
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**

November 19, 2025



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**
- 1.1 Basic Concepts . . . 1
- 1.2 Checking for SIMD capabilities . . . 4
- 1.3 Vector Types . . . 5
- 1.4 Loops auto-vectorization . . . 6
  - 1.4.1 Data dependencies . . . 9

**2 Vector Types** **12**
- 2.1 Memory Alignment . . . 12
  - 2.1.1 Conditional evaluation . . . 14
  - 2.1.2 Vectorization by OpenMP SIMD directive . . . 15
  - 2.1.3 The omp SIMD functions . . . 15

**3 OpenMP** **17**
- 3.1 Tasks . . . 17

**4 Supercomputers** **21**
- 4.1 Leonardo Supercomputer . . . 21
  - 4.1.1 System Architecture and Modules . . . 21
  - 4.1.2 Storage System . . . 22
  - 4.1.3 Accessing Leonardo . . . 23
- 4.2 Working on Leonardo . . . 23
  - 4.2.1 Filesystem Organization . . . 23
  - 4.2.2 Software and Module Environment . . . 24
  - 4.2.3 Job Submission with Slurm . . . 24

**5 CUDA C/C++** **27**
- 5.1 Introduction to Accelerated Computing . . . 27
  - 5.1.1 Architectural Design Philosophies: Latency vs. Throughput . . . 28
- 5.2 The GPU Architecture . . . 29
  - 5.2.1 NVIDIA Ampere Architecture . . . 29
- 5.3 CUDA Programming Model . . . 31
- 5.4 GPU Thread Hierarchy . . . 32
- 5.5 NVIDIA profiling tools . . . 34
  - 5.5.1 CUDA Streams . . . 35

**6 OpenACC** **37**
- 6.1 Data Management in OpenACC . . . 39
- 6.2 Loop Optimization . . . 40
- 6.3 GPU Hardware Hierarchy . . . 41
- 6.4 OMP vs ACC . . . 42

# 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.

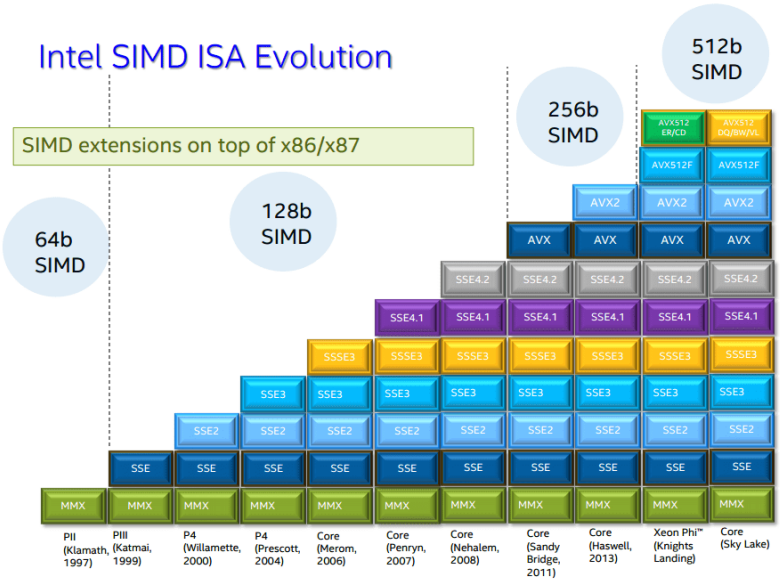



**Figure 1.1:** Intel SIMD ISA Evolution: timeline of vector instruction set extensions introduced with each CPU generation. The diagram shows how SIMD register width increased from MMX and SSE (64-128 bits) to AVX (256 bits) and AVX-512 (512 bits), with each new standard supporting more data parallelism.

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.

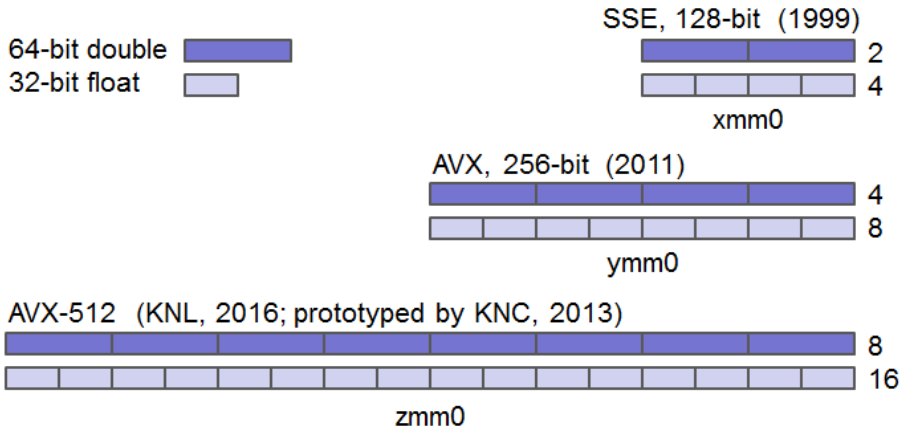



**Figure 1.2:** SIMD registers: each generation widens the registers, allowing more doubles (dark) or floats (light) to be processed in parallel. [3]

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.

