booster Module

The Booster module constitutes the heart of Leonardo's computational power and is optimized for massively parallel, compute-intensive workloads. This partition is composed of GPU-accelerated nodes, each equipped with:

- **CPUs**: 1 Intel Xeon Platinum 8358 processor (32 cores, 2.6 GHz base frequency)
- **GPUs**: 4 NVIDIA A100 GPUs (64 GB HBM2 memory each), providing substantial computational throughput for floating-point operations and tensor computations
- **System Memory**: 512 GB DDR4 RAM per node
- **Node-to-GPU Interconnect**: GPUs connected via NVIDIA NVLink for high-bandwidth, low-latency communication within each node

The Booster module comprises approximately 3,500 such nodes, yielding roughly 14,000 NVIDIA A100 GPUs in total. This configuration is particularly well-suited for applications in machine learning, computational fluid dynamics, materials science, and other domains that benefit from the massive parallelism offered by modern GPUs. Each A100 GPU provides mixed-precision capabilities (FP64, FP32, FP16, and Tensor Core operations), enabling efficient execution of both traditional HPC simulations and AI training workloads.

Data-Centric (DHPC) Module

The Data-Centric module provides CPU-centric compute resources designed for workloads that require large memory capacity, complex branching logic, or I/O-intensive operations. Nodes in this partition feature:

- **CPUs**: 2 Intel Xeon Platinum 8358 processors per node (64 cores total)
- **System Memory**: Typically 512 GB DDR4 RAM per node, with some high-memory configurations available
- **Local Storage**: NVMe SSDs for fast local data staging

This module comprises approximately 1,500 nodes and is intended for traditional HPC applications such as computational chemistry codes, finite element analysis, and large-scale data analytics tasks that do not map efficiently onto GPU architectures.

Inter-Node Network Topology

Leonardo employs a **Dragonfly+** network topology based on NVIDIA Mellanox InfiniBand HDR (High Data Rate) technology and NVIDIA Quantum QM8700 switches. This topology is designed to provide high bandwidth and low latency while maintaining scalability and resilience.

In the Dragonfly+ topology, compute nodes are organized into **groups** (or cells), with dense all-to-all connectivity within each group. Inter-group connections are carefully engineered to minimize the maximum number of hops between any two nodes while balancing cost and performance. Each node is connected via InfiniBand HDR links operating at 200 Gb/s, ensuring that communication-intensive parallel applications can scale efficiently across thousands of nodes.

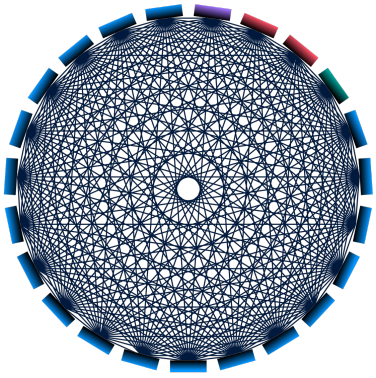


Figure 5.1: Dragonfly+ Topology. Booster Module nodes (blue), I/O cell (purple), Data-centric cells (red), and Hybrid cell (green) [2].

5.1.2 Storage System

Leonardo's storage infrastructure is organized into a two-tier hierarchy:

- **Fast Tier (NVMe-based)**: 5.4 PB of capacity with an aggregate bandwidth of 1.4 TB/s. This tier uses high-speed NVMe SSDs and is intended for I/O-intensive workloads requiring rapid data access, such as checkpointing in large-scale simulations or intermediate data staging for machine learning training.
- **Capacity Tier (HDD-based)**: 106 PB of capacity with an aggregate bandwidth of 2.9 TB/s. This tier provides long-term storage for datasets, simulation outputs, and archival purposes. While slower than the fast tier, it offers significantly greater capacity at a lower cost per terabyte.

Both tiers are accessible via a parallel filesystem (typically GPFS/Spectrum Scale or Lustre) that provides a unified namespace and supports concurrent access from thousands of compute nodes.

Observation: *Top500*

The **Top500** list is the authoritative ranking of the world's fastest supercomputers, updated twice a year (June and November). Leonardo currently holds the rank of 9th fastest supercomputer globally. At its debut in November 2022, Leonardo achieved the impressive position of **4th fastest**, trailing only Frontier (USA), Fugaku (Japan), and LUMI (Finland).

5.1.3 Accessing Leonardo

Access to Leonardo is managed through a project-based allocation system. For users affiliated with Italian universities and research institutions, CINECA offers several account classes tailored to different project sizes and durations:

Tip: *Apply for resources*

If you belong to an Italian university, you can apply for resources in the following classes:

- **Class B**: Large-scale projects, typically requesting millions of core-hours. Duration: 12 months. Calls for proposals are issued twice per year.
- **Class C**: Small to medium-sized projects. Duration: 9 months. Continuous submission process, with up to 10 calls per year.
- **Class D**: Storage allocations related to HPC simulations, useful for projects requiring substantial long-term data retention. Duration: 36 months.
- **Test Account**: Small allocations for testing, debugging, and scaling studies. Duration: 3 months. Ideal for validating code performance before requesting larger allocations.

Upon logging in to Leonardo, users are greeted by the **Message of the Day** (MOTD), an informational banner designed to keep users up-to-date with the system's status and important announcements. The MOTD typically includes:

- A brief description of the cluster architecture or key features
- Real-time system status updates (e.g., queues, ongoing issues, resource availability)
- "In evidence" messages highlighting noteworthy news or actionable items
- Important notifications, such as planned downtime, upcoming maintenance, or urgent advisories

5.2 Working on Leonardo

5.2.1 Filesystem Organization

Upon logging into Leonardo, users find themselves in a hierarchical filesystem with several designated directories, each serving a specific purpose:

- **HOME** (`$HOME`): A personal home directory; it is backed up and is intended for configuration files, scripts, and small datasets. Quota-limited, typically to tens of gigabytes.
- **WORK** (`$WORK`): A larger, project-specific workspace for active development, source code, and intermediate build artifacts. Not typically backed up, but persistent across sessions.
- **SCRATCH** (`$SCRATCH`): A high-performance scratch space for temporary files generated during job execution. This area offers the best I/O performance and is intended for files needed only during the runtime of a job. Files in scratch are subject to automatic purging policies (e.g., files not accessed for 40 days may be deleted), so do not use this for long-term storage.
- **DATA** (`$DATA`): Intended for long-term storage of large datasets and simulation outputs. This directory is often mapped to the capacity storage tier and may have different quota and performance characteristics than `$SCRATCH`.
- **PUBLIC**: A shared directory where project members can exchange data and scripts. Useful for collaboration within a project team.

Active simulation I/O should target `$SCRATCH` for speed, while final results should be moved to `$DATA` or `$WORK` for safekeeping.

5.2.2 Software and Module Environment

Leonardo's software stack is managed via the **Environment Modules** system, with packages built and deployed using the **Spack** package manager. This approach allows for coexistence of multiple versions and configurations of libraries and compilers, enabling reproducibility and flexibility.

Upon login, a default environment is loaded, but users typically need to load additional modules to access specific compilers, MPI implementations, or scientific libraries.

The module system provides commands to query, load, and manage the software environment:

- `module avail` or `module av`: List all available modules.
- `module list`: Show currently loaded modules and profiles.
- `module load <name>`: Load a specific module into the environment.
- `module unload <name>`: Unload a module.
- `module purge`: Remove all loaded modules (useful for starting with a clean environment).
- `module show <name>`: Display information about what a module does (env. variables, paths).
- `modmap -m <name>`: Search or map module names (CINECA-specific tool).

Different *profiles* or module collections may be available depending on whether you are working on CPU-only or GPU-enabled nodes, or whether you are using a specific programming model (MPI, CUDA, OpenMP, OpenACC, etc.).

Programming Environments

Leonardo supports multiple compiler toolchains and programming environments:

- **GNU Compiler Collection (GCC)**: Open-source compilers (`gcc`, `g++`, `gfortran`).
- **NVIDIA HPC SDK**: Provides `nvc`, `nvc++`, `nvcc`, `nvfortran` with OpenACC and CUDA support, essential for GPU programming on the Booster module.
- **Intel oneAPI**: Compilers (`icc`, `icpc`, `ifort`) optimized for Intel CPUs, along with Intel MPI.

To develop GPU-accelerated code, load the NVIDIA HPC SDK module (e.g. `module load nvhpc`) and ensure that the CUDA runtime libraries are accessible. For traditional MPI applications, load an MPI module compatible with the chosen compiler (e.g. `module load openmpi`).

⚠ Warning: *Intel and NVIDIA Compilers*

Important: Intel compilers do *not* support CUDA or GPU offloading on NVIDIA GPUs. If your code targets NVIDIA GPUs using CUDA, OpenACC, or similar technologies, you must use the NVIDIA HPC SDK tools for compilation.

5.2.3 Job Submission with Slurm

CINECA HPC clusters, such as Leonardo, are shared among many users. As such, responsible usage is essential to ensure fair access and optimal performance for everyone. The system is divided into two main types of nodes: login nodes and compute nodes.

When you log in, you access a **login node**, intended only for light tasks such as editing files, compiling code, or performing brief test runs. Running heavy computations or large parallel jobs directly on it is strictly prohibited. The login environment enforces a hard CPU time limit for any process, and jobs exceeding this limit may be killed without warning. Furthermore, GPUs are not available on login nodes, so you cannot test or run GPU code interactively in this environment.

All resource-intensive and production workloads must be submitted to the **compute nodes** through the ***Slurm*** job scheduler. Slurm manages how resources like CPU cores, GPUs, memory, and temporary storage are allocated to different users. Compute nodes support two main job submission styles: batch mode, in which you provide a job script for Slurm to execute, and interactive mode, where you request an interactive session on a compute node for active development or debugging.

While compute nodes themselves are shared across users, once Slurm assigns resources to a job, those resources are dedicated exclusively to that job for its duration. It is crucial to request only the resources you truly need and to release them promptly to maximize efficiency and minimize costs, since accounting is based on *reserved* resources rather than those you actively use.

