



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

December 1, 2025

This document is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike \(CC BY-NC-SA\)](#) license. You may share and adapt this material, provided you give appropriate credit, do not use it for commercial purposes, and distribute your contributions under the same license.

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 <code>omp SIMD</code> functions	15
3 OpenMP	17
3.1 Tasks	17
3.1.1 Memory consistency model	20
3.1.2 Controlling the task creation - Clauses	22
3.1.3 Task dependencies	23
4 Supercomputers	26
4.1 Leonardo Supercomputer	26
4.1.1 System Architecture and Modules	26
4.1.2 Storage System	27
4.1.3 Accessing Leonardo	28
4.2 Working on Leonardo	28
4.2.1 Filesystem Organization	28
4.2.2 Software and Module Environment	29
4.2.3 Job Submission with Slurm	29
5 CUDA C/C++	32
5.1 Introduction to Accelerated Computing	32
5.1.1 Architectural Design Philosophies: Latency vs. Throughput	33
5.2 The GPU Architecture	34
5.2.1 NVIDIA Ampere Architecture	34
5.3 CUDA Programming Model	36
5.4 GPU Thread Hierarchy	37
5.5 NVIDIA profiling tools	39
5.5.1 CUDA Streams	40
6 OpenACC	42
6.1 Data Management in OpenACC	44

6.2	Loop Optimization	45
6.3	GPU Hardware Hierarchy	46
6.4	OMP vs ACC	47
7	MPI and GPU offloading using Python	48
7.1	MPI in Python	48
7.2	GPU offloading in Python	50
7.2.1	Low level APIs: PyCUDA, PyOpenCL	50
7.2.2	Mid level APIs: Numba	50
7.2.3	High level APIs: CuPy	50
8	Message Passing Interface - MPI	52
8.1	Remote Memory Access - RMA	52
8.1.1	Memory Windows	52
8.2	Shared Memory	53
8.2.1	Getting Shared Memory Pools	53
8.3	Cartesian Communication	55
8.3.1	Core API Functions	56

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.

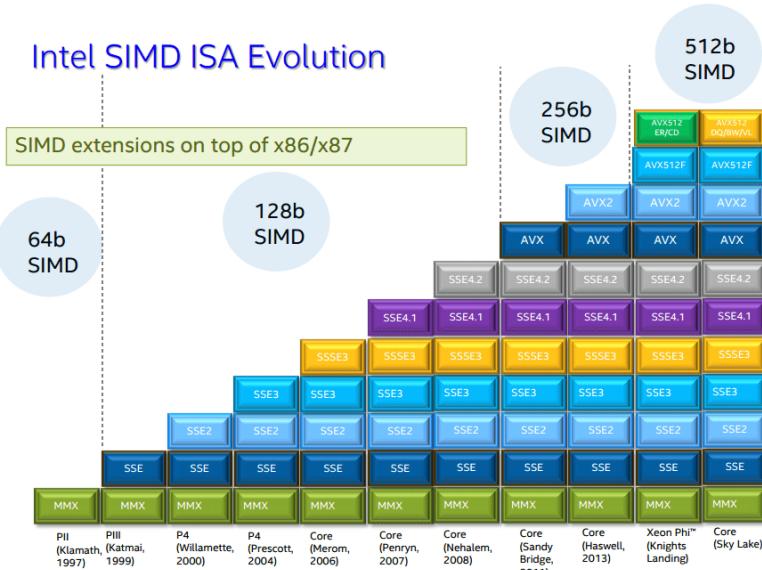


Figure 1.1: Intel SIMD ISA Evolution: timeline of vector instruction set extensions introduced with each CPU generation. The diagram shows how SIMD register width increased from MMX and SSE (64-128 bits) to AVX (256 bits) and AVX-512 (512 bits), with each new standard supporting more data parallelism.

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.

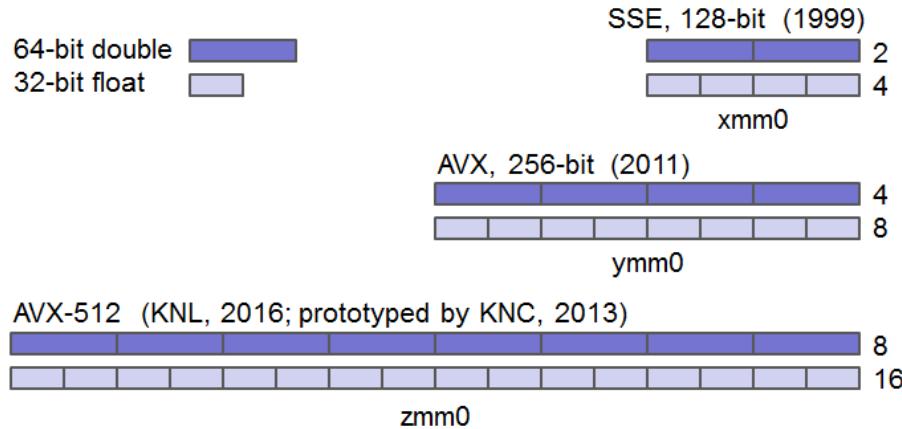


Figure 1.2:
SIMD registers: each generation widens the registers, allowing more doubles (dark) or floats (light) to be processed in parallel.
[3]

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.

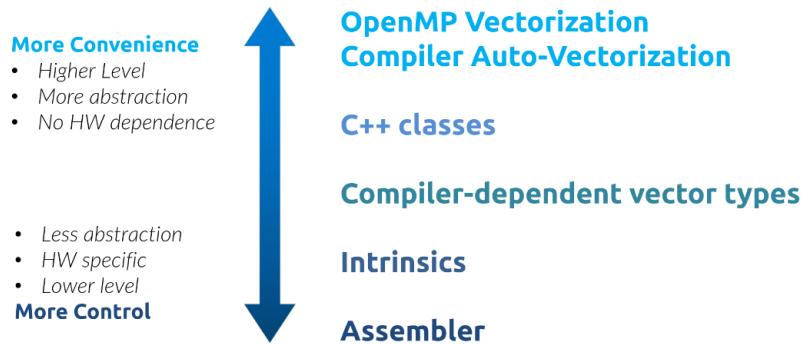
How to achieve vectorization?