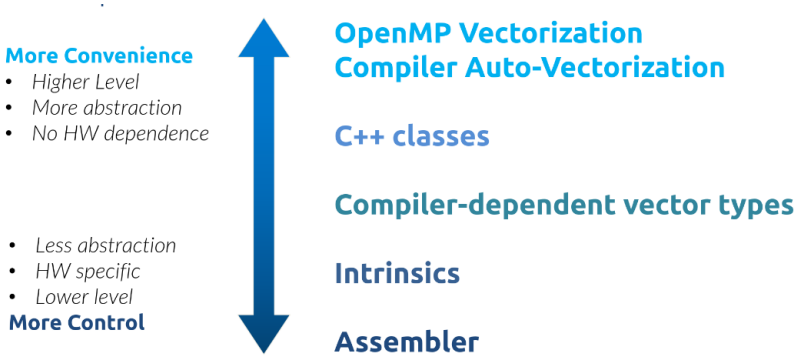



**Figure 1.3:** Approaches to vectorize, from high-level convenience to low-level control. [1]

### 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)<br>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

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.

```
1  // Select vector types based on the available ISA
2  #ifdef __AVX512__
3  typedef __m512d dvector_t;
4  typedef __m512  fvector_t;
5  typedef __m512i ivector_t;
6
7  #elif defined ( __AVX__ ) || defined ( __AVX2__ )
8  typedef __m256d dvector_t;
9  typedef __m256  fvector_t;
10 typedef __m256i ivector_t;
11
12 #elif defined ( __SSE4__ ) || defined ( __SSE3__ )
13 typedef __m128d dvector_t;
14 typedef __m128  fvector_t;
15 typedef __m128i ivector_t;
16
17 #else
18 typedef double dvector_t;
19 typedef float  fvector_t;
20 typedef int    ivector_t;
21 #endif
22
23 // Convenience macros for sizes (elements per vector and bit width)
24 #define DV_ELEMENT_SIZE ( sizeof(dvector_t) / sizeof(double) )
25 #define DV_BIT_SIZE     ( sizeof(dvector_t) * 8 )
26 #define FV_ELEMENT_SIZE ( sizeof(fvector_t) / sizeof(float) )
27 #define FV_BIT_SIZE     ( sizeof(fvector_t) * 8 )
28 #define IV_ELEMENT_SIZE ( sizeof(ivector_t) / sizeof(int) )
29 #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.

**MISSING:** auto-vectorization example (slide 47-50)

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.

**MISSING:** profileing table, loop unrolling -> clear vectorization (slides 52-71)

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:

```
int i = 0;
while (i < N) {
    a[i] = b[i] * c[i];
    if (a[i] > 0) {
        break;
    }
    i++;
}
```

**Tip**: ***Countable While Loops***

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

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` ).

**MISSING:** non contiguous memory access (slides 77-78)

### 1.4.1 Data dependencies

**TODO:** enhance this section

$S_1, S_2 \rightarrow M$

| | | |
|---|---|---|
| $S_1$ W | $S_2$ R | RAW \| TRUE \| FLOW |
| $S_1$ R | $S_2$ W | WAR \| ANTI-DEP |
| $S_1$ W | $S_2$ W | WAW \| OUT-DEP |

where:

- RAW: Read After Write
- WAR: Write After Read
- WAW: Write After Write

```
1 $S_1$: ... = A[i+2];
2 $S_2$: A[i+2] = ...;
3 $S_3$: ... = A[i];
```

$S_1 \delta^a S_2$ FWD loop-indep $S_2 \delta^t S_3$ FWD loop-dep

An of false dependency is:

```
1 $S_1$: A[i] = ;
2 $S_2$: ... = A[i+1];
```

This is a FWD loop-dep dependency. And it is a false dependency: if we simply invert the order of the statements, the code will be semantically correct and completely vectorizable.

**Loop independent**: there can be a dependency but inside the same iteration of the loop

#### Non-contiguous memory access

A contiguous memory access results in fetching and using of entire lines of cache; the bandwidth utilization is optimal.

strided memory access results in gathering sparse memory location, accessed with separate memory calls, with a lot of overhead in memory calls and in mem moves from/to registers.

Also in this case, the ***Read-After-Write (flow-dependency)*** are the main issue.

A variable is written in an interation and read on a subsequent one.

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

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.

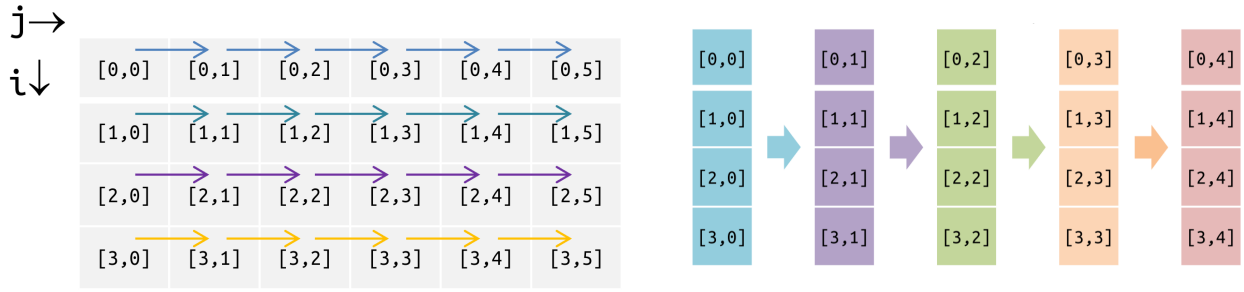



**Figure 1.4:** RAW data dependency: 4 (or 8) outer iterations can be executed together because there are no vertical dependencies (outer loop vectorization)

**MISSING:** notes about aligned memory

### AoS over SoA

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:

```
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):

...

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

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

Data rearranged in SoA approach definitely would expose more, and more efficient, vectorization opportunities.

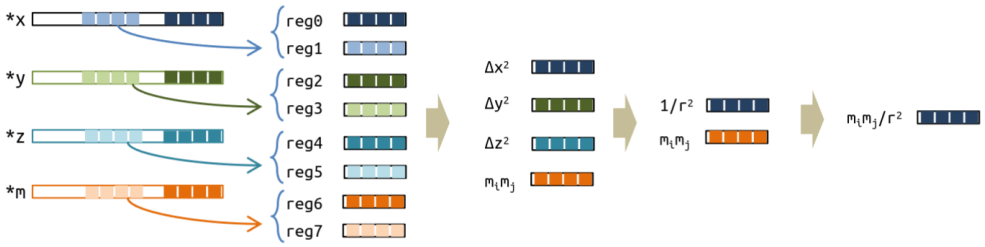



**Figure 1.5:** AoS vs SoA

**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.

# 2 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.

```
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.

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

## 2.1 Memory Alignment

It is mandatory that memory regions accessed by vector instructions are aligned.

…

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);
```

**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.

```
1  double kernel( double *restrict A, double *restrict B, double *
       restrict C, int N ) {
2      double sum = 0;
3      for ( int i = 0; i < N; i++ )
4          sum += A[i]*B[i] + C[i];
5      return sum;
6  }
```

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;
```

### 2.1.1 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.

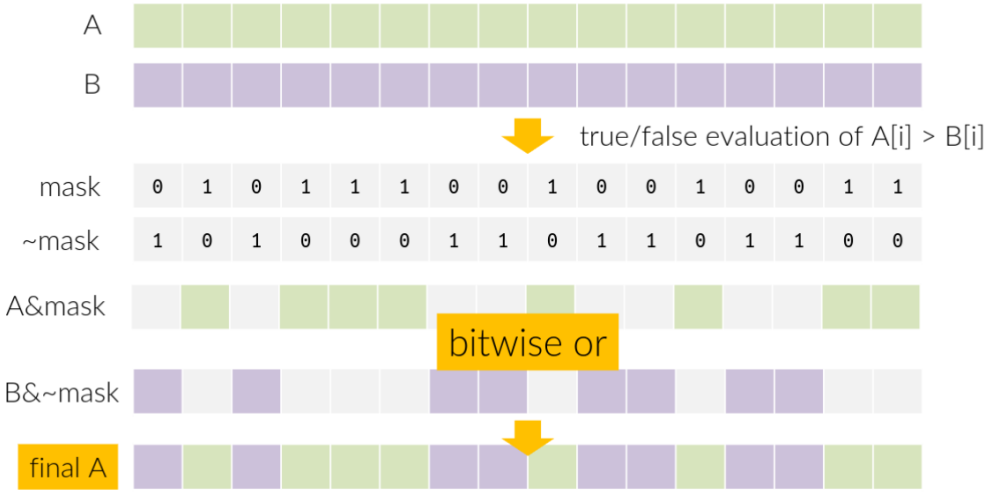



**Figure 2.1:** Conditional evaluation [1]

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.