Batch Job Submission

A typical batch script for Slurm on Leonardo includes a shebang line, a series of `#SBATCH` directives specifying resource requirements, environment setup commands (loading modules), and finally the execution command. Below is an example for a GPU job:

```
#!/bin/bash

# Slurm directives
#SBATCH --job-name=my_gpu_job       # Job name
#SBATCH --nodes=1                   # Number of nodes
#SBATCH --ntasks-per-node=1         # Number of MPI tasks per node
#SBATCH --cpus-per-task=8           # Number of CPU cores per task
#SBATCH --gres=gpu:4                # Request 4 GPUs per node
#SBATCH --mem=494000                # Memory in MB (approx 480 GB)
#SBATCH --time=00:30:00             # Wall-clock time limit (hh:mm:ss)
#SBATCH --partition=boost_usr_prod  # Partition (queue) to submit to
#SBATCH --account=tra25_gputs       # Account/project code
#SBATCH --reservation=s_tra_gputs   # (Optional) reservation name
#SBATCH --output=job_%j.out         # Standard output file (%j = job ID)
#SBATCH --error=job_%j.err          # Standard error file

# Load necessary modules
module <module-name>

# Run the executable
srun my_application # or mpirun
```

Key directives:

- `--nodes` and `--ntasks-per-node`: Define the number of nodes and MPI ranks per node.
- `--cpus-per-task`: Sets CPU cores per MPI task (useful for hybrid MPI+OpenMP jobs).
- `--gres=gpu:N`: Request `N` GPUs per node. On Booster nodes, typically `gpu:4`.
- `--time`: Maximum wall-clock time. Jobs exceeding this limit are terminated.
- `--partition`: Specifies the queue or partition. Common partitions on Leonardo include `boost_usr_prod` (Booster nodes) and `dcgp_usr_prod` (DHPC nodes).
- `--account`: Your project or account code, used for tracking resource usage and billing.

To submit the job, save the script (e.g., `job.sh`) and run:

```
sbatch job.sh
```

Slurm will return a job ID, which you can use to track the job's status.

Interactive Jobs

For debugging, performance profiling, or exploratory work, you may request an interactive session on compute nodes. Use `salloc` or `srun` with the `--pty` option:

- `srun`:

```
srun -N=1 --ntasks-per-node=1 --cpus-per-task=8 --gres=gpu:1 --time
   =00:10:00 --partition=boost_usr_prod --account=<name> --pty bash
```

  Once the allocation is granted, you will receive an interactive shell on a ***compute node***, which means the resources are allocated and the shell is executed directly on the worker node.

- `salloc`:

```
salloc -N=1 --ntasks-per-node=1 --cpus-per-task=8 --gres=gpu:1 --time
   =00:10:00 --partition=boost_usr_prod --account=<name>
```

  A new session starts on the ***login node***, as `salloc` only performs the resource allocation. Therefore, you must explicitly launch commands (usually via `srun`) to execute code on the allocated compute node(s). Useful to run multiple commands within the same allocation.

Warning: *Exit the allocation*

Remember to exit or cancel the allocation when done to avoid wasting resources.

Monitoring and Managing Jobs

When specifying the `--account` option, you must use your project account. Each account has a budget of core-hours shared among team members. On Leonardo, you can check the balance with:

```
saldo -b          # For Booster accounts
saldo -b --dcgp   # For DCGP accounts
```

Booster and DCGP accounts/resources are separate, and can only be used on their respective partitions. Each **Booster node** provides up to 32 CPU cores, 4 GPUs, and approximately 494 GB RAM. You can allocate resources for your job within these limits, but the product `ntasks-per-node` × `cpus-per-task` must not exceed 32. Accounting for allocation will consider the number of CPUs, GPUs, and RAM you request.

To monitor and control your jobs, you can use the following SLURM commands:

- `squeue -u <username>` or `squeue --me`: Shows the list of all your scheduled jobs, along with their status (pending, running, closing, …). Also displays the jobID.
- `scontrol show job <jobid>`: Provides a long list of information for the job requested, including resource allocation, start time, and node assignments. If the job is not running yet, you'll be notified about the reason and, for top priority jobs, an *estimated start time* is provided.
- `scancel <jobid>`: Removes (cancels) the job (queued or running) from the scheduled job list.
- `sacct <options> <jobid>` (e.g., `sacct -Bj <jobid>`): Displays accounting data for all jobs and job steps in the SLURM log or database. Useful for post-mortem analysis of resource usage.
- `sinfo` (e.g., `sinfo -o "%10D %a %20F %P"`): Provides information about SLURM nodes and partitions.

6 GPUs and Accelerated Computing

6.1 Introduction to Accelerated Computing

The constant demand for computational power in science and engineering has driven the evolution of computer architecture. For decades, performance gains were fueled by increasing clock frequencies and instruction-level parallelism, but physical limitations have shifted the focus toward massive parallelism. This paradigm shift requires a corresponding evolution in programming models to effectively harness the capabilities of modern hardware.

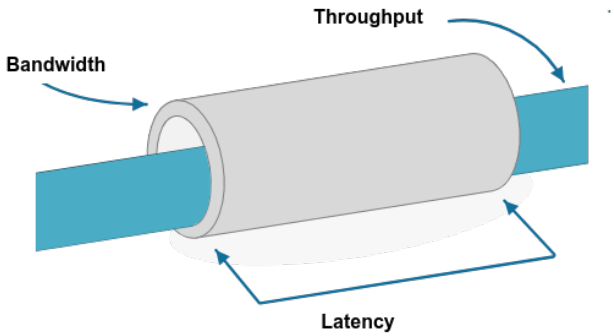
This chapter introduces *accelerated computing*, a paradigm where a specialized, highly parallel processing unit, the Graphics Processing Unit (GPU), is used in conjunction with a traditional Central Processing Unit (CPU) to accelerate computationally intensive portions of an application. We will explore the architectural differences between CPUs and GPUs, delve into the CUDA C/C++ programming model for NVIDIA GPUs, and discuss fundamental concepts such as memory hierarchies, thread organization, and performance optimization.

Key Performance Metrics

When evaluating system performance, three metrics are crucial:

- **Latency**: The time it takes to perform a single operation or access a single piece of data (e.g., time to fetch a value from memory). Measured in seconds.
- **Bandwidth**: The rate at which data can be transferred. Measured in bytes per second.
- **Throughput**: The total number of operations completed in a given amount of time. Measured Floating-Point Operations Per Second (FLOPS).

CPUs are optimized for low latency, while GPUs are designed for high throughput and high memory bandwidth, sacrificing single-thread latency to achieve massive parallelism.



Warning: *Architecture and Programming Models*

A fundamental principle in HPC is that changes in hardware architecture necessitate changes in programming models to achieve efficiency. Code written for a latency-oriented CPU will not perform well on a throughput-oriented GPU.

6.1.1 Architectural Design Philosophies: Latency vs. Throughput

CPUs and GPUs are designed with fundamentally different goals, their architectures reflect this.

Latency-Oriented Design (CPUs)

A CPU is a **latency-oriented** device, designed to execute a single thread of instructions as quickly as possible:

- **Powerful Cores**: A CPU has a small number of powerful, general-purpose cores capable of complex control flow and integer/floating-point arithmetic.
- **Large Caches**: It employs a deep, multi-level cache hierarchy (L1, L2, L3) to minimize memory latency. The goal is to keep data as close to the core as possible, to reduce waiting time for data from main memory.
- **Sophisticated Control Logic**: Features as branch prediction, speculative execution are used to maximize performance.

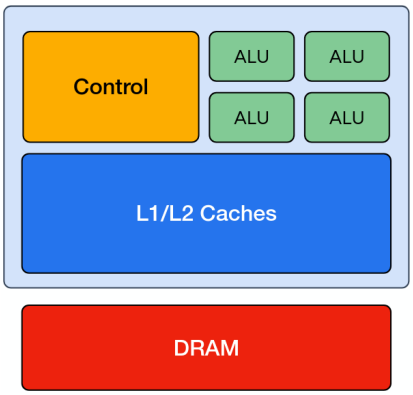


Figure 6.1: Simplified CPU [4]

This design excels at serial, irregular workloads where instructions may depend on previous ones.

Modern architectures rely on a memory hierarchy to balance the trade-off between fast, small, and expensive memory (CPU caches) and large, slower, but cheaper memory (DRAM or storage).

Figure 6.2: Memory Hierarchy [4]

Virtual Memory: The Great Abstraction

Virtual memory is a foundational abstraction in all modern CPUs, allowing each process to believe it has access to a private, contiguous range of addresses, providing several advantages:

- **Simplified Programming Model**: Programs can use memory addresses freely.
- **Isolation**: Each process is protected from accidental or malicious access to others' memory.
- **Efficient Resource Utilization**: The OS can move data between physical memory and storage transparently.

The virtual address used by a program is translated into a physical address by the CPU hardware (the Memory Management Unit, MMU) using page tables maintained by the operating system.

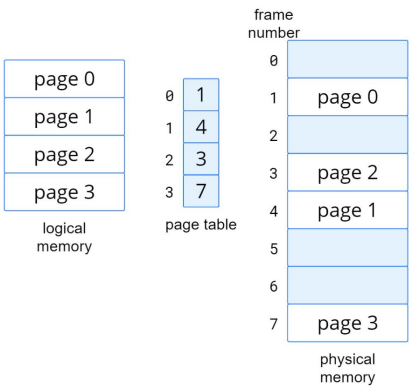


Figure 6.3: Translation of virtual to physical addresses.

Warning:

Page Faults and Swapping If a program accesses memory that is not currently mapped to physical RAM, a **page fault** occurs. The OS may need to load the required page from disk (swap space), causing a major slowdown.

When a CPU needs data, it first checks the fastest cache (L1), then proceeds to L2, L3, and finally DRAM if necessary. The principle of **locality** (recently or nearby accessed data being more likely to be reused) underpins cache design.

Caching increases effective memory bandwidth and reduces average memory access latency—key for maximizing CPU (and to a certain extent, GPU) performance.

From single core to multicore

The end of frequency scaling pushed CPU vendors towards **multicore** designs, where several physical cores share the package and (typically) the last-level cache. Each core retains the latency-oriented features described above; what changes is that the chip now exposes *thread-level* parallelism in addition to instruction-level parallelism.