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`, `-floop-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) 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 <code>v0</code>)
Configuration	Implicit (by the called instruction)	Implicit (type chosen; length is loop-invariant)	Explicit via <code>vsetvl</code>
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.

```
1 #include <cpuid.h>
2
3 int main() {
4     __builtin_cpu_init();
5
6     if (__builtin_cpu_supports("avx512f"))
7         ...
8     if (__builtin_cpu_supports("avx2"))
9         ...
10    if (__builtin_cpu_supports("avx"))
11        ...
12    if (__builtin_cpu_supports("sse4.2"))
13        ...
14    if (__builtin_cpu_supports("sse4.1"))
15        ...
16    if (__builtin_cpu_supports("sse3"))
17        ...
```

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

```
1 #include <immintrin.h>
2
3 #ifdef __AVX512__
4 #define V_DSIZE (sizeof( __m512d ) / sizeof(double) )
5
6 #elif defined ( __AVX__ ) || defined ( __AVX2__ )
7 #define V_DSIZE (sizeof( __m256d ) / sizeof(double) )
8
9 #elif defined ( __SSE4__ ) || defined ( __SSE3__ )
10 #define V_DSIZE (sizeof( __m128d ) / sizeof(double) )
11
12 #else
13 #define V_DSIZE 1UL
14 #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 intrinsics 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.

FLOATING-POINT types

INTEGER types

types name	int 8bits	int 16bits	int 32bits	int 64bits
<code>__m64</code>	8x	4x	2x	1x
<code>__m128i</code>	16x	8x	4x	2x
<code>__m256i</code>	32x	16x	8x	4x
<code>__m512i</code>	64x	32x	16x	8x

types name	single precision	double precision
<code>__m128</code>	4x	—
<code>__m128d</code>	—	2x
<code>__m256</code>	8x	—
<code>__m256d</code>	—	4x
<code>__m512</code>	16x	—
<code>__m512d</code>	—	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:

```

1  #ifdef __AVX512__
2  #define VDSUM(v1, v2, r) ((r) = _mm512_add_pd((v1), (v2)))
3  #define VFSUM(v1, v2, r) ((r) = _mm512_add_ps((v1), (v2)))
4  #define VISUM(v1, v2, r) ((r) = _mm512_add_epi32((v1), (v2)))
5  #elif defined(__AVX__) || defined(__AVX2__)
6  #define VDSUM(v1, v2, r) ((r) = _mm256_add_pd((v1), (v2)))
7  #define VFSUM(v1, v2, r) ((r) = _mm256_add_ps((v1), (v2)))
8  #define VISUM(v1, v2, r) ((r) = _mm256_add_epi32((v1), (v2)))
9  #elif defined(__SSE4__) || defined(__SSE3__)
10 #define VDSUM(v1, v2, r) ((r) = _mm_add_pd((v1), (v2)))
11 #define VFSUM(v1, v2, r) ((r) = _mm_add_ps((v1), (v2)))
12 #define VISUM(v1, v2, r) ((r) = _mm_add_epi32((v1), (v2)))
13 #else
14 #define VDSUM(v1, v2, r) ((r) = (v1) + (v2))
15 #define VFSUM(v1, v2, r) ((r) = (v1) + (v2))
16 #define VISUM(v1, v2, r) ((r) = (v1) + (v2))
17 #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	<code>-ftree-vectorize</code>	enables loops vectorization
	<code>-funroll-loops</code>	enables the loop unrolling (may or may not issue faster code)
	<code>-march=native</code>	specifies the current one as target architecture
	<code>-mtune=native</code>	ask maximum code tuning for the host architecture
clang	<code>-mllvm</code> <code>-force-vector-width=4</code>	(uses LLVM's internal vectorization) enforces vector width (e.g., 4) for SIMD instructions
Intel	<code>-xHost</code> <code>-axFLAG1 -axFLAG2 ...</code>	enables maximum vectorization available on the target cpu 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	<code>-fopt-info-vec-optimized</code>	reports on the successful vectorizations
	<code>-fopt-info-vec-missed</code>	reports on the reasons for unsuccessful vectorization
	<code>-fopt-info-vec-all</code>	reports all vectorization optimizations
clang	<code>-Rpass=loop-vectorize</code>	identifies loops that were successfully vectorized
	<code>-Rpass-missed=loop-vectorize</code>	identifies loops that were NOT successfully vectorized
	<code>-Rpass-analysis=loop-vectorize</code>	identifies statements that caused vectorization to fail (with <code>-fsave-optimization-record</code> , detailed causes may be listed)
Intel	<code>-qopt-report=n</code>	enables diagnostic output; n=1: vectorized, n=2: non-vectorized+reasons, n=3: dependency info, n=4: only vectorization, n=5: dependency issues
	<code>-qopt-report-file=name</code>	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	<code>#pragma GCC ivdep</code>	Ignore potential loop-carried dependencies that are not formally proven
	<code>#pragma GCC unroll n</code>	Unroll the subsequent loop body <code>n</code> times
clang	<code>#pragma clang ivdep</code>	Ignore potential loop-carried dependencies
	<code>#pragma clang vectorize(enable disable)</code>	Enable/disable auto-vectorization of the loop
	<code>#pragma clang vectorize_width(n)</code>	Specify the vector length (VL)
	<code>#pragma clang interleave(enable disable)</code>	Enable/disable unrolling/interleaving of the loop
	<code>#pragma clang interleave_count(n)</code>	Specify how many iterations are to be unrolled/interleaved
Intel	<code>#pragma ivdep</code>	Ignore potential loop-carried dependencies
	<code>#pragma vector always</code>	Vectorize even if the estimated gain is low/negative
	<code>#pragma vector aligned</code>	Assume aligned data
	<code>#pragma vector unaligned</code>	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.

```
1 .L4:  
2     vaddss  xmm0, xmm0, DWORD PTR [rax]  
3     add    rax, 32  
4     vaddss  xmm0, xmm0, DWORD PTR -28[rax]  
5     vaddss  xmm0, xmm0, DWORD PTR -24[rax]  
6     vaddss  xmm0, xmm0, DWORD PTR -20[rax]  
7     vaddss  xmm0, xmm0, DWORD PTR -16[rax]  
8     vaddss  xmm0, xmm0, DWORD PTR -12[rax]  
9     vaddss  xmm0, xmm0, DWORD PTR -8[rax]  
10    vaddss  xmm0, xmm0, DWORD PTR -4[rax]  
11    cmp    rax, rcx  
12    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:

```
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.