```
dvector_t *vA = (dvector_t *)__builtin_assume_aligned(A, VALIGN);
dvector_t *vB = (dvector_t *)__builtin_assume_aligned(B, VALIGN);

int VN = (N/DVSIZE)&0xFFFFFFFC;
IVDEP
LOOP_VECTORIZE
LOOP_UNROLL_N(4)
for ( int i = 0; i < VN; i++ ) {
    dvector_t a = vA[i];
    dvector_t b = vB[i];
    llvector_t keep = (vA[i]>=vB[i]);
    vA[i] = (dvector_t)(((llvector_t)a & keep) |
            ((llvector_t)b & ~keep));
    vB[i] = (dvector_t)(((llvector_t)b & keep) |
            ((llvector_t)a & ~keep));
}

int j = VN*DVSIZE;
process ( &A[j], &B[j], N-j+1);
```

### 2.1.2 Vectorization by OpenMP SIMD directive

```
#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.
- `reduction(identifier: list)`;
- `collapse(n)`;
- `simdlen(length)`;
- `safelen(length)`: It is used to specify at the compiler the maximum length not having dependences inside the loop;

```
#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 ])`;

### 2.1.3 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.

penMP 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.

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.

# 3 OpenMP

## 3.1 Tasks

A **task** in OpenMP represents an independent unit of work, comprising a well-defined set of instructions or operations that can be executed asynchronously. Tasks enable the efficient decomposition of complex computations into smaller, manageable pieces that can be scheduled and executed in parallel by different threads. This **task abstraction** allows for flexible parallelism beyond the constraints of loop-based parallel regions, supporting dynamic workloads with unpredictable execution paths.

**Data Abstraction** refers to the encapsulation of logically uniform pieces of information (such as arrays, objects, or 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. These dependencies form a **task dependency graph**, where nodes are tasks and directed edges represent data dependencies (i.e., one task must complete before another can start due to shared data). It is crucial that this graph is **acyclic** ensuring there are no circular wait conditions or deadlocks, and allowing the runtime system to correctly schedule and execute tasks respecting all required data dependencies.

It is sometimes possible to parallelize a workflow that is irregular or depends on runtime conditions using OpenMP ***sections***. However, solutions built with sections often become convoluted and difficult to manage, and, more importantly, the intrinsic rigidity of the sections construct makes it nearly impossible to express truly dynamic patterns of parallelism.

Since version 3.0, OpenMP introduced the **task** construct, which provides an elegant and flexible way to handle precisely these kinds of problems—where execution flow is irregular and only determined at runtime. 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, efficiently balancing the workload and enabling dynamic parallelism.

We have a main thread which generates tasks and a pool of threads which execute them. As soon as they are free, the free threads pickup one queued task each, and execute it. In the same way, when the main thread finishes to create tasks, it picks up one of the queued tasks and executes it.

To make it clearer: 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, of course, there is nested parallelism involved) and mapped onto the underlying physical cores.

....

The management of tasks in OpenMP involves three main key concepts: the data environment, creation and execution, and synchronization and dependence. The **data environment** refers to how data is assigned to each task, and how the framework distinguishes between shared and private variables within task contexts. **Creation and execution** concerns when tasks are generated, how many tasks are created, who is responsible for their execution, and whether there is any prioritization among them. Finally, **synchronization and dependence** focuses on the ways tasks are coordinated, including how tasks are synchronized, scheduled, and whether they depend on the results or completion of other tasks. A robust understanding and careful management of these aspects is essential for writing correct and efficient parallel code using OpenMP's tasking model.

```
1  int main( int argc, char **argv )
2  {
3      #pragma omp parallel
4      {
5      #pragma omp single
6      {
7          printf( " »Yuk yuk, here is thread %d from "
8                  "within the single region\n", omp_get_thread_num() );
9          #pragma omp task
10         {
11         printf( "\tHi, here is thread %d "
12                 "running task A\n", omp_get_thread_num() );
13         }
14         #pragma omp task
15         {
16         printf( "\tHi, here is thread %d "
17                 "running task B\n", omp_get_thread_num() );
18         }
19     }
20     printf(" :Hi, here is thread %d after the end "
21             "of the single region, I was stuck waiting "
22             "all the others\n", omp_get_thread_num() );
23     }
24     return 0;
25 }
```

### Scheduler points

- **barrier**: either implicit or explicit.
  all tasks reated by any thread in the current parallel region are guaranteed to complete after the barrier exits.
- **taskwait**:
  all children tasks are completed, the encountering task is suspended until that is true it does not apply to descendants: i.e. it includes only direct children tasks.
- **taskgroup**:
  All descendant tasks are guaranteed to be completed at the exit of the taskgroup region (see later); it baheves as an implicit omp barrier.

### Scope of variables

in the code above, we called `omp_get_thread_num()` multiple times because it is a thread-local variable. If we had declared it as a global variable, it would have been shared by all threads and the output would have been the same for all threads (the one who enters the single region).

The main point to consider here is that by its very nature, the tasks' creation is driven by the coeval data context and is not related to the values that any variable will have in the future at the moment of their execution.

As such, the rule-of-thumb is **data, unless otherwise stated, are copied in local copies so that to preserve the data context at the moment of creation**.