A few consequences are worth keeping in mind when reasoning about CPU-side performance:

- **Shared last-level cache**. L1 and (usually) L2 are private per core, while the L3 is shared. Threads cooperating on the same data benefit from the shared cache, but threads with disjoint working sets compete for capacity.
- **Cache coherence**. Hardware protocols (e.g. MESI) keep private caches consistent across cores, so a write by one core is eventually visible to the others. Coherence has a cost: ping-ponging a cache line between cores (*false sharing*) is one of the most common performance pitfalls in multithreaded code.
- **SMT / hyperthreading**. Each physical core often exposes two logical threads that share the execution resources. SMT helps hide short stalls but does *not* double throughput.
- **NUMA**. On multi-socket nodes, memory is partitioned across sockets and access to a remote socket is slower than to the local one. We discussed first-touch placement and pinning in chapter 3; the same considerations apply whenever a CPU thread feeds work to a GPU.

This is the architectural baseline against which a GPU is compared in the next subsection: many simple cores in place of a few sophisticated ones, no per-thread cache, and a programming model that explicitly partitions work across thousands of execution contexts rather than relying on the hardware to extract parallelism from a serial instruction stream.

Throughput-Oriented Design (GPUs)

A GPU, in contrast, is a **throughput-oriented** device, designed to execute thousands of parallel threads simultaneously. Its architecture is specialized for data-parallel tasks:

- **Many Simple Cores**: A GPU contains thousands of simpler, more energy-efficient cores grouped into *Streaming Multiprocessors* (SMs).
- **Smaller Caches, High-Bandwidth Memory**: While GPUs have caches, they are smaller relative to the number of cores. The design prioritizes high memory bandwidth to feed the massive number of cores with data concurrently. The philosophy is to ”hide” latency by having other work to do. While one group of threads waits for data, the SM switches to another group that is ready to execute.
- **Simple Control Logic**: GPUs use simpler control logic and are optimized for executing the same instruction on multiple data elements (a model known as SIMT, or Single Instruction, Multiple Threads).

This design is ideal for workloads where the same computation can be performed independently on many different data points, such as matrix multiplication, image processing, and scientific simulations.

6.2 The GPU Architecture

In a typical accelerated node, the CPU is referred to as the **host** and the GPU as the **device**. They are physically distinct processors with their own memory spaces, connected by a high-speed interconnect like the PCIe bus. A key task in accelerated computing is managing the transfer of data and control between the host and device.

GPUs are designed to execute a massive number of threads simultaneously. In the CUDA model, threads are launched in groups of 32, known as **warps**. All threads in a warp execute the same instruction at the same time, making them the fundamental unit of scheduling on the GPU. This execution model is highly efficient for data-parallel problems but requires careful programming to avoid performance pitfalls.

6.2.1 NVIDIA Ampere Architecture

The NVIDIA A100 GPU, used in the Leonardo supercomputer, is based on the **Ampere architecture**. A full Ampere GPU consists of several **Graphics Processing Clusters (GPCs)**, which are further subdivided into **Streaming Multiprocessors (SMs)**. The SM is the core processing unit of the GPU.

Figure 6.4: High-level overview of the NVIDIA Ampere A100 GPU architecture, showing the arrangement of GPCs and SMs.

The **Streaming Multiprocessor (SM)** is the heart of the NVIDIA GPU. In the Ampere architecture, each SM is partitioned into four processing blocks, each with its own instruction buffer, warp scheduler, and dispatch units. This allows the SM to manage and execute multiple warps concurrently.

Key components within an Ampere SM include:

- **FP32/FP64 Cores**: CUDA cores dedicated to single- and double-precision floating-point arithmetic.
- **Tensor Cores (3rd Gen)**: Specialized units that accelerate mixed-precision matrix multiply-accumulate operations, crucial for AI and deep learning workloads. They support new data formats like TF32 and BFloat16.
- **Shared Memory and L1 Cache**: A configurable block of high-speed on-chip memory that can be used as a software-managed cache (`L1`) or a scratchpad for thread cooperation (`shared memory`).
- **Warp Schedulers**: Hardware units that select which warps are ready to execute their next instruction.

The SM is designed to keep its execution units busy by rapidly context-switching between different warps. If one warp stalls (e.g., waiting for data from global memory), the scheduler immediately switches to another resident warp that is ready to execute, thereby hiding memory latency and maximizing computational throughput.

Figure 6.5: Simplified block diagram of an NVIDIA Ampere Streaming Multiprocessor (SM).

6.2.2 Leonardo as a reference machine

Throughout this chapter we will use the **Leonardo** pre-exascale system at CINECA as the reference accelerated machine, since it is the platform the course's exercises target. Each *booster* node packs:

- 1 × Intel Xeon Platinum 8358 CPU (32 cores, host side);
- 4 × NVIDIA A100 64 GB SXM GPUs (Ampere, compute capability 8.0), interconnected via NVLink so that GPU-to-GPU traffic stays inside the node without going over PCIe;
- 512 GB of host DRAM and a 2 × 100 Gb/s InfiniBand HDR fabric for the inter-node network.

On the software side, the NVIDIA HPC SDK 24.3 module exposes `nvc` / `nvc++` / `nvfortran` compilers, CUDA, and the Nsight tools. To get an interactive shell on a GPU node from the login node, the course provides a small wrapper script:

```
# from the login node, source the helper and run get_gpu
$ source get_gpu
$ get_gpu      # 10 min, 1 node, 1 GPU, partition boost_usr_prod
```

which under the hood runs an `srun` with `--gres=gpu:1` on the `boost_usr_prod` partition. All Nsight, `nvcc` and `nvc` commands shown later in this chapter assume the user is sitting on such an interactive node, with the SDK module loaded.

Tip: ***Compute capability***

The `-arch=sm_80` flag mentioned later in this chapter is what selects the Ampere instruction set (the A100). On older or newer GPUs the suffix changes (e.g. `sm_75` for Turing, `sm_90` for Hopper); the value can also be set to `native` when compiling on the same machine that will run the binary.

6.3 CUDA Programming Model

CUDA is NVIDIA's C/C++ extension for programming the device. The model rests on three pillars: *function qualifiers* that say where a function lives, the *kernel-launch syntax* that schedules a function on the GPU, and the runtime API for moving data between host and device. We start with the syntax and then look at how errors are reported, since CUDA's asynchronous execution makes error handling non-trivial.

6.3.1 Function qualifiers and kernel launches

A function annotated with one of CUDA's qualifiers is compiled for the corresponding side of the device boundary:

- `__global__` — the function is a **kernel**: callable from the host (and, on recent GPUs, from the device), executed on the device. It must return `void`.
- `__device__` — callable from device code only, executed on the device. Used for helper routines that run inside a kernel.
- `__host__` — ordinary host function. Implicit when no qualifier is given.
- `__host__ __device__` — compiled for both sides; useful for utilities that need to be reused on host and device.

A kernel is launched with the `<<< grid, block >>>` *execution-configuration* syntax, which fixes how many thread blocks and how many threads per block the GPU will instantiate:

```
__global__ void GPUFunction() {
    printf("Hello from GPU!\n");
}

int main() {
    CPUFunction();
    GPUFunction<<<1, 1>>>();    // 1 block, 1 thread
    cudaDeviceSynchronize();    // wait for the GPU to finish
}
```

A CUDA kernel runs as a *grid* of threads, all executing the same code. Each thread sees a unique identifier through `threadIdx`, `blockIdx` and `blockDim`; we use it in the next section to map threads to data elements.

6.3.2 SIMT vs SIMD

In cuda programming, we use the SIMT (Single Instruction Multiple Thread) model.

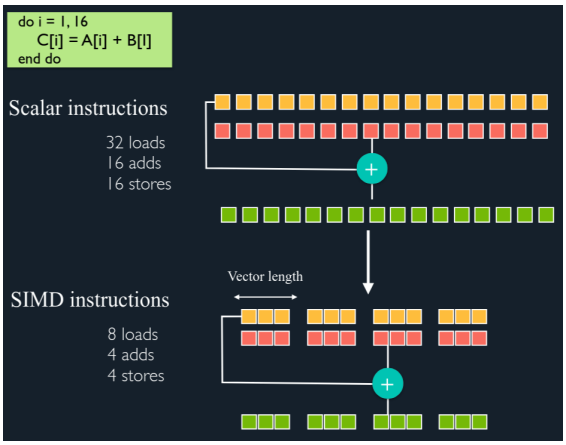


Figure 6.6: SIMT vs SIMD

- **SIMD**: "One instruction, one data chunk."
  - A single instruction operates on multiple data elements simultaneously
  - Requires wide vector units in hardware (e.g., 4 or 8 lanes)
  - A SIMD register (or a vector register) can hold many values (2 - 16 values or more) of a single type
  - Operates on packed data in wide registers (e.g., 128-bit or 256-bit)
  - All elements in the vector must follow the same control flow—no divergence
  - Vectorisation helps you write code which has good access patterns to maximise bandwidth
- **SIMT** "Single Instruction, Multiple Threads".
  - Uses scalar execution units, not wide vectors, rigid, lockstep vector processing
  - 32 threads in a warp share a single instruction fetch, executed over multiple cycles (e.g., 4 cycles on 8 CUDA cores)
  - Flexible, thread-level parallelism with divergence support.
  - Single instruction, multiple flow paths if statements are allowed!

SIMT allows CUDA GPU to perform "vector" computations on scalar cores. Much easier to vectorise than getting compiler to autovectorize on CPU.

6.3.3 Compilation with nvcc

CUDA source files (`.cu`) mix host and device code. The NVIDIA compiler driver `nvcc` separates them: device-side code is compiled to PTX (a virtual ISA) and then to a target-specific binary, while host-side code is delegated to the system compiler (`gcc` / `clang`). A typical invocation on Leonardo is

```
1 nvcc -arch=sm_80 -O3 -o vecadd vecadd.cu
```

- `-arch=sm_80` selects the target compute capability (Ampere = 8.0). Without this flag `nvcc` compiles for an older default and you may miss hardware features.
- `-O3` enables host-side optimisation (device-side optimisation is on by default).
- `-run` (used in many of the course examples) compiles *and* runs the resulting binary, which is convenient for one-off experiments.