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     }
}
```

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.

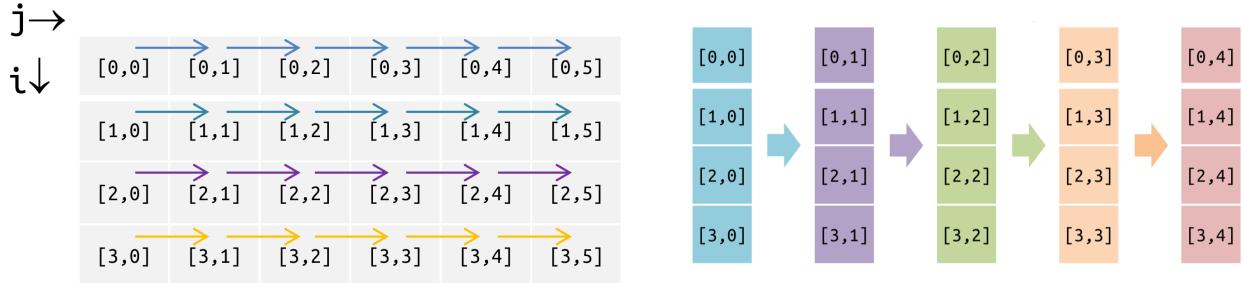


Figure 1.4: RAW data dependency: 4 (or 8) outer iterations can be executed together because there are no vertical dependencies (outer loop vectorization)

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.

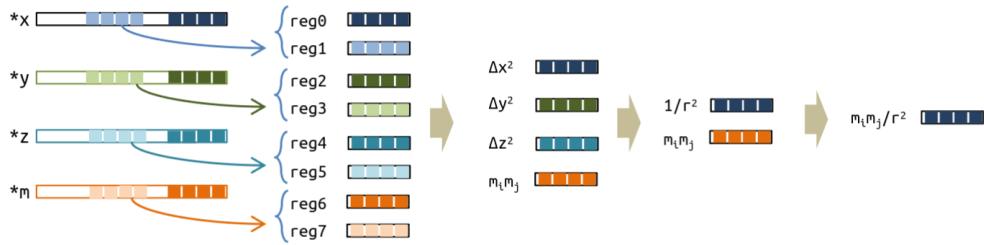


Figure 1.5: AoS vs SoA

⚠️ 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.

```
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.

```
1 /* Declare a vector of 4 doubles */
2 typedef double v4d __attribute__(vector_size,(4*sizeof(double)));
3 /* Declare a vector of 8 doubles */
4 typedef double v8d __attribute__(vector_size,(8*sizeof(double)));
5 /* Declare a vector of 4 integers */
6 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.

```
1 typedef union {
2     v4d v;
3     double d[VD_SIZE];
4 } 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);
```

③ 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&0xFFFFFFFFC;
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.

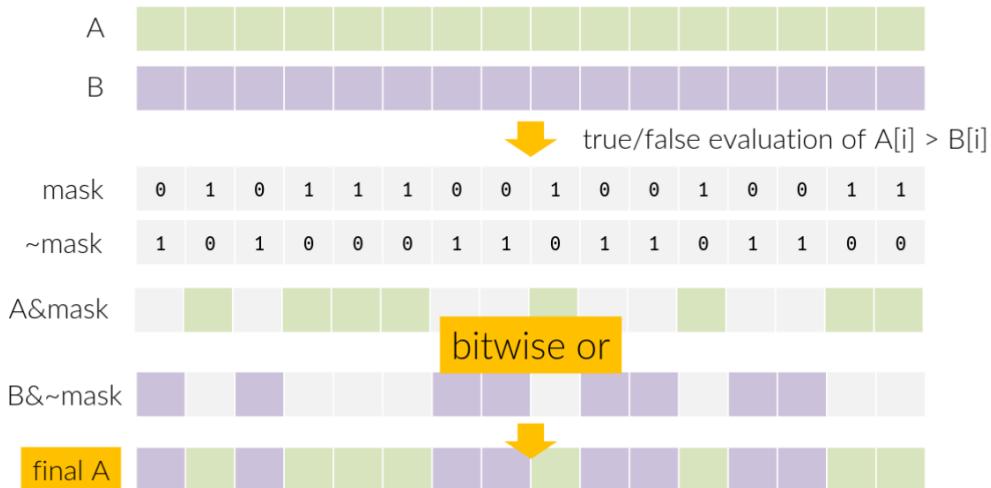


Figure 2.1: Conditional evaluation [1]

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 }
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)&0xFFFFFFFFC;
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);

```

2.1.2 Vectorization by OpenMP SIMD directive

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

The clauses are:

- `private(list)` and `lastprivate(list)`: they have exactly 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;

```

1 #pragma omp simd safelen(8)
2 for (int i = 0; i < N; i++) {
3     a[i] = pow(b[i-k], 3.14); // here the compiler does not know a
                           // priori
4 }
```

- `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.

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.

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 created 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 behaves 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.**

```
1 ...
2 #pragma omp single
3 {
4     int me = omp_get_thread_num();
5     printf( " »Yuk yuk, here is thread %d from "
6             "within the single region\n", me );
7     #pragma omp task
8     {
9         printf( "\tHi, here is thread %d "
10            "running task A\n", me );
11    }
12    #pragma omp task
13    {
14        printf( "\tHi, here is thread %d "
15            "running task B\n", me );
16    }
17 }
18 ...
```

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

```
1 // set the lock, blocking
2 omp_set_lock(&lock);
3 // set the lock, non-blocking
4 omp_test_lock(&lock);
5
```

```

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

```

3.1.1 Memory consistency model

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

Thread a:

```

1 A = 1;
2 read(B);
3 print(B);

```

Thread b:

```

1 B = 1;
2 read(A);
3 print(A);

```

A natural question is: what values can be printed for A and B by the respective threads? To answer this, let's introduce the concept of **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.

...

```

1 #pragma omp parallel num_threads(2)
2 {
3     int tid = omp_get_thread_num();
4     if (tid == 0) {
5         x = 1;
6         #pragma omp flush(x)
7         y = 1;
8         #pragma omp flush(y)
9     }
10    else {
11        int local_x, local_y;
12        #pragma omp flush(y)
13        local_y = y;
14
15        if (local_y == 1) {
16            local_x = x;
17        }
18    }
19 }

```

OpenMP utilizes a **relaxed consistency** memory model, which means that each thread's view of memory can temporarily diverge from the actual contents of main memory. In other words, updates

performed by one thread may not be immediately visible to others unless specific actions are taken to ensure consistency. This flexibility allows for significant performance improvements, but also places a responsibility on the programmer to manage memory visibility, often using `flush` or synchronization constructs.

