UniTs - University of Trieste

Faculty of Scientific and Data Intensive Computing

Department of mathematics informatics and geosciences

Advanced High Performance Computing

Lecturer:
Prof. Luca Tornatore

Author:
Andrea Spinelli

November 13, 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.

github.com/Spina02 andreaspinelli2002@gmail.com

Preface

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

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

Contents

Draft

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.

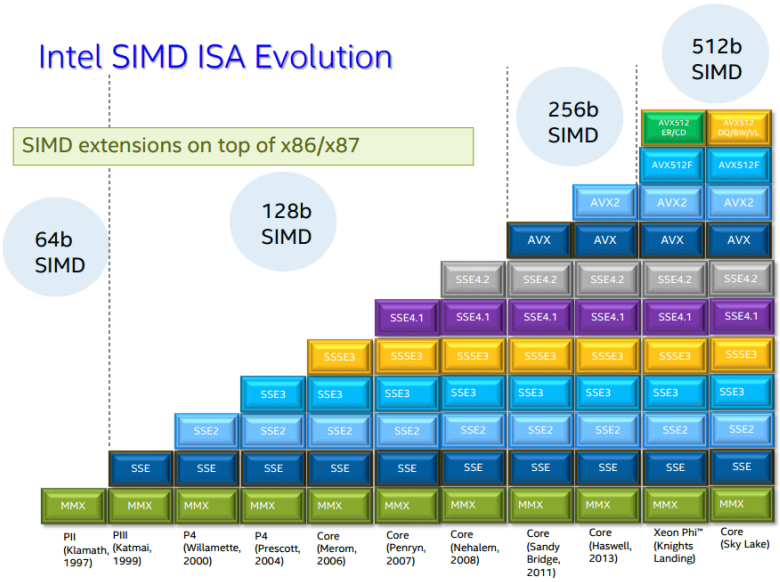

Figure 1.1: Intel SIMD ISA Evolution: timeline of vector instruction set extensions introduced with each CPU generation. The diagram shows how SIMD register width increased from MMX and SSE (64-128 bits) to AVX (256 bits) and AVX-512 (512 bits), with each new standard supporting more data parallelism.

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.

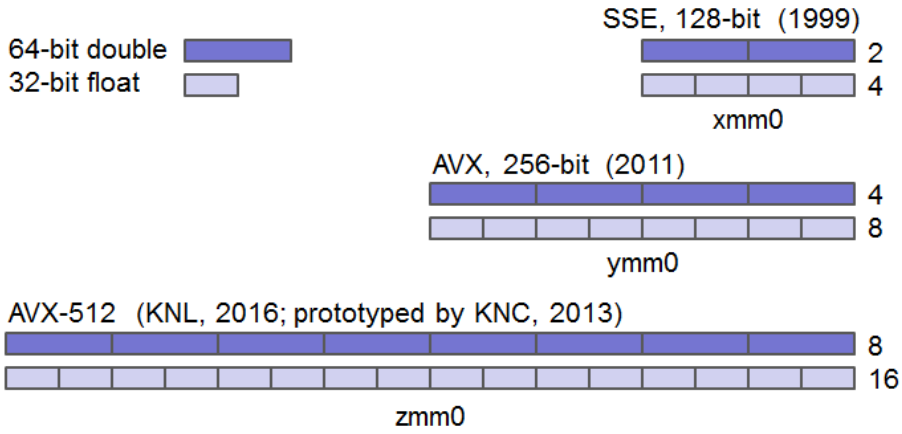

Figure 1.2: SIMD registers: each generation widens the registers, allowing more doubles (dark) or floats (light) to be processed in parallel. [2]

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

How to achieve vectorization?