```
...
#pragma omp single
{
    int me = omp_get_thread_num();
    printf( " »Yuk yuk, here is thread %d from "
            "within the single region\n", me );
    #pragma omp task
    {
    printf( "\tHi, here is thread %d "
            "running task A\n", me );
    }
    #pragma omp task
    {
    printf( "\tHi, here is thread %d "
            "running task B\n", me );
    }
}
...
```

here `me` will have always the same value, also if different threads will execute the tasks. This is a **very common error to avoid**.

Let's remind the following rules:

- When a variable is shared on the task creation the storage used is that referred with that name at the point where the task was created
- When a variable is private on the task creation the references to it (inside the task code region) use the uninitialized storage that is created when the task is executed
- When a variable is firstprivate on a task creation the references to it inside the task code region are to the new storage that is created and initialized with the value of the existing storage of that name when the task is created

…

With the `untied` clause, you are signalling that this task, if ever suspended, can be resumed by any free thread. The default is the opposite, a task to be tied to the thread that initially starts it.

If untied, you must take care of the data environment, for instance, no `threadprivate` variables can be used, nor the thread number, and so on.

---

locks

```
    // set the lock, blocking
    omp_set_lock(&lock);
    // set the lock, non-blocking
    omp_test_lock(&lock);

```

```
6       // unset the lock
7       omp_unset_lock(&lock);
```

# 4 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.

## 4.1 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.

### 4.1.1 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.

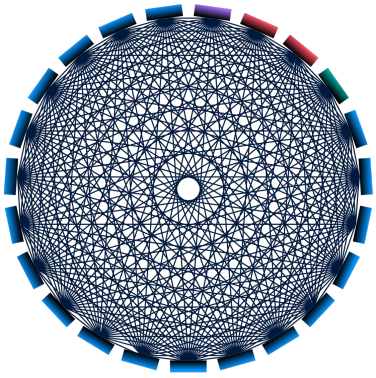


**Figure 4.1:** Dragonfly+ Topology. Booster Module nodes (blue), I/O cell (purple), Data-centric cells (red), and Hybrid cell (green) [2].

### 4.1.2 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).

### 4.1.3 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

## 4.2 Working on Leonardo

### 4.2.1 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.

### 4.2.2 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.

### 4.2.3 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.

# 5 CUDA C/C++

## 5.1 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.

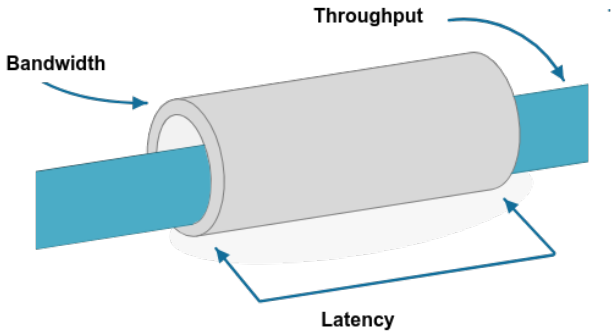
This chapter introduces *accelerated computing*, a paradigm where a specialized, highly parallel processing unit, the Graphics Processing Unit (GPU), is used in conjunction with a traditional Central Processing Unit (CPU) to accelerate computationally intensive portions of an application. We will explore 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.

### 5.1.1 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.

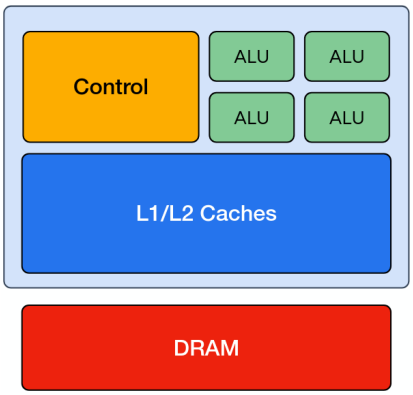



**Figure 5.1:** Simplified CPU [4]

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 5.2:** 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.

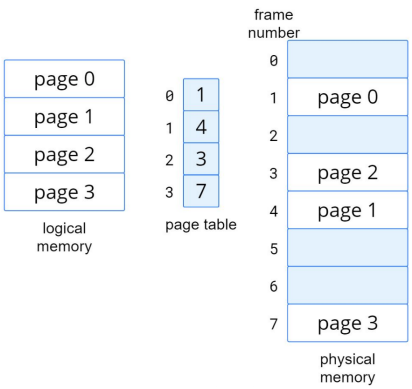



**Figure 5.3:** Translation of virtual to physical addresses.

**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.

**MISSING:** multicore CPUs

#### 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.

## 5.2 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.

### 5.2.1 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.

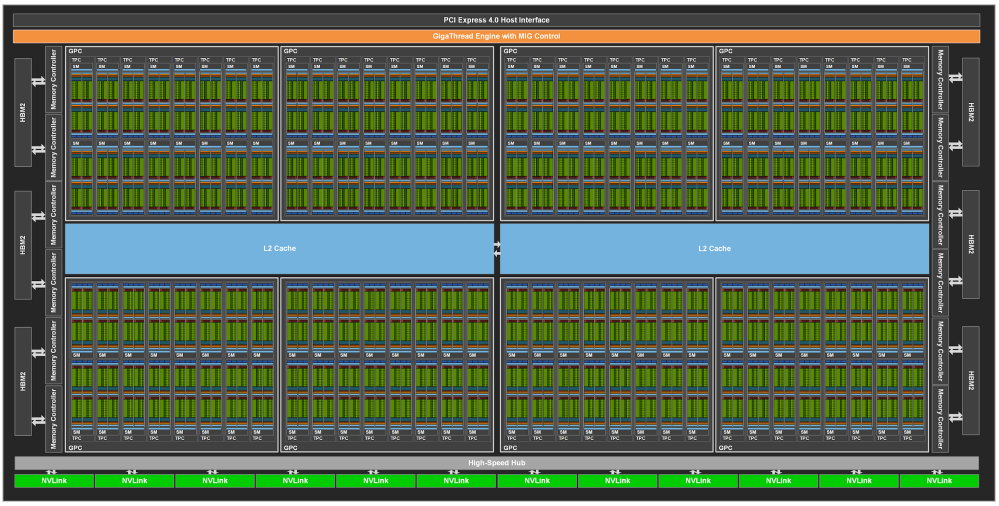



**Figure 5.4:** High-level overview of the NVIDIA Ampere A100 GPU architecture, showing the arrangement of GPCs and SMs.

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 5.5:** Simplified block diagram of an NVIDIA Ampere Streaming Multiprocessor (SM).

## 5.3 CUDA Programming Model

`<<<>>>`: are used to quantify the number blocks and threads to launch the kernel.

We have dirrerent qualifiers for the functions:

- `__global__`: It indicates that the function will run in the GPU, and can be invoked generally both from the host and the device.

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

int main() {
    CPUFunction();
    GPUFunction<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

- `__device__`: device function, executed on the device.
- `__host__`: host function, executed on the host.
- `__host__ __device__`: host and device function, compiled for both.

A cuda kernel is executed as a grid (array) of threads, each of them runs in the same kernel.

Each thread has a unique index `threadIdx.x`, which is used to access the data.

To compile a program we must use the NVIDIA `nvcc` compiler. It is able to identify the functions to be executed on the GPU and the ones to be executed on the host.