Within this relaxed model, certain rules must still be followed. All so-called “S operations” (synchronization operations, such as barriers, flushes, and locks) are required to appear in the same order across all threads, maintaining a sequentially consistent view for these events. Furthermore, within a single thread, memory reads and writes (R and W operations) cannot be reordered with respect to S operations—the order between computation and synchronization is preserved. The weak consistency model thus enforces ordering only at synchronization points, guaranteeing, for example, that $S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, and $S \rightarrow S$ are respected. This ensures that, even though memory can appear temporarily out-of-date, any synchronization will properly reconcile views between threads and allow safe, predictable parallel programming.

...

Implicit flushes

```
1 #pragma omp barrier
2 #pragma omp critical
3 #pragma omp ordered
4 #pragma omp parallel
5 #pragma omp for
6 #pragma omp single
7 #pragma omp master
```

⚠ Warning: Locks

Lock operations (`omp_set_lock`, `omp_unset_lock`) have implicit flush of all variables.

...

```
1 seq_cst // sequentially consistency (default)
2 acq_rel // acquire-release
3 release // release
4 acquire // acquire
5 relaxed // relaxed (no synchronization)
```

- `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.
- `acq_rel`: is the acquire-release memory order. It is a mix of acquire and release. It is less strict than `seq_cst` and it is cheaper. It is used when you need to ensure that all operations are executed in the order they are written in the code, but you do not need to ensure that all operations are executed in the order they are written in the code.
- `release`: it is like doing a write back of whatever is in the local cache to the main memory.
- `acquire`: it is like doing a read from the main memory to the local cache.
- `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.

Data race

💡 Tip: Debugging data races

The compiler flag `-fsanitize=thread` can be used to detect data races:

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

- Start with correct, simple synchronization (`seq_cst`, `critical`)
- ...
- ...
- ...

❓ Advanced Concept: Locks hints

Note: the locks can be initialized with an hint about what will be their typical usage:

```
1 omp_init_lock_with_hint(omp_lock_t *t, omp_lock_hint_t hint);
2 void omp_init_nest_lock_with_hint (omp_lock_t *t, omp_lock_hint_t
   hint);
3
4 typedef enum omp_lock_hint_t {
5     omp_lock_hint_none = 0,           // No hint
6     omp_lock_hint_uncontended = 1,    // lock is usually uncontended
7     omp_lock_hint_contented = 2,     // lock is often contended
8     omp_lock_hint_nonspeculative = 4, // prefer non-speculative lock
9     omp_lock_hint_speculative = 8,    // speculative lock allowed
10 } omp_lock_hint_t;
```

The values are powers of two (bitmask), so they can be combined with the bitwise OR operator. E.g., you can use `omp_lock_hint_contented | omp_lock_hint_speculative` to combine two hints. Each value represents a specific optimization the runtime may apply for the lock.

!!! At the time of writing (2025), it is not really supported by the compilers !!!

3.1.2 Controlling the task creation - Clauses

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

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

Clause	Description
<code>if (expr)</code>	When <code>expr</code> evaluates <code>false</code> , 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 <i>undefined</i> and started. The task suspended is resumed afterwards.
<code>final (expr)</code>	The <code>final(expr)</code> clause can be used to suppress the task creation and then to control the tasking overhead, especially in recursive task creation. If <code>expr</code> evaluates <code>true</code> , no more tasks are generated and the code is executed immediately. That is propagated to all the children tasks. That is called an <i>included</i> task and is always undefined. Use <code>omp_in_final()</code> to check if the task is a final one.
<code>mergeable</code>	This clause avoids a separated data environment to be created if a task is undefined or included.
<code>tied, untied</code>	This clause lets a suspended task be resumed by a different thread than the one that started it. The default option is <code>tied</code> .
<code>depend, priority</code>	<i>We will cover this afterwards</i>

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 for-loops // (perfectly nested)
```

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.

3.1.3 Task dependencies

Often, there are data dependencies among different tasks: a given tasks may have to use the results of another one, or in any case to wait for its operations to terminate.

There are 3 types of dependencies:

- `IN`:

the task will be dependent on a *previously generated* task if that task has an `OUT`, `INOUT`, or `MUTEXINOUTSET` dependence on the same *memory region*.

- **OUT , [INOUT]* :**

the task will be dependent on a *previously generated* task if that task has an **IN** , **OUT** , or **MUTEXINOUTSET** dependence on the same *memory region*.

- **MUTEXINOUTSET :**

the task will be dependent on a previously generated task if that task has an **IN** or **OUT** dependence on the same *memory region*; it will be *mutually exclusive* with another **MUTEXINOUTSET** sibling task, meaning that they can be executed in any order but not at the same time.

Observation: INOUT

In modern OpenMP standards, **INOUT** dependence type is considered obsolete and is treated as an alias for **OUT** . Its use is discouraged in new code.

- **RaW (Read after Write):**

When an instruction (task) depends on the result of another instruction (task), that is called a **flow dependency** or **true dependency** and referred as **Read-After-Write** (you need to read a result after it is written)

```
1 #pragma omp task depend(out:the_answer)
2 function_wise(*the_answer);
3
4 #pragma omp task depend(in:the_answer)
5 function_curious(*the_answer);
```

- **WaR (Write after Read):**

When an instruction (task) may change the result of another instruction (task), the ordering must be strict. That is called **anti-dependency** and referred as **Write-After-Read**

```
1 #pragma omp task depend(in:the_question)
2 function_sage(*the_question);
3
4 #pragma omp task depend(out:the_question)
5 function_curious(*the_question);
```

- **WaW (Write after Write):**

When two instructions (tasks) are independent of each other but writes in the same memory location, then there is an **output dependency** or a **Write-After-Write**

That may also be considered a **false dependency** or **name dependency**, as it may be sufficient a variable renaming to remove it

```
1 function_sage(*the_question);
2 function_curious(*the_question);
```

- **RaR (Read after Read):**

When two instructions (tasks) read the same memory region, there is actually no dependency

```
1 #pragma omp task depend(in:the_question)
2 function_wise1(*the_question);
3
4 #pragma omp task depend(in:the_question)
```