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

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

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

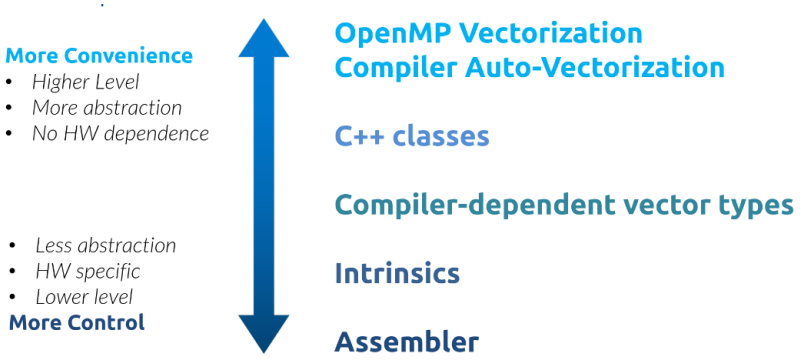

Figure 1.3: Approaches to vectorize, from high-level convenience to low-level control. [1]

Different architectures

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

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

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

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

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

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

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

1.2 Checking for SIMD capabilities

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

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

```
#include <cpuid.h>

int main() {
    __builtin_cpu_init();

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

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

```
#include <immintrin.h>

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

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

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

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

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

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

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

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

1.3 Vector Types

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

INTEGER types

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

FLOATING-POINT types

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

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

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

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

Example: *Vector summation*

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

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

Warning: *Production-ready headers*

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

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

1.4 Loops auto-vectorization

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Here, we are using the `vaddss` instruction which is actually a scalar instruction.

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

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

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

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

Tip: ***Countable While Loops***

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

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

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

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

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

1.4.1 Data dependencies

TODO: enhance this section

$S_1, S_2 \rightarrow M$

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

where:

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

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

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

An of false dependency is:

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

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

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

Non-contiguous memory access

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

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

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

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

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

This loop can NOT be vectorized without leading to wrong results.

However, let's consider the following 2D case:

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

```
6 }
```

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

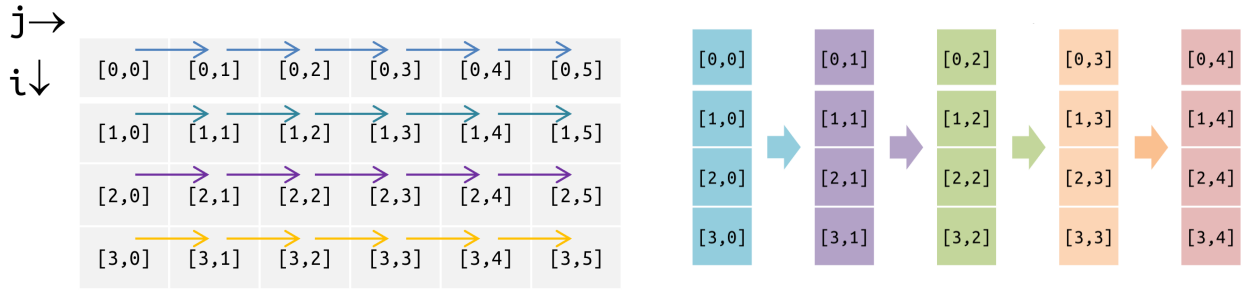

Figure 1.4: RAW data dependency: 4 (or 8) outer iterations can be executed together because there are no vertical dependencies (outer loop vectorization)

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.

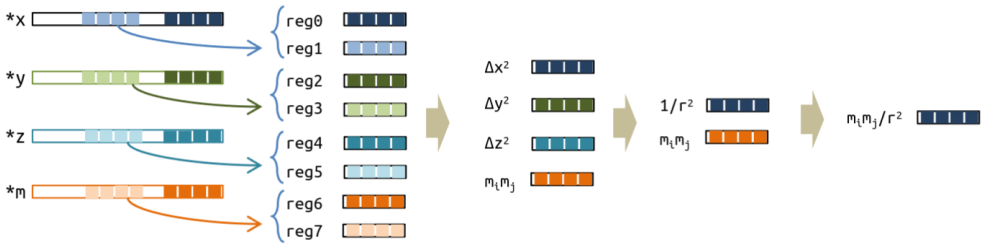

Figure 1.5: AoS vs SoA

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.