```
nvcc -arch=sm_80 -o <output-file> <cuda-code>.cu -run
```

Where:

- `-arch=sm_80`: specifies the architecture of the GPU to compile for.
- `-run`: runs the program after compilation.

### SIMT vs SIMD

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

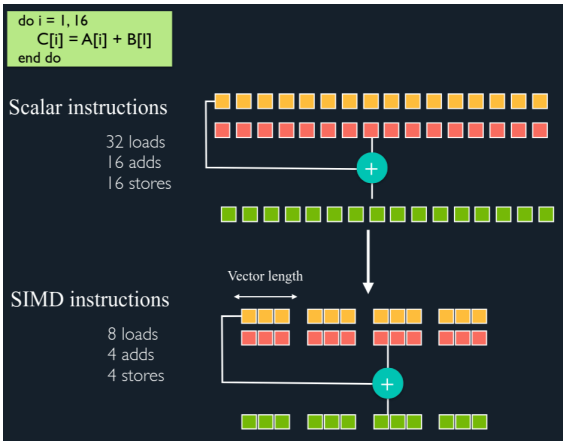



**Figure 5.6:** SIMT vs SIMD

- **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.

## 5.4 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.

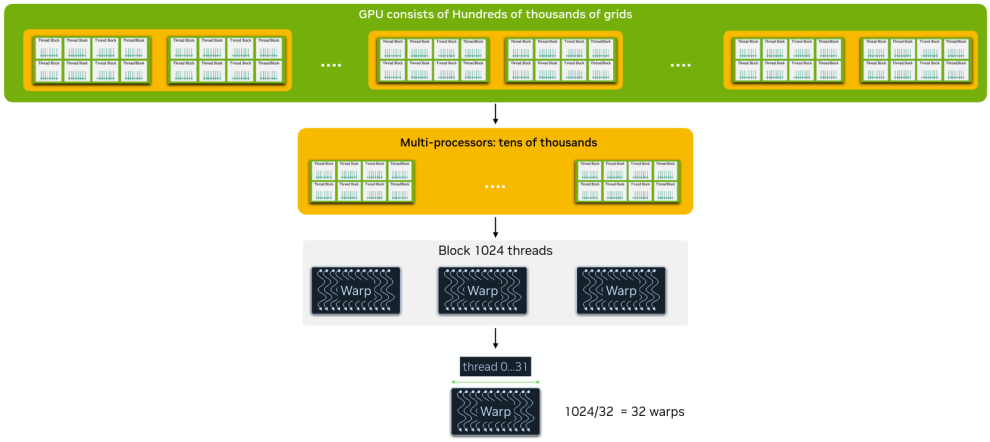



**Figure 5.7:** GPU Thread Hierarchy

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 To increase parallelism, we need to coordinate work among thread blocks

This is achieved by mapping element of data vector to threads using global index

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

### Error handling

All CUDA APIs returns an error code of type `cudaError_t`. A special value `cudaSuccess` is returned on success, otherwise an error code is returned.

To understand wich problem occurred, we can use the cudaGetErrorString() function.