```
5  function_wise2(*the_question);
```

function_wise2

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.

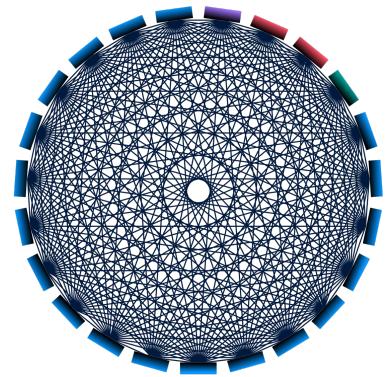


Figure 4.1: Dragonfly+ Topology. Booster Module nodes (blue), I/O cell (purple), Data-centric cells (red), and Hybrid cell (green) [2].

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:

```
1 #!/bin/bash
2
3 # Slurm directives
4 #SBATCH --job-name=my_gpu_job          # Job name
5 #SBATCH --nodes=1                      # Number of nodes
6 #SBATCH --ntasks-per-node=1            # Number of MPI tasks per node
7 #SBATCH --cpus-per-task=8              # Number of CPU cores per task
8 #SBATCH --gres=gpu:4                  # Request 4 GPUs per node
9 #SBATCH --mem=494000                  # Memory in MB (approx 480 GB)
10 #SBATCH --time=00:30:00               # Wall-clock time limit (hh:mm:ss)
11 #SBATCH --partition=boost_usr_prod   # Partition (queue) to submit to
12 #SBATCH --account=tra25_gput          # Account/project code
13 #SBATCH --reservation=s_tra_gput      # (Optional) reservation name
14 #SBATCH --output=job_%j.out           # Standard output file (%j = job ID)
15 #SBATCH --error=job_%j.err            # Standard error file
16
17 # Load necessary modules
18 module <module-name>
19
20 # Run the executable
21 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 `dcp_gput_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.

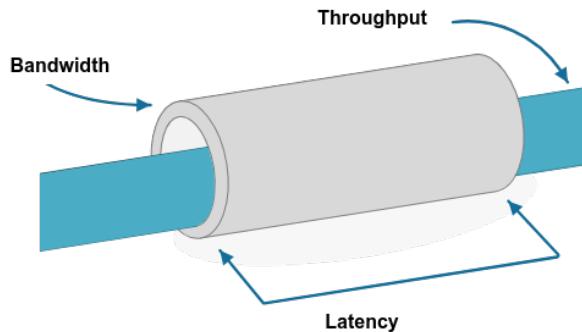
This chapter introduces *accelerated computing*, a paradigm where a specialized, highly parallel processing unit, the Graphics Processing Unit (GPU), is used in conjunction with a traditional Central Processing Unit (CPU) to accelerate computationally intensive portions of an application. We will explore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

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).

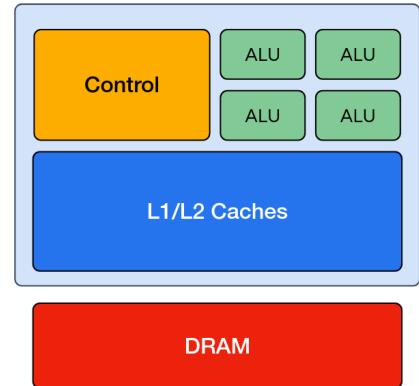


Figure 5.1: Simplified CPU [4]



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.

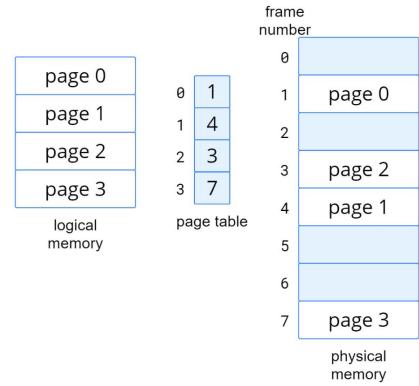


Figure 5.3: Translation of virtual to physical addresses.



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.

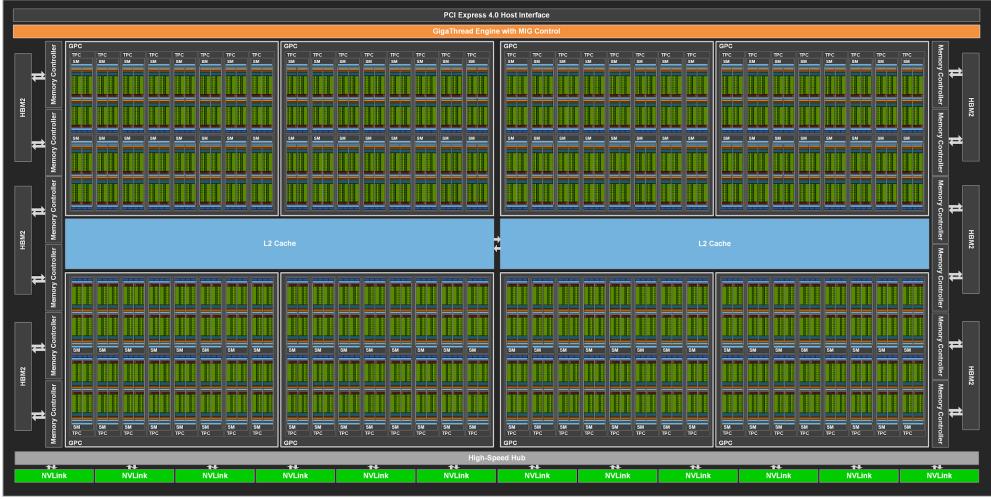


Figure 5.4: High-level overview of the NVIDIA Ampere A100 GPU architecture, showing the arrangement of GPCs and SMs.