6.3.4 Error handling and the asynchronous error model

Every CUDA Runtime API call returns a value of type `cudaError_t`; `cudaSuccess` means everything went well, anything else is an error code that `cudaGetErrorString()` translates into a human-readable string. The synchronous case is straightforward:

```
cudaError_t err = cudaMallocManaged(&a, size);
if (err != cudaSuccess)
    fprintf(stderr, "alloc error: %s\n", cudaGetErrorString(err));
```

Kernel launches are different: they are **asynchronous**. The `<<< … >>>` call enqueues the kernel and returns immediately, even if the kernel cannot start (e.g. because the launch configuration is invalid) or fails later (e.g. a segmentation fault on the device). For that reason CUDA exposes *two* layers of error reporting around a launch:

```
myKernel<<<blocks, threads>>>(args);

// 1) launch-time errors (bad configuration, too much shared memory, ...)
cudaError_t launchErr = cudaGetLastError();
if (launchErr != cudaSuccess) { /* handle */ }

// 2) execution-time errors (only visible at the next sync point)
cudaError_t runErr = cudaDeviceSynchronize();
if (runErr != cudaSuccess) { /* handle */ }
```

`cudaGetLastError()` clears the sticky error flag, so calling it once after each launch is the safe pattern. Run-time faults inside a kernel only surface at the next synchronisation, which is why projects routinely wrap launches in a `CHECK_CUDA` macro that calls both functions.

Tip: ***Debugging tip***

Setting the environment variable `CUDA_LAUNCH_BLOCKING=1` forces every kernel launch to be synchronous, so a crash points exactly at the offending kernel instead of at a far-away `cudaDeviceSynchronize()`. Do this for debugging only — it serialises all kernels and destroys the latency-hiding the GPU relies on.

6.4 GPU Thread Hierarchy

A gpu consists of thousands of grids, each one containing thousands of thread blocks (teams), each one containing 1024 threads divided into warps of 32 threads each.

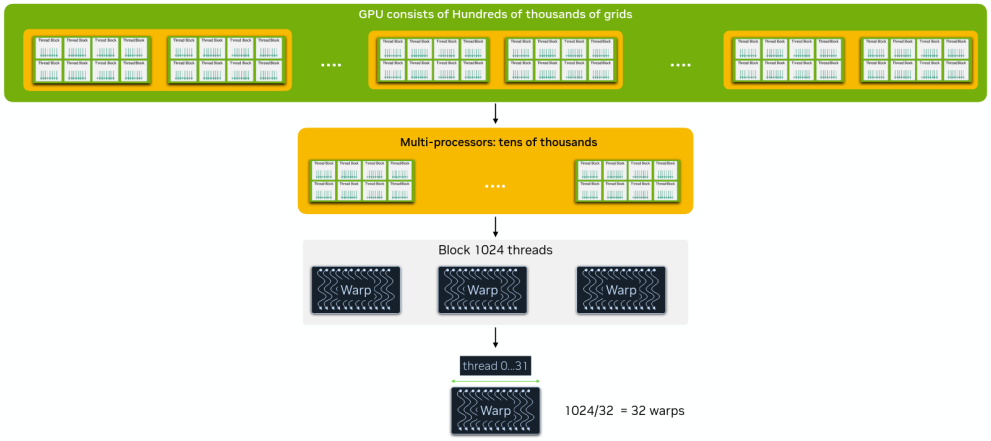


Figure 6.7: GPU Thread Hierarchy

All thread blocks in a grid must have the same number of threads, ensuring homogeneity across the GPU workload. To process *N* elements in parallel, we need to launch at least *N* concurrent threads on the device. Threads within the same block (or team) can efficiently cooperate by exchanging data through a shared memory cache. Importantly, each block is executed independently, and there is no guarantee on the order in which blocks are scheduled or executed by the GPU.

CPUs and GPUs have physically distinct memory regions connected by the PCIe bus.

Every time we want to compute something on a GPU we need to perform some steps:

- Allocate GPU memory
- Copy data from the host to the GPU
- Load GPU program and execute, caching data on chip for performance
- Copy results from the GPU memory to the host memory
- Free the GPU memory

We can see how the code for cpu is very similar to the one for gpu, but with the addition of the cuda functions.

CPU code

```
int N = 10000;
size_t size = N * sizeof(int);

int *a;
a = (int *) malloc(size);

free(a);
```

GPU code

```
int N = 10000;
size_t size = N * sizeof(int);

int *a;
cudaMallocManaged(&a, size);

cudaFree(a);
```

Choosing the optimal grid size

Let's start by choosing the ***optimal block size***. To do that, we need to write an execution configuration that creates more threads than necessary, then pass a value as an argument into the kernel (N) that represents that total size if the data set to be processed/total threads needed to complete the work.

Then, we need to calculate the global index and if it does not exceed N perform the kernel work.

```
__global__ vectorSum(int N) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) { // only do work if it does}
}
```

Maximum size at each level of the thread hierarchy is device dependent.

On A100 typicaly you get:

- Maximum number of threads per block: 1024
- Maximum sizes of x-, y-, and -z dimensions of threads block: 1024 x 1024 x 64
- Maximum sizes of each dimension of grid of thread blocks: 65535 x 65535 x 65535 (about 280,000 billion blocks)

The best performance is achieved for blocks that contain a number of threads that is a multiple of 32, due to GPU hardware traits.

Example: *Choosing the optimal block size*

We want to run 1000 parallel task with blocks containing 256 threads. How do we choose the optimal block size?

```
int N = 100000; size_t threads_per_blocks = 256;
size_t number_of_blocks = (N + threads_per_block - 1) /
    threads_per_block;
kernel<<<number_of_blocks, threads_per_block>>>(N);
```

The ”-1” term is added to round up the division if necessary.

A limited number of threads (1024) can fit inside a thread block, so to increase parallelism we coordinate work across many blocks. Each thread computes its global index and works on the corresponding data element:

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
```

6.4.1 Grid-stride loops

The ”one-thread, one-element” pattern works well when the number of launched threads matches N, but it ties the kernel’s correctness to a specific launch configuration. A more flexible idiom is the **grid-stride loop**: each thread starts at its global index and walks through the array in steps of `blockDim.x * gridDim.x`, the total number of threads in the grid.

```
__global__ void vectorAdd(const float* a, const float* b, float* c, int
    n) {
    int idx    = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = idx; i < n; i += stride)
        c[i] = a[i] + b[i];
}
```

The kernel now works correctly for *any* grid size: launch a tiny grid and each thread processes many elements; launch a large grid and each thread processes one. This decouples the kernel from the launch configuration and is the recommended idiom in modern CUDA. The course example

at `cuda/source/vectorAddstride/testStrideNostride.cu` compares the two patterns under three regimes (small vector, large vector with enough threads, large vector with too few blocks) and shows that the strided version stays performant in the under-provisioned case where the non-strided kernel simply does not cover the array.

Observation: *Why "grid-stride", not "block-stride"?*

The stride is the number of threads in the entire grid, not just in one block. This makes consecutive iterations of the loop in different threads access *contiguous* memory addresses — exactly the pattern the GPU coalesces into a single transaction (see section 6.5).

6.5 CUDA Memory Hierarchy

The execution model only tells half the story: a kernel is fast only as long as the threads can be fed with data. The GPU exposes a small zoo of memory spaces with very different bandwidth, latency, scope and lifetime, and choosing the right one is the single largest optimisation lever available. The figure below summarises the on-chip / off-chip layout of an SM, and the small table beside it shows the rough bandwidth and latency ratios between the three main tiers (registers, shared memory / L1, global memory).

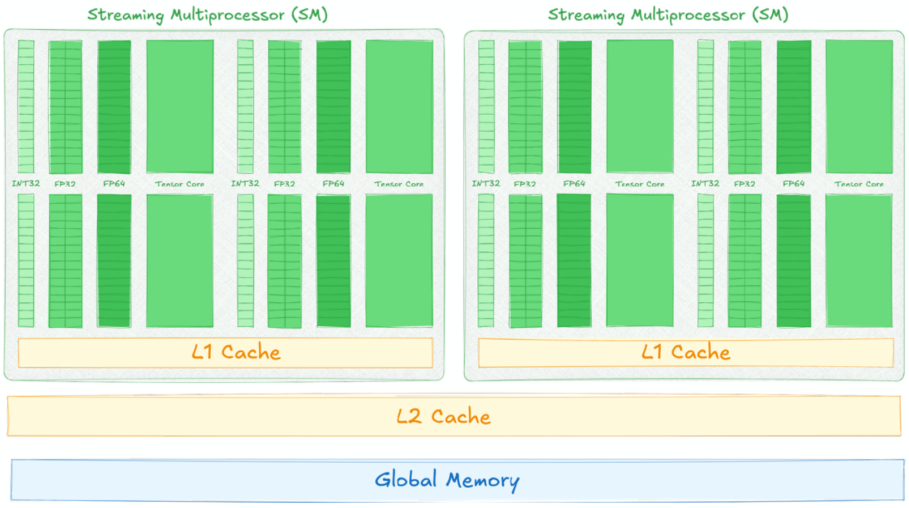


Figure 6.8: CUDA memory hierarchy.

Bandwidth	Latency
13×	1×
3×	5×
1×	15×

Figure 6.9: Bandwidth and latency, normalised.

The four memory spaces a CUDA programmer normally interacts with are:

- **Global memory**: the off-chip DRAM, the largest space (tens of GB on Ampere) but also the slowest. *Scope*: accessible by all threads of all blocks and grids. *Usage*: `cudaMalloc` / `cudaMemcpy` .
- **Shared memory**: a small (up to 164 KB per SM on A100), fast on-chip scratchpad. *Scope*: shared by all threads within a block. *Usage*: declared with the `__shared__` qualifier; used to cache data for reuse inside a block (see 6.5.2).
- **Registers**: the fastest storage on the GPU, mapped to the SM's register file. *Scope*: private to each thread. *Usage*: automatic variables inside a kernel. Spilling to "local" memory (which is really global) happens silently when register pressure is too high, and tanks performance — watch the compiler report.
- **Constant memory**: a 64 KB read-only space, cached and optimised for the case where all threads in a warp read the same address. *Scope*: read-only from device code. *Usage*: `__constant__` qualifier.

6.5.1 Coalesced access in practice

The Ampere memory subsystem services global-memory transactions in 32-, 64- or 128-byte segments. When the threads of a warp request *contiguous* addresses inside a single segment, the hardware merges them into one transaction — a **coalesced** access. When the addresses are spread across multiple segments, the hardware issues several transactions and bandwidth utilisation drops accordingly.

Tip: ***Memory coalescing***

For optimal performance, threads in a warp should access consecutive memory addresses. Coalesced access can improve bandwidth utilisation by up to 10× compared to non-coalesced patterns.

The classic counter-example is a transposed access pattern. Reading a row of a matrix (`A[row*N + col]` , with `col` varying with `threadIdx.x`) is coalesced; reading a column (`A[col*N + row]`) is not, because successive threads jump by N floats. The course example at `cuda/source/matrixTranspose/matrixTranspose.cu` measures both patterns and shows the bandwidth gap.

6.5.2 Shared memory and tiling

Shared memory shines when the same global-memory value is read by many threads of a block — which is exactly what happens in matrix multiplication: every element of A contributes to a whole row of C, and every element of B to a whole column. The **tiled matmul** pattern exploits this by loading TILE×TILE blocks of A and B into shared memory cooperatively, reusing them many times, and then advancing along the K dimension.

A trimmed version of the kernel from `cuda/source/Global_Shared_streaming/matmul_shared.cu` (`TILE_SIZE = 16`):

