OpenMP for Computational Scientists

3: Vectorisation and optimisations

Tom Deakin
University of Bristol

Outline

Now you know how to parallelise programs using OpenMP, how do you write fast programs in OpenMP?

- The cache hierarchy
- Performance analysis
- Vectorisation
- Array of structures vs Structure of arrays
- Memory access patterns
- Memory alignment

Recap

- Data sharing clauses:
  - `shared`, `private`, `firstprivate`, `lastprivate`
- Atomics and `critical` regions
- False sharing and cache thrashing
- Reductions with the `reduction` clause

Combined with the `parallel` and worksharing constructs from before, we've covered the OpenMP "common core".

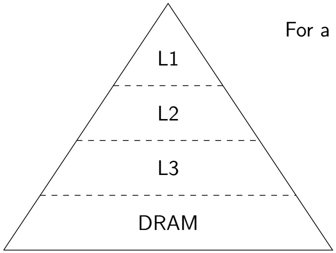
Previous exercise

Take your parallel 5-point stencil, and implement a reduction:

```
total = 0.0
!$omp parallel do collapse(2) reduction(+:total)
do i = 1, nx
  do j = 1, ny
    Atmp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j) + A(i,j-1)
    ↪   + A(i,j+1)) / 5.0
    total = total + Atmp(i,j)
  end do
end do
!$omp end parallel do
```

- Well done if you managed this!
- 5-point stencil is simple, but captures the *essence* of more complicated codes.
- Extension: did anyone try the parallelising the Jacobi solver?

Cache hierarchy



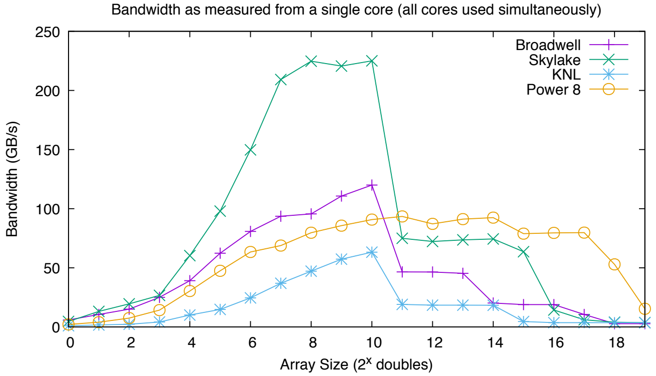
For a Skylake processor

L1	4 cycles
L2	12 cycles
L3	$\sim$ 44 cycles
DRAM	$\sim$ 90 ns ($\sim$ 200 cycles @ 2.2 GHz)

- Most integer and floating point operations are single cycle.
- Memory access is relatively slow.
- Moving memory between nodes is hugely expensive: $\sim 3\mu s$
- How long is a nanosecond? 11.8 inches — Grace Hopper: `https://youtu.be/JEpsKnWZrJ8`.
- Therefore very easy to become bound by memory movement.

Cache bandwidth

Graph of aggregate cache bandwidth on different architectures:



- Clear cliff edges at cache capacity sizes (3 times x-axis).
- As with latency: more performance from lower levels.

[1] Deakin, T., Price, J., and McIntosh-Smith, S. (2017). Portable Methods for Measuring Cache Hierarchy Performance (poster). In Supercomputing. Denver, CO.

Streaming data

STREAM Triad kernel:

```
!$omp parallel do
do i = 1, N
  a(i) = b(i) + scalar * c(i)
end do
!$omp end parallel do
```

- Where `N` is large, the arrays exceed cache capacity.
- This kernel has *no* data reuse: data items are read or written once, then never used again.
- Example of a *streaming* data access pattern.
- Performance is then bound by main memory bandwidth.

Performance analysis

- Optimisations can help code go faster, but how do you know when it's performing *well*?
- Helpful to think about characteristics of the algorithm:
  - Algorithmic complexity for compute.
  - Algorithmic complexity for data movement.
- Examples:
  - Vector-vector and vector-matrix are $O(n)$ for compute and data movement.
  - Matrix-matrix multiply is $O(n^3)$ for compute and $O(n^2)$ for data movement.
  - Matrix multiplication becomes *compute bound* at large enough n, but other examples remain memory bandwidth bound.