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 different 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.

```
1 __global__ void GPUFunction() {
2     printf("Hello from GPU!\n");
3 }
4
5 int main() {
6     CPUFunction();
7     GPUFunction<<<1, 1>>>();
8     cudaDeviceSynchronize();
9 }
```

- `__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.

```
1 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.

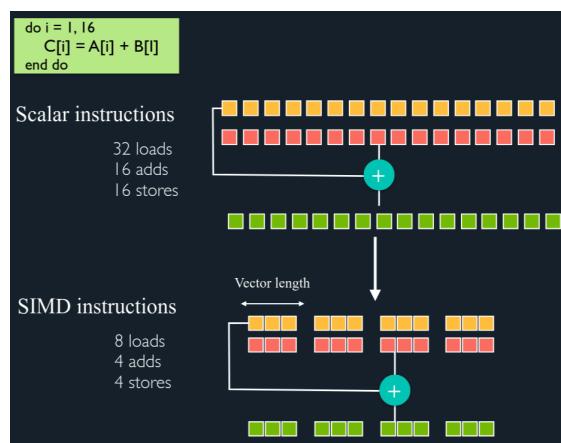


Figure 5.6: SIMT vs SIMD

- **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.

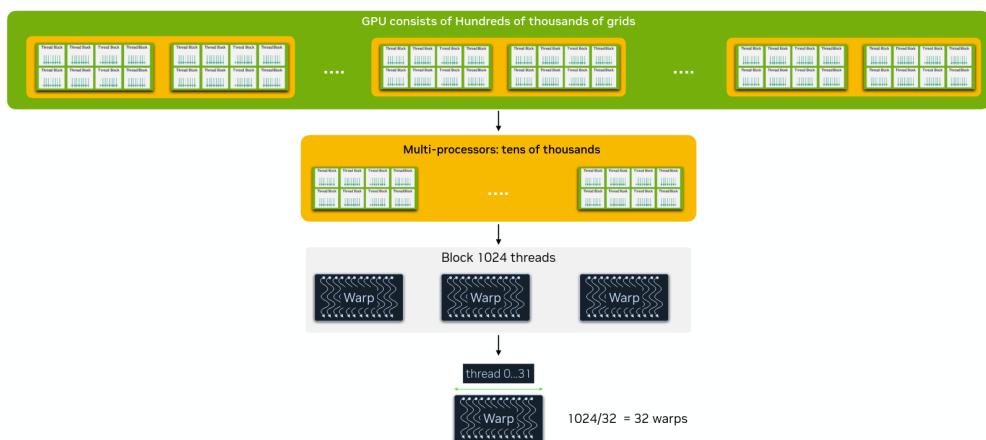


Figure 5.7: GPU Thread Hierarchy

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

```

1 int N = 10000;
2 size_t size = N * sizeof(int);
3
4 int *a;
5 a = (int *) malloc(size);
6
7 free(a);

```

GPU code

```

1 int N = 10000;
2 size_t size = N * sizeof(int);
3
4 int *a;
5 cudaMallocManaged(&a, size);
6
7 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.

```

1 __global__ vectorSum(int N) {
2     int idx = threadIdx.x + blockIdx.x * blockDim.x;
3     if (idx < N) { // only do work if it does}
4 }

```

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

On A100 typically 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 tasks with blocks containing 256 threads. How do we choose the optimal block size?

```

1 int N = 100000; size_t threads_per_blocks = 256;
2 size_t number_of_blocks = (N + threads_per_block - 1) /
    threads_per_block;
3 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

```
1 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 which problem occurred, we can use the `cudaGetErrorString()` function.

```
1 cudaError_t err;
2 err = cudaMallocManaged(&a, size);
3 if (err != cudaSuccess)
4     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.

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

Nsight Compute profiler

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

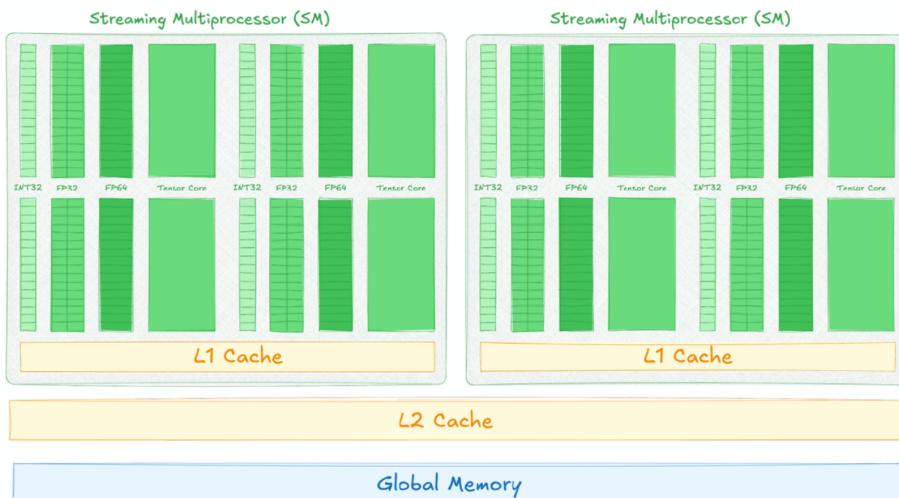


Figure 5.8: Memory Hierarchy

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 default 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

Bandwidth	Latency
13×	1×
3×	5×
1×	15×

Figure 5.9: Bandwidth and Latency comparison

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.

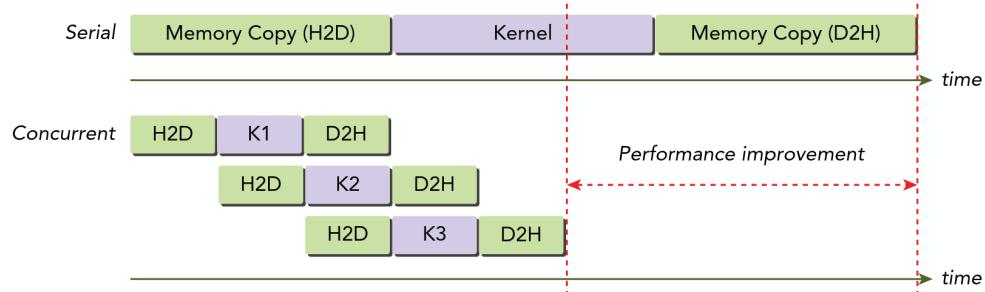


Figure 5.10: CUDA Streams Overlap

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

?

Advanced Concept: *non-blocking streams*

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

6

OpenACC

```
1 #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 of 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, parallelize 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.

```
1 #pragma acc parallel
2 {
3     for (int i = 0; i < N; i++)
4         A[i] = B[i] + C[i];
5 }
```

- `loop`: identifies a loop that should be distributed across threads. Parallel and loop are often placed together

```
1 #pragma acc parallel
2 {
3     #pragma acc loop
4     for (int i = 0; i < N; i++)
5         A[i] = B[i] + C[i];
6 }
```