Draft

2 Vector Types

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

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

where:

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

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

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

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

2.1 Memory Alignment

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

…

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

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

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

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

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

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

Example: *Working with vectors*

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

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

At first we need to define the appropriate vector variables:

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

Now we can actually implement the sum:

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

2.1.1 Conditional evaluation

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

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

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

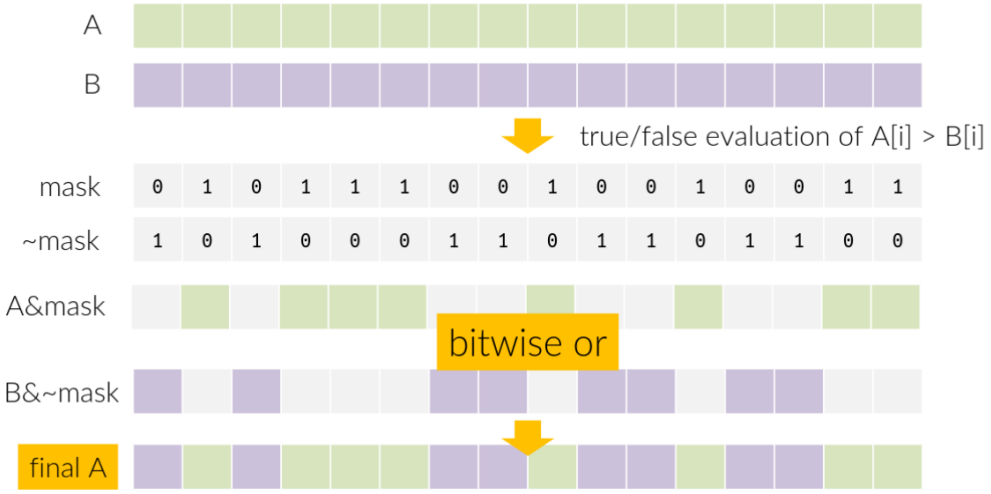

Figure 2.1: Conditional evaluation [1]

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

```
1  void kernel( double *A, double *B, int N ) {
2      // element-wise swap of A and B so that
3      // A[i] > B[i] for every i
4
5      for ( int i = 0; i < N; i++ ) {
6          double min = (A[i]>B[i] ? B[i] : A[i]);
7          double max = (A[i]>=B[i] ? A[i] : B[i]);
8          A[i] = max;
9          B[i] = min;
10     }
11     return;
12 }
```

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

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

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

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

2.1.2 Vectorization by OpenMP SIMD directive

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

The clauses are:

- `private(list)` and `lastprivate(list)`: they have exatly the same meaning as in "thread-related" OpenMP
  The execution instances in this context are the SIMD lanes; hence, an instance of every private variables is created per SIMD lane.
- `reduction(identifier: list)`;
- `collapse(n)`;
- `simdlen(length)`;
- `safelen(length)`: It is used to specify at the compiler the maximum length not having dependences inside the loop;

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

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

2.1.3 The omp SIMD functions

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

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

the possible clauses are:

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

Draft

3 OpenMP

3.1 Tasks

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

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

Tasks often interact through shared data, giving rise to dependencies. These dependencies form a **task dependency graph**, where nodes are tasks and directed edges represent data dependencies (i.e., one task must complete before another can start due to shared data). It is crucial that this graph is **acyclic** ensuring there are no circular wait conditions or deadlocks, and allowing the runtime system to correctly schedule and execute tasks respecting all required data dependencies.

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

Since version 3.0, OpenMP introduced the **task** construct, which provides an elegant and flexible way to handle precisely these kinds of problems—where execution flow is irregular and only determined at runtime. When defining a task, OpenMP internally creates a ”unit of work,” bundling together all necessary code, data, and local variables. This work is then scheduled for execution at some future point.

Behind the scenes, OpenMP employs a queuing system to orchestrate the assignment of these tasks to the available threads, efficiently balancing the workload and enabling dynamic parallelism.

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

To make it clearer: creating tasks does not mean creating threads. The tasks are “units of work” that are assigned to the running threads. The pool of threads is unaltered(unless, of course, there is nested parallelism involved) and mapped onto the underlying physical cores.

....

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

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

Scheduler points

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

Scope of variables

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

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

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

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

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

Let's remind the following rules:

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

…

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

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

---

locks

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

```

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

Draft

Bibliography

[1] *Advanced-High-Performance-Computing-2024*. https://github.com/Foundations-of-HPC/Advanced-High-Performance-Computing-2024.

[2] *Cornell Virtual Workshop > Vectorization > Vector Hardware > Registers — cvw.cac.cornell.edu*. https://cvw.cac.cornell.edu/vector/hardware/registers.

Draft