```
__global__ void matmul_shared(const float* A, const float* B,
                              float* C, int M, int N, int K) {
    __shared__ float s_A[TILE_SIZE][TILE_SIZE];
    __shared__ float s_B[TILE_SIZE][TILE_SIZE];

    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;
    float sum = 0.0f;

    for (int k = 0; k < K; k += TILE_SIZE) {
        s_A[threadIdx.y][threadIdx.x] = A[row*K + k + threadIdx.x];
        s_B[threadIdx.y][threadIdx.x] = B[(k + threadIdx.y)*N + col];
        __syncthreads();                        // tile fully loaded

        for (int i = 0; i < TILE_SIZE; ++i)
            sum += s_A[threadIdx.y][i] * s_B[i][threadIdx.x];
        __syncthreads();                        // before overwriting
    }
    if (row < M && col < N) C[row*N + col] = sum;
}
```

Two ideas are at play here. The cooperative load amortises each global-memory read over `TILE_SIZE` multiply-adds, turning a memory-bound kernel into a compute-bound one. And

the two `__syncthreads()` barriers separate the two phases of the loop body: *all* threads must finish loading the tile before *any* thread starts using it, and *all* threads must finish using it before *any* thread overwrites it on the next iteration.

⚠ Warning: *Shared memory bank conflicts*

Shared memory is split into 32 *banks*; an access by a warp is conflict-free only if at most one thread accesses each bank (or all threads access the same address, which is broadcast). A common cause of conflicts is column-major access into a 32-wide shared array; the standard fix is to over-pad the array (e.g. `__shared__ float s[32][33]`) so that successive columns fall into different banks.

6.5.3 Host memory: pageable, pinned, managed

Allocations on the *host* side also have a performance dimension. The runtime offers three flavours:

- **Pageable** memory — ordinary `malloc` / `new`. The OS may page it out, so before each `cudaMemcpy` the runtime has to copy the buffer into an internal pinned staging area. This double-copy roughly halves the achievable PCIe bandwidth.
- **Pinned** (page-locked) memory — allocated with `cudaMallocHost` or `cudaHostAlloc`. The OS cannot page it out, so the GPU's DMA engine reads directly from it. Transfer bandwidth is typically 1.5×–2× that of pageable memory, and pinned buffers are a prerequisite for *asynchronous* `cudaMemcpyAsync` (see section on streams).
- **Managed** memory — allocated with `cudaMallocManaged`. A single virtual address is valid on both host and device; the driver migrates pages on demand. Convenient (one allocator, no explicit `cudaMemcpy`) but the migration is implicit, which makes performance harder to predict.

```
1 float *pageable = (float*)malloc(bytes);              // CPU only
2 float *pinned;   cudaMallocHost(&pinned, bytes);      // page-locked
3 float *managed;  cudaMallocManaged(&managed, bytes);  // unified
```

The course example at `cuda/source/vecAdd_pinnedpeageble/vecAdd_managed.cu` (the directory name is a typo for "pinned/pageable") benchmarks pageable vs. pinned transfers side-by-side; the managed version lives at `cuda/source/vecAddtion_managed/vecAdd_managed.cu`.

6.5.4 Unified memory and page faults

When a kernel touches a managed-memory page that is currently resident on the host, the GPU raises a **page fault**: execution stalls, the driver migrates the page over PCIe, and only then does the kernel resume. The first kernel after an allocation pays this cost on every page; subsequent kernels see the data already on the device and run at full speed.

To avoid surprises, CUDA exposes two hint APIs:

- `cudaMemPrefetchAsync(ptr, size, deviceId)` — asynchronously migrate a range to a target device (use `cudaCpuDeviceId` for the host) before it is needed. This is the cheapest way to make managed memory perform like an explicit `cudaMemcpy`.
- `cudaMemAdvise(ptr, size, advice, deviceId)` — tell the driver about an access pattern, e.g. `cudaMemAdviseSetReadMostly` or `cudaMemAdviseSetPreferredLocation`. The driver uses these as scheduling hints, not as commands.

Example: *Prefetching managed buffers before launch*

```
cudaMallocManaged(&a, size);
cudaMallocManaged(&b, size);
cudaMallocManaged(&c, size);

cudaMemPrefetchAsync(a, size, deviceId);   // warm the GPU
cudaMemPrefetchAsync(b, size, deviceId);
cudaMemPrefetchAsync(c, size, deviceId);

addVectors<<<blocks, threads>>>(c, a, b, N);
cudaDeviceSynchronize();

cudaMemPrefetchAsync(c, size, cudaCpuDeviceId);  // bring result home
```

This is the pattern used in `cuda/solution/04-stream/stream1.cu`.

6.6 Asynchronous Execution and CUDA Streams

So far we have launched kernels on the implicit *default stream*: every operation issued by the host is appended to a single FIFO queue and executed in order. The default stream is convenient but it forbids exactly the kind of overlap a GPU can exploit — copying the next batch of data while the current batch is being processed, or running independent kernels side by side.

A **CUDA stream** is just an ordered queue of operations. Operations within a stream execute in order; operations in *different* streams may run concurrently if the hardware has the resources for it. Creating, using and destroying a stream is straightforward:

```
cudaStream_t stream;
cudaStreamCreate(&stream);

someKernel<<<numberOfBlocks, numberOfThreads, 0, stream>>>();   // 4th launch arg

cudaStreamDestroy(stream);
```

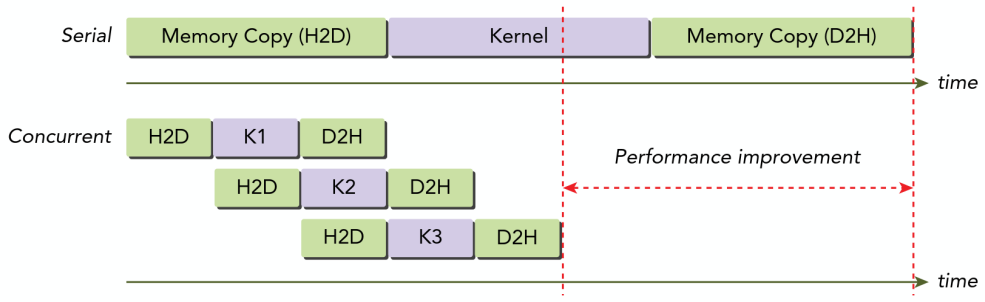


Figure 6.10: Concurrent execution of kernels and transfers using multiple streams.

6.6.1 Overlapping H2D, kernel and D2H

The motivating use case for streams is the classical pipeline: split a large array into chunks, and for each chunk overlap the host-to-device copy of one chunk with the kernel execution of the previous chunk and the device-to-host copy of the chunk before that. This requires two ingredients:

- a non-blocking copy primitive — `cudaMemcpyAsync(dst, src, n, kind, stream)`, which enqueues the transfer on `stream` and returns immediately;
- *pinned* host memory, otherwise the runtime cannot DMA directly and silently downgrades the call to a synchronous copy.

```
const int nStreams = 4, chunk = N / nStreams;
cudaStream_t s[nStreams];
for (int i = 0; i < nStreams; ++i) cudaStreamCreate(&s[i]);

for (int i = 0; i < nStreams; ++i) {
    int off = i * chunk;
    cudaMemcpyAsync(d_x + off, h_x + off, chunk*sizeof(float),
                    cudaMemcpyHostToDevice, s[i]);
    kernel<<<blocks, threads, 0, s[i]>>>(d_y + off, d_x + off, chunk);
    cudaMemcpyAsync(h_y + off, d_y + off, chunk*sizeof(float),
                    cudaMemcpyDeviceToHost, s[i]);
}
for (int i = 0; i < nStreams; ++i) cudaStreamDestroy(s[i]);
```

The course solution at `cuda/solution/04-stream/stream1.cu` applies the same idea to a vector-addition workload, using `cudaMallocManaged` + `cudaMemPrefetchAsync` on three streams to overlap the initialisation of `a`, `b` and `c` before adding them.

6.6.2 Synchronisation: do not over-synchronise

Streams give the GPU permission to overlap work; you still need synchronisation to know when results are safe to read on the host. Two primitives cover most cases:

- `cudaStreamSynchronize(stream)` — block the host until *this* stream has drained. Other streams keep running.
- `cudaDeviceSynchronize()` — block until *all* streams on the current device have drained. Heavy-handed, but necessary at the very end of a phase or before tearing down resources.

A common mistake is to scatter `cudaDeviceSynchronize()` between operations "to be safe": each call effectively serialises the GPU and makes streams pointless. Prefer per-stream synchronisation, and let the runtime keep operations from *different* streams overlapping.

Tip: ***Stream callbacks and events***

For finer-grained synchronisation, CUDA also exposes `cudaEvent_t` (timestamps that can be recorded into a stream and queried/waited on later) and `cudaStreamAddCallback` (a host function fired when the stream reaches that point). Events are also the right tool for measuring kernel execution time, as in the `matmul_shared.cu` example.

Advanced Concept: *Non-blocking streams*

`cudaStreamCreate` creates a stream that synchronises with the default (legacy) stream. The `cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking)` variant decouples it from the default stream, which is preferable when mixing default-stream code (e.g. from a library) with explicitly-managed streams. Not covered in detail in this course.

6.7 Profiling Workflow

A typical optimisation cycle on the GPU starts from a CPU baseline and proceeds in two passes: first a *system-wide* profile to understand where time is spent (CPU vs. GPU, transfer vs. compute, idle gaps in the timeline), then a *per-kernel* deep-dive on the worst offender. The two NVIDIA tools that map onto these passes are:

- **Nsight Systems** (`nsys`) — a low-overhead system-level tracer. It produces a timeline showing kernels, memory transfers, OS-level CPU activity and (optionally) MPI / NVTX ranges. This is what you look at first to confirm that the GPU is busy when it should be and that transfers are overlapping with compute.
- **Nsight Compute** (`ncu`) — a per-kernel hardware-counter profiler. It reports occupancy, achieved memory bandwidth, instruction mix, warp stall reasons and many other metrics, but at the cost of replaying each kernel several times. Use it after Nsight Systems has identified the kernel that matters.

6.7.1 Capturing a profile

The minimal Nsight Systems invocation traces CUDA and NVTX activity and writes a `.nsys-rep` file that the `nsys-ui` GUI (or `nsys stats`) can open later:

```
nsys profile -t cuda,nvtx --stats=true --force-overwrite=true \
             -o report ./my_program
```

The `-t` flag selects which traces to collect (add `mpi` or `openacc` as needed); `--stats=true` prints a summary on the console after the run, which is usually enough for a first triage.

For a full per-kernel report, switch to Nsight Compute:

```
ncu --set full -o report ./my_program
```

The `--set full` preset enables every metric group; cheaper presets like `--set basic` or `--set roofline` are available when only a subset is needed. Both tools require write access to GPU performance counters; on Leonardo this is granted automatically inside a `boost_usr_prod` job.

6.7.2 Annotating the timeline with NVTX

Without help, an Nsight Systems timeline is a sea of unnamed CUDA API calls. The **NVIDIA Tools Extension** (NVTX) library lets you mark regions of host code so that they appear as named ranges on the timeline, which makes correlating profile features with source code trivial. The pattern is just push/pop:

```
#include <nvtx3/nvToolsExt.h>

nvtxRangePushA("H2D Transfer");
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);
nvtxRangePop();

nvtxRangePushA("MatMul Kernel (Shared Memory)");
matmul_shared<<<grid, block>>>(d_A, d_B, d_C, M, N, K);
cudaDeviceSynchronize();
nvtxRangePop();
```

This snippet is condensed from `cuda/source/Global_Shared_streaming/matmul_shared.cu`, where every phase (H2D, kernel, D2H, CPU computation) is wrapped in its own NVTX range. The corresponding `.nsys-rep` files in that directory show exactly how the labelled bands stack up on the timeline.

6.7.3 What to look at first

Tip: ***Profiler triage checklist***

1. **Is the GPU busy?** On the Nsight Systems timeline, a wide band of GPU activity is good; large idle gaps between kernels usually mean over-synchronisation on the host or too few streams.
2. **Transfers vs. compute.** If the H2D / D2H bands are as wide as the kernel band, the kernel is transfer-bound: pin host memory, use `cudaMemcpyAsync` on streams, or move to managed memory with prefetching.
3. **Kernel duration.** Once the timeline looks healthy, sort kernels by total time and pick the heaviest one for an Nsight Compute deep-dive.
4. **Inside Nsight Compute**: check *achieved occupancy* (low values point at register or shared-memory pressure), *memory throughput* as a fraction of peak (low values mean uncoalesced access or insufficient ILP), and the *warp stall reason* histogram (memory dependency, execution dependency, etc.).

Observation: ***A note on the roofline model***

A complementary — and more quantitative — way to read these metrics is the *roofline model*, which plots achieved performance against arithmetic intensity (FLOPs per byte) and bounds it by the peak compute and peak memory-bandwidth lines. We do not develop it here; the `ncu --set roofline` preset already produces the plot for any kernel.

6.8 GPU programming with OpenMP target offload

CUDA gives the programmer full control over the GPU but also full responsibility for managing devices, memory, kernels and streams; and the resulting code is bound to NVIDIA hardware. **OpenMP target offload** (introduced in OpenMP 4.0 and consolidated in 4.5/5.x) takes the opposite trade-off: a small set of pragmas asks the compiler to move a region of code and the data it needs to an attached accelerator, while the rest of the program stays in standard C/C++/Fortran. The same source can be retargeted to a different GPU vendor or fall back to the CPU by changing only compiler flags — at the price of less direct control over the hardware.

This section walks through the constructs that the OpenMP school examples (`GPU/OpenMP_2024/`) exercise, in roughly the same order: *target* first, then *teams/threads*, then data mapping, then asynchrony, then multi-GPU.

6.8.1 The target construct

The basic block is `#pragma omp target` : it marks a structured region whose statements should execute on the device. Scalars and array sections referenced inside the region are mapped to the device automatically (with `tofrom` semantics for arrays, more on this below). The course's `Simple/simple0.c` uses it just to query the GPU:

```
if (omp_get_num_devices() == 0) {
    printf("No accelerator found\n"); exit(1);
}

#pragma omp target device(0)
{
    int nteams   = omp_get_num_teams();
    int nthreads = omp_get_max_threads();
    printf("device 0: %d teams, %d threads/team\n", nteams, nthreads);
}
```

Without further directives the compiler runs the body on a *single* team with a *single* thread — functionally correct, but only a tiny fraction of the GPU is used. To get parallelism we need to combine `target` with the team/thread constructs.

6.8.2 Teams, threads and the SIMT mapping

OpenMP exposes three nested levels of parallelism that map onto the GPU thread hierarchy:

- a **league** of **teams** (`#pragma omp teams`) — each team becomes a CUDA thread block;
- a team's **threads** executing a worksharing loop (`distribute parallel for`) — each thread becomes a CUDA thread inside the block;
- optionally, **SIMD lanes** inside a thread (`simd`) — which on the GPU lines up with how a warp executes a SIMT instruction.

The composite construct that one sees most often in practice is

```
#pragma omp target teams distribute parallel for simd \
        map(to: a, n, x[0:n]) map(tofrom: y[0:n])
for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
```

which is the SAXPY kernel from `VectorMultiplications/saxpi.gpu.simple.c` . Reading the directives outside-in: the `target` sends control to the device; `teams` creates a league of thread blocks; `distribute` hands successive chunks of loop iterations to successive teams; `parallel for` parallelises within each team; and `simd` hints at vectorisation inside a thread.

The number of teams and threads can be left to the compiler or controlled with the `num_teams` , `thread_limit` , `num_threads` and `dist_schedule` clauses. The example progression `saxpi.gpu.v0.c` → `v2.c` explores this experimentally:

- v0: 1 team, 1 thread — correct, painfully slow (left as a baseline).
- v1: a single tunable team count; throughput grows roughly with the number of teams up to the SM count.
- v2: full control over both axes via `num_teams(ial)` and `num_threads(itr)` with `dist_schedule(static, itr)` ; performance is best when the team count saturates the SMs and the thread count is a multiple of 32 — the same warp-friendly constraint we saw in CUDA.

Tip: ***Rule of thumb***

Letting the compiler pick `num_teams` and `thread_limit` is usually within 10–20% of the hand-tuned configuration. Do not micro-tune these clauses unless the profiler says the kernel is launch-bound or you have a specific layout in mind.

6.8.3 Data mapping and persistent device data

By default, every `target` region copies the data it touches onto the device on entry and back on exit. This is safe but expensive: in an iterative algorithm the same arrays bounce across PCIe on every iteration. The `map` clause makes the intent explicit and lets the runtime optimise:

- `map(to: A[0:n])` — copy host → device on entry, do not copy back;
- `map(from: A[0:n])` — allocate on device, copy device → host on exit;
- `map(tofrom: A[0:n])` — both directions (the implicit default for arrays);
- `map(alloc: A[0:n])` — allocate on device, no transfer in either direction.

For arrays that survive across multiple kernels, the right tool is a `target data` region (structured) or a pair of `target enter data` / `target exit data` directives (unstructured), which decouple the data lifetime from any particular kernel. The course's GEMV progression illustrates the pay-off. Version 1 maps the matrix on every call:

```
void gemv(float alpha, float* A, float* x, float beta, float* y,
          size_t rows, size_t cols) {
    #pragma omp target teams distribute \
            map(to: A[0:rows*cols], x[0:cols], alpha, beta) \
            map(from: y[0:rows])
    for (size_t i = 0; i < rows; ++i) { /* ... */ }
}
```

Version 3 hoists the allocation out of the kernel by attaching it to the array's lifetime:

```
float* allocate_and_initialize_array(size_t rows, size_t cols) {
    float* a = (float*)malloc(rows*cols*sizeof(float));
    /* ... initialise on the host ... */
    #pragma omp target enter data map(to: a[:rows*cols])
    return a;
}

void deallocate(float* a, size_t n) {
    #pragma omp target exit data map(delete: a[:n])
    free(a);
}
```

The kernel itself now runs without per-call mapping, and a final `#pragma omp target update from(y[:rows])` brings the result back. On a $32{,}768 \times 32{,}768$ GEMV this turns a transfer-bound run into a compute-bound one.

6.8.4 Asynchronous offload and OpenMP "streams"

A target region without further annotation is synchronous: the host blocks until the kernel finishes. The `nowait` clause turns it into an OpenMP *task* that runs asynchronously, and the host can continue with other work — including launching more target tasks. `taskwait` (or `depend` clauses for fine-grained ordering) re-synchronises.

The course example `Asyncomputing/multiple.streams.c` simulates CUDA streams entirely with tasks: it splits an array into chunks and fires one asynchronous `target` per chunk inside an `omp single` region:

```
#pragma omp parallel
{
    #pragma omp single
    {
        for (int s = 0; s < num_streams; ++s) {
            int start = s * chunk_size;
            int end   = (s == num_streams - 1) ? N : start + chunk_size;
            #pragma omp task firstprivate(start, end) nowait
            gpu_task(data, start, end);   // contains a #pragma omp
               target
        }
    }
    #pragma omp taskwait
}
```

Each task carries its own target region; the runtime is free to schedule them on different CUDA streams under the hood, giving the same overlap of transfers and kernels as explicit `cudaMemcpyAsync` would.

6.8.5 Multi-GPU offload

A node like Leonardo has four A100s; OpenMP exposes them as device IDs $0, 1, \ldots, N-1$. Two ingredients are needed to use more than one:

- the `device(d)` clause on `target` / `target data` / `target enter data` , which selects the GPU that the construct will run on;
- the `#pragma omp declare target` directive, which marks a function (or variable) as callable from device code so that the same routine can be reused across GPUs.

The dual-GPU SAXPY in `ModularProgramming/saxpi.dualgpu.simple.c` packages the per-GPU kernel into a `declare target` function and dispatches it to a chosen device:

```
#pragma omp declare target
int computeLoop(int n, float a, const float *x, float *y, int dev) {
    #pragma omp target data device(dev) \
            map(to: a, n, x[0:n]) map(tofrom: y[0:n])
    {
        #pragma omp target teams distribute parallel for simd device(dev
```

```
            )
        for (int i = 0; i < n; ++i)
            y[i] = a * x[i] + y[i];
    }
    return dev;
}
#pragma omp end declare target
```

Splitting work across GPUs is then just a matter of partitioning the index range and calling `computeLoop` once per device — typically inside an `omp parallel` so the host issues launches on multiple GPUs concurrently.

6.8.6 Compilers and Leonardo

Three compilers support OpenMP target offload to NVIDIA GPUs; the syntax is the same, only the flags change.