- `kernel`: a parallel region of code. It allows the programmer to step back, and rely solely on the compiler.

```

1 // example of a kernel that performs two different operations
2 #pragma acc kernels
3 {
4     for (int i = 0; i < N; i++)
5         A[i] = B[i] + C[i];
6
7     for (int i = 0; i < N; i++)
8         D[i] = A*x[i] + B*y[i];
9 }
10
11 // loop independent: the compiler can parallelize the loop
12 #pragma acc kernels loop independent
13 {
14     for (int i = 0; i < N; i++)
15         A[i] = B[i] + C[i];
16 }
```

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 target
-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 CPU
-gpu=cc80	map OpenACC parallelism to an NVIDIA GPU, compile target
-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

```
1 int *A = (int*) malloc(N * sizeof(int));
2 #pragma acc parallel loop
3 {
4     for (int i=0; i<N; i++) A[i] = 0;
5 }
```

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

```
1 for (int i=0; i<N; i++) A[i] = 0;
2 #pragma acc parallel loop copy(A[0:N])
3 {
4     for (int i=0; i<N; i++) A[i] = 1;
5 }
```

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

```
1 #pragma acc parallel loop copyin(A[0:N]) copyout(B[0:N])
2 {
3     for (int i=0; i<N; i++) B[i] = A[i];
4 }
```

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.

```
1 #pragma acc parallel loop
2 for (int i = 0; i < N; i++) {
3     #pragma acc loop
4     for (int j = 0; j < M; j++) {
5         #pragma acc loop seq
6         for (int k = 0; k < K; k++) {
7             A[i][j][k] = B[i][j][k] * C[i][j][k];
8         }
9     }
10 }
```

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.

```
1 #pragma acc parallel loop collapse(2)
2 for (int i = 0; i < N; i++) {
3     for (int j = 0; j < M; j++) {
4         // code
5     }
6 }
```

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.

```
1 #pragma acc parallel loop tile(32,32)
2 for (int i = 0; i < 128; i++) {
3     for (int j = 0; j < 128; j++) {
4         // code
5     }
6 }
```

⌚ 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. Its architecture is composed of 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 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
```

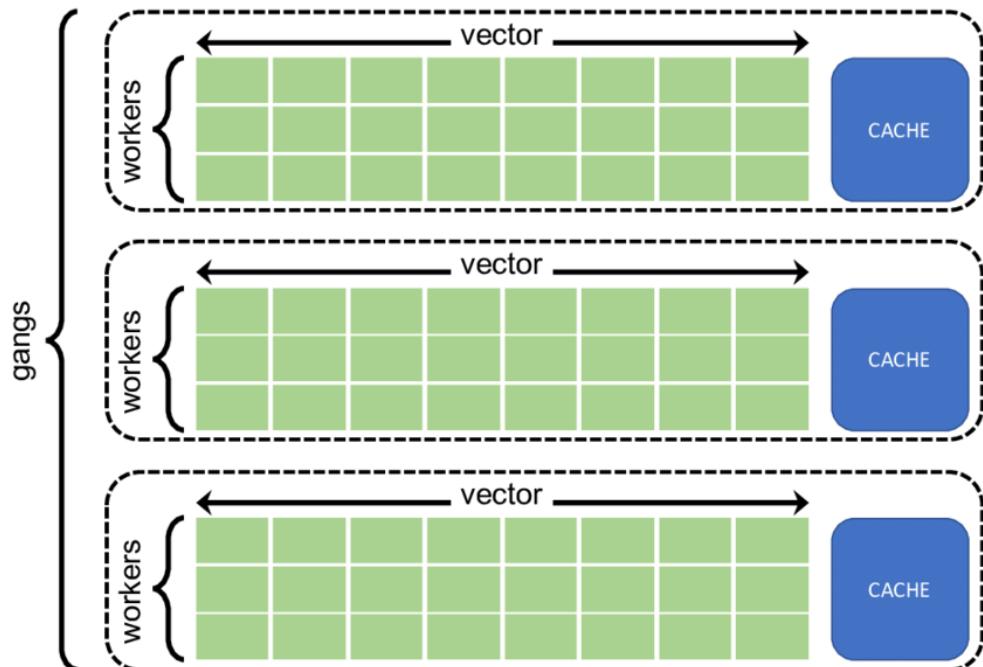


Figure 6.1: Gang, Worker and Vector Hierarchy

💡 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

7

MPI and GPU offloading using Python

7.1 MPI in Python

Message Passing Interface (MPI) is a standard for distributed-memory parallel computing, used to enable communication between multiple processes. MPI is useful in HPC applications, where we need to distribute 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

```
1 if rank == 0:  
2     data = "Hello, Process 1"  
3     comm.send(data, dest=1)  
4 elif rank == 1:  
5     data = comm.recv(source=0)  
6     print(f"Process 1 received: {data}")
```

- Non-blocking communications

```
1 if rank == 0:  
2     data = "Hello, Process 1"  
3     comm.isend(data, dest=1, tag=11)  
4 elif rank == 1:  
5     data = comm.irecv(source=0, tag=11)  
6     print(f"Process 1 received: {data}")
```

- Collective communications

```
1 comm.bcast(data, root=0)
```

③ 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.

```
1 import numpy as np
2 import mpi4py
3 from mpi4py import MPI
4
5 npart_data = int(sys.argv[1])
6 Nbins = 20
7
8 comm = MPI.COMM_WORLD
9 rank = comm.Get_rank()
10 size = comm.Get_size()
11
12 if rank == 0:
13     Data1 = np.random.rand(npart_data, 2)
14     Data2 = np.random.rand(npart_data, 2)
15     bins = np.linspace(0, 1.5, Nbins)
16 else:
17     Data1 = None
18     Data2 = np.empty((npart_data, 2))
19     bins = np.empty(Nbins)
20
21 comm.Barrier()
22
23 comm.Bcast(Data1, root=0)
24 comm.Bcast(Data2, root=0)
25
26 Data1_on_task = scatter_data(Data1)
27
28 dist_hist = get_statistical_distance([Data1_on_task, Data2, bins])
29
30 dist_hist_fin = None
31 if rank == 0:
32     dist_hist_fin = np.empty([size, len(dist_hist)])
33
34 comm.Gatherv(dist_hist, dist_hist_fin, root=0)
35
36 if rank == 0:
37     dist_hist_fin = np.sum(dist_hist_fin, axis=0)
```

where `scatter_data` is a function that scatters the data to the different tasks.