Rate limiting factors

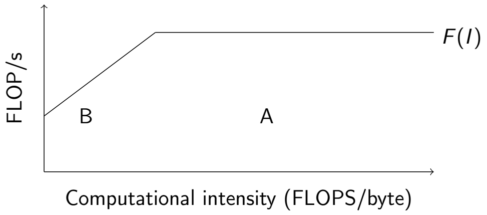
- Most HPC codes are *memory bandwidth bound*.
- A few are *compute bound*.
- Other possibilities:
  - Network bound (e.g. MPI communication).
  - I/O bound (e.g. writing to the filesystem).
  - Memory latency bound.
  - Memory capacity bound.
  - ...
- Worth thinking about the bound for your own code.
- Arithmetic (integer and floating point) is very cheap $O(1)$ cycle.
- Division, transcendentals, exponentials/logs are relatively slow.
- Load/store is 2–3 times slower than an arithmetic operation, even if it's an L1 cache hit (the best case).
- Consider the ratio of bytes moved vs. floating point operations.

Computational intensity

- The ratio of FLOPS to bytes moved is known as *computational intensity*, or CI.
- Originally only DRAM traffic counted, but this causes problems.
- Bytes moved is best calculated from the kernel perspective.
- Take this example:
  - `a(i) = a(i) + b(i) * c(i)`
  - Assume FP64 arrays.
  - Count the data movement and floating point operations for each `i`.
  - 24 bytes loaded, 8 bytes stored.
  - Two floating point operations: one `+` and one `*`.
  - CI of $2/32 = 1/16$.

Roofline model

Useful conceptual tool to establish whether compute or memory bandwidth bound.



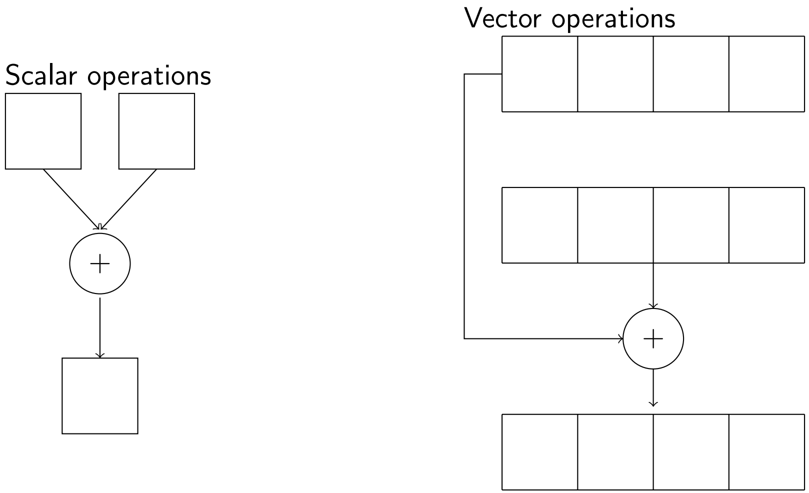
- The roof $F(I)$ is found from tech sheet data and/or micro-benchmarks.
- Runtime performance of kernel gives FLOP/s, analysis of bytes/FLOPS gives CI.
- Kernel A is compute bound; Kernel B is memory bandwidth bound.
- Both kernels require optimisation!
- If kernel is on the roof, then rate limiting factor is achieved.

Intel Advisor: Roofline

- Can use Intel Advisor to run a Roofline analysis on your code.
- First, it runs some micro-benchmarks to generate the Roofline model.
- Then, it runs your code, calculating the CI and performance.
- Can be helpful to visualise how performant your code is.
- More information: `https://software.intel.com/en-us/articles/intel-advisor-roofline`.
- But beware, the CI is calculated from executed instructions, so not always the whole picture.

Vectorisation

$$C = A + B$$



Why vectorise?

- Vectorisation gives you more compute per cycle.
- Hence may increase the FLOP/s rate of the processor.
- Also results in fewer instructions to process (less pressure on instruction decode units).
- Vectors help make good use of the memory hierarchy (often the main benefit).
- Vectorisation helps you write code which has good access patterns to maximise bandwidth.