```
# NVIDIA HPC SDK
nvc -mp=gpu -gpu=cc80 -O3 -fast -Minfo=all -o prog prog.c

# Clang
clang -fopenmp -fopenmp-targets=nvptx64 \
      -Xopenmp-target -march=sm_80 -o prog prog.c

# GCC offload
gcc -fopenmp -foffload=nvptx-none \
    -foffload-options="-march=sm_80" -o prog prog.c
```

On Leonardo the recipe documented at `GPU/OpenMP_2024/resources/HOWTO_compile.md` is

```
module load nvhpc/24.3 openmpi/4.1.6--nvhpc--24.3

OMP_FLAG="-mp=multicore,gpu"      # OpenMP on both CPU and GPU
GPU_FLAG="-gpu=cc80 -target=gpu -gpu=nomanaged"

nvc -O3 -fast -Minfo=all $GPU_FLAG $OMP_FLAG prog.c -o prog
```

A few environment variables are worth knowing about:

- `OMP_TARGET_OFFLOAD=MANDATORY` — abort if offload is not available, instead of silently falling back to the CPU. Use this in benchmarks.
- `OMP_NUM_TEAMS`, `OMP_TEAM_LIMIT` — runtime override for the team count and the maximum threads per team.
- `OMP_DISPLAY_ENV=true` — prints the OpenMP runtime configuration on startup; useful when you suspect an environment misconfiguration.

6.9 CUDA, OpenMP target, and what comes next

Looking back at the chapter, the two programming models we have seen sit at opposite ends of the productivity / control trade-off. CUDA exposes the hardware almost directly: the programmer chooses the launch configuration, places data in shared memory, schedules transfers on streams, picks pinned versus managed allocations. The reward is full performance and full responsibility,

locked to the NVIDIA stack. OpenMP target offload hides most of those decisions behind a handful of pragmas: the same loop annotated with `target teams distribute parallel for` delivers a respectable fraction of the CUDA performance on the same A100, and could in principle be retargeted to a different vendor by changing only compiler flags.

A third directive-based model, **OpenACC**, occupies roughly the same niche as OpenMP target but with a slightly different vocabulary (`parallel` / `kernels` , `gang` / `worker` / `vector` , `copyin` / `copyout`) and a stronger focus on incremental porting of existing scientific codes. OpenACC is the subject of chapter 7, which also includes a side-by-side comparison of the OpenACC and OpenMP directives and clauses, and a Laplace-equation case study that walks the same algorithm through both ecosystems.

7 OpenACC

```
#pragma acc <directive> <clauses>
```

`#pragma` in C/C++ is what's known as a "compiler hint." These are very similar to programmer comments, however, the compiler will actually read our pragmas.

Pragmas are a way for the programmer to "guide" the compiler, without running the chance damaging the code. If the compiler does not understand the pragma, it can ignore it, rather than throw a syntax error.

`acc` is an addition to our pragma. It specifies that this is an OpenACC pragma. Any non-OpenACC compiler will ignore this pragma. Even the nvc/nvc++ compiler can be told to ignore them. (which lets us run our parallel code sequentially!)

`directives` are commands in OpenACC that will tell the compiler to do some action. For now, we will only use directives that allow the compiler to parallelize our code.

`clauses` are additions/alterations to our directives. These include (but are not limited to) optimizations. The way that I prefer to think about it: directives describe a general action for our compiler to do (such as, paralellize our code), and clauses allow the programmer to be more specific (such as, how we specifically want the code to be parallelized)

Directives

- `parallel`: a parallel region of code. The compiler generates a parallel kernel for that region. Each gang will execute the entire loop.

```
#pragma acc parallel
{
    for (int i = 0; i < N; i++)
        A[i] = B[i] + C[i];
}
```

- `loop`: dentifies a loop that should be distributed across threads. Parallel and loop are often placed together

```
#pragma acc parallel
{
    #pragma acc loop
    for (int i = 0; i < N; i++)
        A[i] = B[i] + C[i];
}
```

- `kernels` a parallel region of code. It allows the programmer to step back, and rely solely on the compiler.

```
// example of a kernel that performs two different operations
#pragma acc kernels
{
    for (int i = 0; i < N; i++)
        A[i] = B[i] + C[i];

    for (int i = 0; i < N; i++)
        D[i] = A*x[i] + B*y[i];
}

// loop independent: the compiler can parallelize the loop
#pragma acc kernels loop independent
{
    for (int i = 0; i < N; i++)
        A[i] = B[i] + C[i];
}
```

Compiling flags

Flag	Description
`-acc`	enable OpenACC targeting device
`-acc=host`	generate an executable that will run serially on the host CPU
`-acc=multicore`	parallelize for a multicore CPU
`-acc=gpu -gpu=cc80`	map OpenACC parallelism to an NVIDIA GPU, compile targ
`-gpu=managed`	place all allocatables in CUDA Unified Memory
`-gpu=pinned`	use CUDA pinned memory for all allocatables
`-Minfo=accall`	compiler diagnostics for OpenACC
`export NVCOMPILER_ACC_NOTIFY=1|2|3`	environment variable for NOTIFY
`-mp`	enable OpenMP targeting device
`-mp=gpu`	to generate an executable that will run serially on the host CP
`-gpu=cc80`	map OpenACC parallelism to an NVIDIA GPU, compile targ
`-gpu=managed`	place all allocatables in CUDA Unified Memory
`-gpu=pinned`	use CUDA pinned memory for all allocatables
`-Minfo=accall`	Compiler diagnostics for OpenACC
`export NVCOMPILER_ACC_NOTIFY = 1|2|3`	Environment variable for NOTIFY

...

7.1 Data Management in OpenACC

Below are examples of how different OpenACC data clauses control the movement and allocation of data between the CPU (host) and GPU (device):

1. Allocating memory on the GPU

```
int *A = (int*) malloc(N * sizeof(int));
#pragma acc parallel loop
{
    for (int i=0; i<N; i++) A[i] = 0;
}
```

In this snippet, memory for array `A` is allocated on the host using `malloc`. The OpenACC pragma without any explicit data clause does not allocate or manage the corresponding device memory. If `A` is not already present on the device, the compiler may implicitly manage this, but it's best to use data directives for clarity and control.

2. `copy(A[0:N])` – Copy data from host to device and back

```
for (int i=0; i<N; i++) A[i] = 0;
#pragma acc parallel loop copy(A[0:N])
{
    for (int i=0; i<N; i++) A[i] = 1;
}
```

The `copy(A[0:N])` clause tells the compiler to allocate space for array `A` on the GPU. Before the kernel runs, the data from the host (CPU) `A` is copied to the device (GPU). When the parallel region ends, the (possibly updated) data is copied back from device to host. This is useful when the GPU kernel is both reading from and writing to `A`, and you need to preserve the changes on the host after execution.

3. `copyin(A[0:N])` and `copyout(B[0:N])` – Directional data transfer

```
#pragma acc parallel loop copyin(A[0:N]) copyout(B[0:N])
{
    for (int i=0; i<N; i++) B[i] = A[i];
}
```

Here, `copyin(A[0:N])` means the data for `A` will be copied from the host to the device before the parallel region begins, but any modifications to `A` on the device are not copied back. `copyout(B[0:N])` means a fresh array `B` is allocated on the device, and after the region completes, the contents are copied from device to host. This is suitable when you need only to read from `A` on the device and produce output in `B`.

Data region constructs

…

Unstructured data

…

7.2 Loop Optimization

Seq clause

If we are in a situation where we have nested loops but we want to parallelize the outer loops, but not the inner one, we can use the `SEQ` clause.

```
#pragma acc parallel loop
for (int i = 0; i < N; i++) {
    #pragma acc loop
    for (int j = 0; j < M; j++) {
        #pragma acc loop seq
        for (int k = 0; k < K; k++) {
            A[i][j][k] = B[i][j][k] * C[i][j][k];
        }
    }
}
```

If we want to parallelize the outer loop, but not the inner one, we can use the `SEQ` clause.

Collapse clause

If you want to combine the next *N* tightly nested loops into a single loop, you can use the `collapse` clause. It is extremely useful for increasing memory locality, as well as creating larger loop to expose more parallelism.

```
#pragma acc parallel loop collapse(2)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        // code
    }
}
```

Tile clause

If you want to parallelize a loop but you want to divide it into smaller chunks, you can use the `tile` clause. It is extremely useful for increasing memory locality, as well as executing multiple tiles simultaneously.

```
#pragma acc parallel loop tile(32,32)
for (int i = 0; i < 128; i++) {
    for (int j = 0; j < 128; j++) {
        // code
    }
}
```

Observation: *Note*

Similar to the `collapse` clause, the inner loops should not have the `loop` directive.

Reduction clause

…

7.3 GPU Hardware Hierarchy

We have seen that the GPU is a very parallel machine, but it is not a single core machine. It's architechture is composed on Gangs, Workers and Vectors.

- **Gang**: Multiple gangs will be generated, and loops iterations will be spread across the gangs. Gangs are independent of each other. There is no way to for the programmer to know exactly how many gangs are running at a given time.
- **Worker**: To have multiple vectors within a gang. It splits up one large vector into multiple smaller vectors. It is an intermediate level between the low-level parallelism implemented in vector and group of threads. It is useful when our inner parallel loops are very small, and will not benefit from having a large vector.
- **Vector**: Lowest level of parallelism. Every gang will have at least 1 vector. Threads work in lockstep (SIMD/SIMT parallelism).