```
    cudaError_t err;
    err = cudaMallocManaged(&a, size);
    if (err != cudaSuccess)
        printf("Error: %s\n", cudaGetErrorString(err));
```

To check for errors occurring at the time of kernel execution, we can use the cudaGetLastError() function.

## 5.5 NVIDIA profiling tools

A typical scenario while accelerating a code is to start with a cpu-only version and then accelerate it using cuda.

TO understand if everything is working correctly, we start with a **Nsight System profiler**: it is a system-wide performance viewer, which tracks CPU and GPU interactions across time.

If we find something wrong, we can use the **Nsight Compute profiler**: it is a GPU-only profiler, which tracks the occupancy, memory usage, and instruction throughput.

Let's see how to use the Nsight System profiler.

```
nsys profile -t cuda,nvtx,mpi,openacc -stats=true --force-overwrite=true
    -o <output-file> <executable>
```

### Nsight Compute profiler

```
nsys profile -t cuda -o <output-file> <executable>
```

—

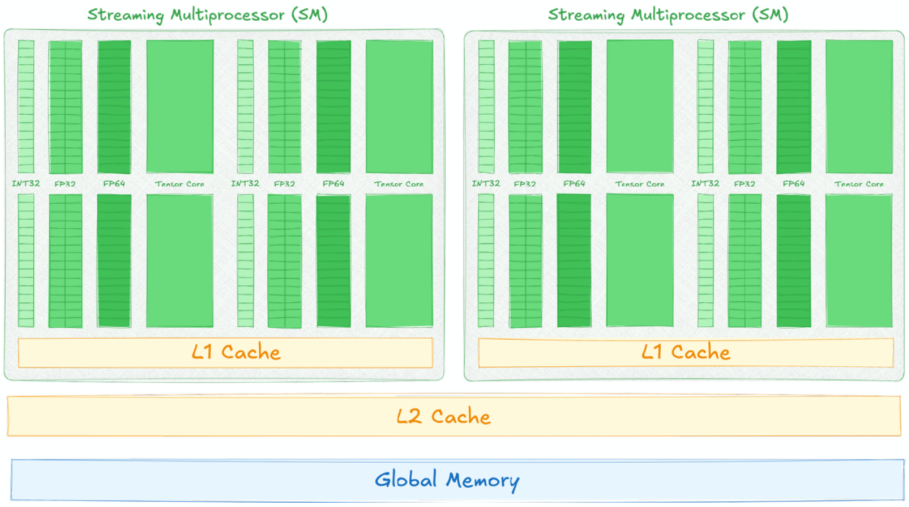



**Figure 5.8:** Memory Hierarchy

| Bandwidth | Latency |
|---|---|
| 13× | 1× |
| 3× | 5× |
| 1× | 15× |

**Figure 5.9:** Bandwidth and Latency comparison

In such architecture, CUDA provides 4 different memory types:

- **Global Memory**: The largest memory space available on the device, but also the slowest in terms of access latency.
  - Scope: Accessible by all threads (from any block or grid).
  - Usage: Memory allocated with `cudaMalloc()` and data transferred using `cudaMemcpy()`.
- **Shared Memory**: Fast and low-latency memory, but much smaller than global memory.
  - Scope: Shared between all threads within the same block.
  - Usage: Useful for data reuse and to reduce global memory accesses. Declared with the `__shared__` qualifier.
- **Registers**: The fastest memory available, mapped directly to the hardware registers of each Streaming Multiprocessor (SM).
  - Scope: Private to each thread; cannot be accessed by other threads.
  - Usage: Used for automatic variables declared within a kernel.
- **Constant Memory**: Read-only memory space cached and optimized for uniform access by all threads. Offers efficient broadcast for all threads reading the same location.
  - Scope: Accessible from all threads, but cannot be modified from device code.
  - Usage: Declared with the `__constant__` qualifier.

**Tip**: ***Memory coalescing***

For optimal performance, threads in a warp should access consecutive memory addresses. Coalesced memory access can improve bandwidth utilization by up to 10x compared to non-coalesced access patterns.

### 5.5.1 CUDA Streams

A CUDA stream is a sequence of CUDA operations. The defaualt stream executes those operations in the order they are issued, therefore, an instruction must be completed before the next one can begin.

Multiple streams or Non-default streams can be created and utilise by CUDA programmers. While single stream kernels must execute in order, kernels in different, non-default streams, can interact

concurrently, and therefore have no fixed order of execution

This can be useful to overlap the execution of kernels and memory transfers, or to execute kernels in parallel.

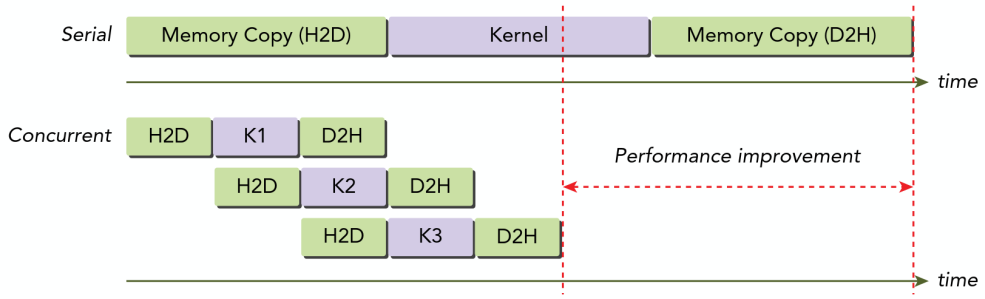



**Figure 5.10:** CUDA Streams Overlap