```

1 def scatter_data(Data):
2     rows_per_rank = np.full(size, npart_data // size)
3     rows_per_rank[:npart_data % size] += 1
4
5     # Allocate memory for the data on the current task
6     Data1_on_task = np.empty((rows_per_rank[rank], 2))
7
8     # Calculate the displacement
9     displacement = np.cumsum(np.insert(rows_per_rank[:-1], 0, 0)) *
10        2
11
12     # Scatter the data
13     comm.Scatterv([Data1, rows_per_rank * 2, displacement, MPI.
14                   DOUBLE], Data1_on_task, root=0)
15
16     return Data1_on_task

```

7.2 GPU offloading in Python

It is possible to offload the computation to the GPU using different libraries.

7.2.1 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.

```
1 ...
```

7.2.2 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.

```
1 ...
```

7.2.3 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 comparable 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.

```

1 import cupy as cp
2
3 def distances_GPU_cupy(Data1, Data2, bins):
4     Data1 = cp.asarray(Data1)
5     Data2 = cp.asarray(Data2)
6     bins = cp.asarray(bins)
7
8     dist_hist = get_distances_GPU_cupy(Data1, Data2, bins)

```

```

9     return dist_hist
10
11 def get_distances_GPU_cupy(Data1, Data2, bins):
12     # calculate the distances in a vectorized manner
13     dx = Data1[:, 0][:, cp.newaxis] - Data2[:, 0][cp.newaxis, :]
14     dy = Data1[:, 1][:, cp.newaxis] - Data2[:, 1][cp.newaxis, :]
15     dist = cp.sqrt(dx**2 + dy**2)
16
17     # use cp.histogram for GPU histogramming
18     dist_hist = cp.histogram(dist, bins=bins)
19     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 executed 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.

8

Message Passing Interface - 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

Two-Sided communications

By its very nature, the message-passing paradigm is designed around the concept of cooperative exchange of informations among two or more processes whose address space are isolated and not directly inaccessible by other processes.

8.1 Remote Memory Access - RMA

8.1.1 Memory Windows

⚠ 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:

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

⌚ Observation: *Flush vs Sync*

The fundamental difference between `MPI_Win_flush` and `MPI_Win_sync` is that:

- `MPI_Win_flush`: Forces completion of RMA operations from the origin to the target.
- `MPI_Win_sync`: Synchronizes local memory with the window (target-side operation).

Or, in other words:

- `MPI_Win_flush` is about *Network Completion* (*Did the data arrive?*)
- `MPI_Win_sync` is about *Memory Consistency* (*Is the data visible to the CPU?*)

This distinction is critical and often misunderstood: these calls operate on different sides of the communication and serve completely different purposes.

Locks

Locks are a way to synchronize the access to a shared resource. They are a way to ensure that only one process can access the shared resource at a time.

Passive mode: the target does not participate in operations: one sided asynchronous communication.

```
1 MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win);
2
3 MPI_Win_unlock(int rank, MPI_Win win);
4
5 MPI_Win_flush/flush_local(int rank, MPI_Win win);
```

where `lock_type` is one of the following:

- `MPI_LOCK_EXCLUSIVE`: only one process can access the shared resource at a time.
- `MPI_LOCK_SHARED`: multiple processes can access the shared resource at a time.

`rank` is the rank of the process that will access the shared resource.

`assert` is the assertion that will be used to synchronize the access to the shared resource.

- `MPI_Win_flush`: complete operations on remote target process; data will be then available to the target task, or to other tasks
- `MPI_Win_flush_local`: locally complete operations to the target process

⚠ Warning: Misconception about locks

Despite the unfortunate naming, MPI locks are not mutex-like. They do not provide mutual exclusion for the target process's local access. The target can modify its window memory during a locked epoch, leading to race conditions.

Assertions

8.2 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.

8.2.1 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 characteristics:

- 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.):

```

1 MPI_Comm_split_type (
2     MPI_Comm comm,           // the communicator to be splitted
3     int split_type,         // the splitting property
4     int key,                // hint for new ranks
5     MPI_Info info,          // helper for the splitting property
6     MPI_Comm *newcomm        // the new communicator
7 );

```

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 characteristics.

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

Placement

```

1 MPI_Win_allocate_shared (
2     MPI_Aint size,           // the size of the window
3     int disp_unit,           // the displacement unit
4     MPI_Info info,          // the information to be used
5     MPI_Comm comm,           // the communicator
6     void *baseptr,           // the base pointer
7     MPI_Win *win             // the window
8 );

```

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 can allow the MPI implementation to more flexibly optimize the placement, possibly enhancing the affinity for each individual task.

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:

```

1 MPI_Win_shared_query(
2     MPI_Win    win,           // the window object
3     int       rank,          // rank whose segment to query, or
4             MPI_PROC_NULL
5     MPI_Aint *size,         // returns segment size
6     int       *disp_unit,   // returns displacement unit
7     void     **baseptr      // returns base pointer in shared memory
8 );

```

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.

8.3 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:

```

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

```

The possible issues, or inconveniences, 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 \leftrightarrow 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)
i=2	R4	R5	R2:(0,1) R3:(1,1) R4:(0,2) R5:(1,2)

8.3.1 Core API Functions

MPI_Dims_create

Computes a "balanced" distribution of processes across dimensions:

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

MPI_Cart_create

Creates a new communicator with processes laid out on a Cartesian grid:

```
1 int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
                     const int periods[], int reorder, MPI_Comm *comm_cart);
```

Key behavior: Pre-set dimensions ($\text{dims}[i] > 0$) are preserved; zero entries are computed.

The algorithm tries to minimize surface-to-volume ratio for communication efficiency.

③ Example: 2D grid, 12 processes

```
1 // unconstrained: let MPI choose
2 int dims[2] = {0, 0};
3 MPI_Dims_create(12, 2, dims); // Result: dims = {4, 3} or {3, 4}
4
5 // constrained: force 4 columns
6 int dims2[2] = {0, 4};
7 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:

```
1 int MPI_Cart_create(
2     MPI_Comm comm_old,           // Input communicator
3     int ndims,                 // Number of dimensions
4     const int dims[],           // Size of each dimension
5     const int periods[],        // Periodicity flags (0 or 1)
6     int reorder,                // Allow rank reordering?
7     MPI_Comm *comm_cart         // Output: new Cartesian comm
8 );
```

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.

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.