```
1 #pragma acc parallel loop gang
2 for (int i = 0; i < N; i++)
3     #pragma acc loop worker
4     for (int j = 0; j < M; j++)
5         #pragma acc loop vector
6         for (int k = 0; k < K; k++)
7             // code
```

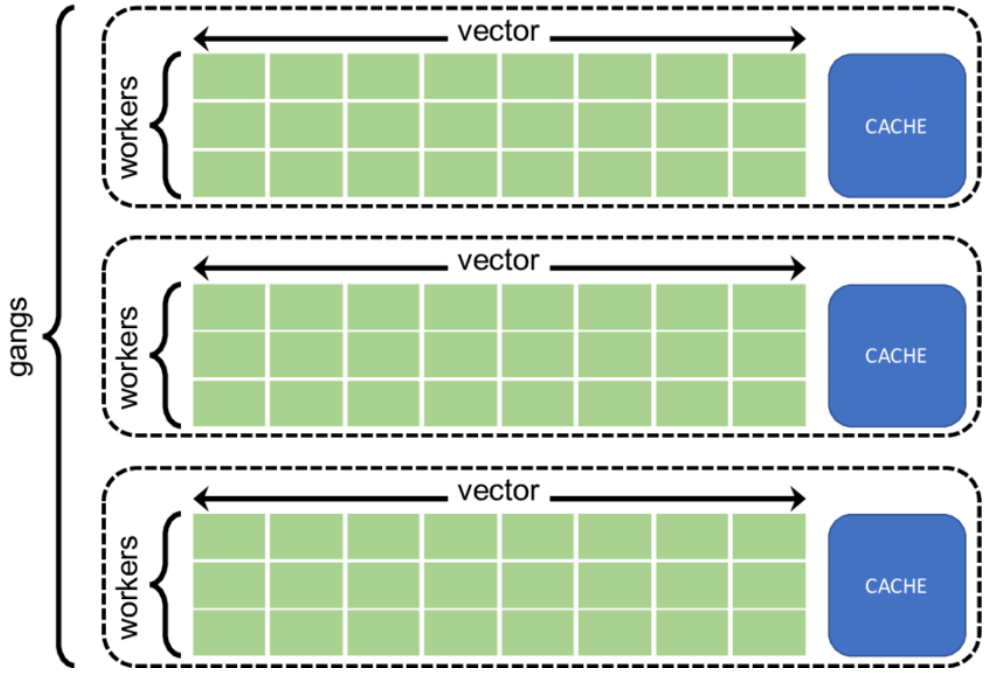


Figure 7.1: Gang, Worker and Vector Hierarchy

Tip: ***Rule of 32***

The rule of thumb to decide the gang, worker and vector sizes when programming for NVIDIA GPUs is to use a vector length that is a multiple of 32.

MISSING: good and bad example of usage

7.4 OMP vs ACC

OpenMP and OpenACC both provide high-level, directive-based approaches for parallel programming, allowing the developer to easily control how loops and computations are parallelized through the use of clauses. These models enable the compiler to map code onto multiple levels of parallelism, such as teams of threads or gangs, workers, and vectors, across different hardware architectures.

OpenACC	CUDA	Mapping to NVIDIA GPU	OpenMP
Parallel/Kernel	Kernel	GPU	Parallel
Gang	Thread block	SMs	Team
Worker	Thread	SP or Compute unit	Thread
Vector	Warp (32 threads)	32-wide thread	SIMD

Table 7.1: Comparison of Parallel Hierarchy: OpenACC, CUDA, NVIDIA GPU, and OpenMP

In theory (according to the standards) the implementation of the levels adapt to the hardware, but in reality some compilers struggle with certain parallelisation levels:

OpenACC	OpenMP
`acc parallel`	`omp target teams`
`acc loop gang`	`omp distribute`
`acc loop worker`	`omp parallel loop`
`acc loop vector`	`omp simd`
`acc declare`	`omp declare target`
`acc data`	`omp target data`
`acc update`	`omp target update`
`copy/copy_in/copy_out`	`map(to/from/tofrom: ... )`

Table 7.2: OpenACC to OpenMP Directive and Clause Correspondence

OMP Parallel

- Creates a team of threads
- Very well-defined how the number of threads is chosen
- May synchronize within the team
- Data races are the user's responsibility

...

ACC Parallel

- Creates 1 or more gangs to workers
- Compiler free to choose number of gangs, workers and vector lengths
- May not synchronize between gangs
- Data races are not allowed

8 MPI and GPU offloading using Python

8.1 MPI in Python

Message Passing Interface (MPI) is a standard for disrtribuited-memory parallel computing, using to enable communication between multiple processes. MPI is useful in HPC applications, where we need to distribuite large computations across multiple nodes.

MPI in python is implemented using the mpi4py library. To install it, we can use the following command:

```
pip install mpi4py
```

how to use:

```
mpirun -n <number_of_processes> python <script.py>
```

Communications

- **Point-to-point communications**

```
if rank == 0:
    data = "Hellp, Process 1"
    comm.send(data, dest=1)
elif rank == 1:
    data = comm.recv(source=0)
    print(f"Process 1 received: {data}")
```

- **Non-blocking communications**

```
if rank == 0:
    data = "Hellp, Process 1"
    comm.isend(data, dest=1, tag=11)
elif rank == 1:
    data = comm.irecv(source=0, tag=11)
    print(f"Process 1 received: {data}")
```

- **Collective communications**

```
comm.bcast(data, root=0)
```

Example: *Example*

We want to implement a code that computes the distance between each point of a dataset and of another one, and then compute the distance distribution.

```
import numpy as np
import mpi4py
from mpi4py import MPI

npart_data = int(sys.argv[1])
Nbins = 20

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    Data1 = np.random.rand(npart_data, 2)
    Data2 = np.random.rand(npart_data, 2)
    bins = np.linspace(0, 1.5, Nbins)
else:
    Data1 = None
    Data2 = np.empty((npart_data, 2))
    bins = np.empty(Nbins)

comm.Barrier()

comm.Bcast(Data1, root=0)
comm.Bcast(Data2, root=0)

Data1_on_task = scatter_data(Data1)

dist_hist = get_statistical_distance([Data1_on_task, Data2, bins])

dist_hist_fin = none
if rank == 0:
    dist_hist_fin = np.empty([size, len(dist_hist)])

comm.Gatherv(dist_hist, dist_hist_fin, root=0)

if rank == 0:
    dist_hist_fin = np.sum(dist_hist_fin, axis=0)
```

where `scatter_data` is a function that scatters the data to the different tasks.

```python
def scatter_data(Data):
    rows_per_rank = np.full(size, npart_data // size)
    rows_per_rank[:npart_data % size] += 1

    # Allocate memory for the data on the current task
    Data1_on_task = np.empty((rows_per_rank[rank], 2))

    # Calculate the displacement
    displacement = np.cumsum(np.insert(rows_per_rank[:-1], 0, 0)) *
        2

    # Scatter the data
    comm.Scatterv([Data1, rows_per_rank * 2, displacement, MPI.
       DOUBLE], Data1_on_task, root=0)

    return Data1_on_task
```

8.2 GPU offloading in Python

It is possible to offload the computation to the GPU using different libraries.

8.2.1 Low level APIs: PyCUDA, PyOpenCL

PyCUDA is a library that allows to write CUDA code in Python. It provides a low level API for CUDA programming.

```
...
```

8.2.2 Mid level APIs: Numba

Numba is a just-in-time compiler for Python that allows to write CUDA code in Python. It provides a mid level API for CUDA programming.

```
...
```

8.2.3 High level APIs: CuPy

CuPy is an open-source array library for GPU-accelerated computing in Python on NVIDIA CUDA or AMD ROCm platforms.

It is highly comparible with NumPy and SciPy; in most cases it can be used as a drop-in replacement.

At the same time it allows programmers to write custom CUDA kernels.

```python
import cupy as cp

def distances_GPU_cupy(Data1, Data2, bins):
    Data1 = cp.asarray(Data1)
    Data2 = cp.asarray(Data2)
    bins = cp.asarray(bins)

    dist_hist = get_distances_GPU_cupy(Data1, Data2, bins)
```

```
    return dist_hist

def get_distances_GPU_cupy(Data1, Data2, bins):
    # calculate the distances in a vectorized manner
    dx = Data1[:, 0][:, cp.newaxis] - Data2[:, 0][cp.newaxis, :]
    dy = Data1[:, 1][:, cp.newaxis] - Data2[:, 1][cp.newaxis, :]
    dist = cp.sqrt(dx**2 + dy**2)

    # use cp.histogram for GPU histogramming
    dist_hist = cp.histogram(dist, bins=bins)
    return dist_hist
```

Extensions

- cuDF is a GPU DataFrame library for managing data, that extends the capabilities of Pandas to the GPU.
- Dpnp is a library used by Intel that implements a subset of the NumPy API that can be executes on any data parallel device.

Warning: *Offloading only when needed*

Offloading only when needed. If the data is small, it is not worth offloading it to the GPU, and it can take more time than the computation on the CPU.

Bibliography

[1] *Advanced-High-Performance-Computing-2024*. https://github.com/Foundations-of-HPC/Advanced-High-Performance-Computing-2024.

[2] CINECA. *Leonardo HPC System | Leonardo Pre-exascale Supercomputer — leonardo-supercomputer.cineca.eu*. https://leonardo-supercomputer.cineca.eu/hpc-system/.

[3] *Cornell Virtual Workshop > Vectorization > Vector Hardware > Registers — cvw.cac.cornell.edu*. https://cvw.cac.cornell.edu/vector/hardware/registers.

[4] Nitin Shukla Michael Redenti. *SCAI Training / Trieste GPU Computing School - Nov25 · GitLab — gitlab.hpc.cineca.it*. https://gitlab.hpc.cineca.it/training/trieste-gpu-computing/. 2025.