```
1 // Create a new stream
2 cudaStream_t stream;
3 cudaStreamCreate(&stream);
4
5 // Launch a kernel in the new stream
6 someKernel<<<numberOfBlocks, numberOfThreads, 0, stream>>>();
7
8 // Destroy the stream
9 cudaStreamDestroy(stream);
```

**Advanced Concept**: ***non-blocking streams***

cudaStreamCreate creates a blocking stream, but it exitsts also a non-blocking versions, like `cudaStreamNonBlocking`. This topic is not covered in this course.

# 6 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 |

...

## 6.1 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**

…

## 6.2 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

…

## 6.3 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
```

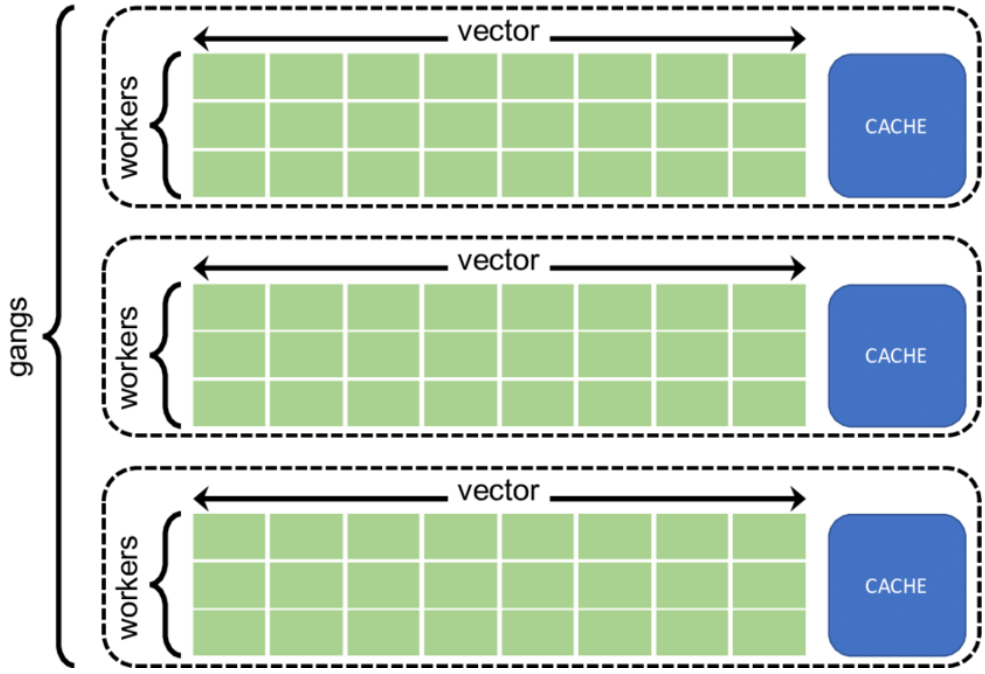



**Figure 6.1:** Gang, Worker and Vector Hierarchy

**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

## 6.4 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 6.1:** 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 6.2:** 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

# 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.