Auto-vectorisation

- Modern compilers are very good at automatically vectorising your loops.
- Fortran helps as arrays can not alias (overlap), unlike C.
- But compiler needs to be sure it's safe to vectorise.
- Read compiler reports to see if it's already vectorising.
  - Intel: `-qopt-report=5`
  - Cray: `-hlist=a`
  - GNU (old): `-ftree-vectorizer-verbose=2`
  - GNU (new): `-fopt-info-vec`
  - Clang: `-Rpass=loop-vectorize`
    `-Rpass-missed=loop-vectorize`
    `-Rpass-analysis=loop-vectorize`
- Often the memory access pattern prevents (efficient) auto-vectorisation.

OpenMP SIMD

- Sometimes the compiler needs help in confirming loops are vectorisable.
- OpenMP `simd` constructs give this information.
- Can combine with `parallel do` construct to ensure a parallel vector loop: `omp parallel do simd`
- Generally want to vectorise inner loops and parallelise outer loops.

```
!$omp simd
do i = 1, N
  C(i) = A(i)+B(i)
end do
!$omp end simd
```

SIMD functions

Say you've written an update function to update values in the loop:

```
do i = 1, N
  A(i) = magic_maths(A(i))
end do
```

- The situation gets complicated.
- If the function is small, then likely inlined and loop will auto-vectorise.
- Otherwise need to use the `simd` construct, but need compiler to create a vector version of the function.

```
function magic_maths(value) result(r)
!$omp declare simd(magic_maths)
  implicit none
  real(kind=8) :: value, r
  r = value * value
end function
```

SIMD clauses

- All the usual data-sharing and reduction clauses can be applied.
- `safelen(4)`: distance between iterations where its safe to vectorise.

```
!$omp simd safelen(4)
do i = 1, N-4
  A(i) = A(i) + A(i+4)
end do
!$omp end simd
```

- `simdlen(4)`: preferred iterations to be performed concurrently as a vector. Specifying explicit vector lengths builds in obsolescence to the code as hardware vector lenghts continually change — don't recommend using this clause.

SIMD clauses

- `linear(var)`: variable is private and linear to the loop iterator.

```
!$omp simd linear(j)
do i = 1, N
  j = j + 1
  A(j) = B(i)
end do
!$omp end simd
```

- `aligned(var)`: says the array is aligned (more on this shortly).
- `uniform(var)`: for `declare simd` construct, the variable is the same in all vector lanes.

SIMD summary

- Sometimes need to force the compiler to auto-vectorise (the correct) loop with the `simd` construct.
- As with `parallel`, you are telling the compiler it is safe to vectorise and to ignore its data dependancy analysis.
- Check the compiler report before and after the check it did the right thing!
- Use `declare simd` and appropritae clauses if you need to create vectorised versions of functions.
  - The clauses can give more information to the compiler so it does a better job.

Derived types

2D grid of cells, each cell containing 4 different values.

```fortran
type cell
  real(kind=8) :: property1
  real(kind=8) :: property2
  real(kind=8) :: property3
  real(kind=8) :: property4
end type

type(cell), allocatable :: grid(:,:)

do j = 1, ny
  do i = 1, nx
    grid(i,j)%property1 = update_1()
    grid(i,j)%property2 = update_2()
    grid(i,j)%property3 = update_3()
    grid(i,j)%property4 = update_4()
  end do
end do
```

Derived types

- What do Fortran derived types look like in memory?
- Organised as an array of structures.

P1	P2	P3	P4	P1	P2	P3	P4	P1	P2	P3	P4	P1

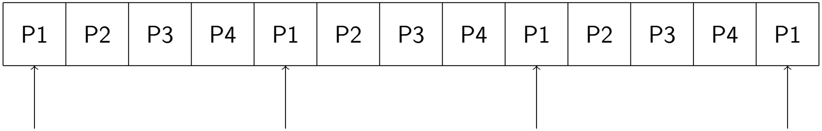
Derived types

- What do Fortran derived types look like in memory?
- Organised as an array of structures.
- What happens when we vectorise our loop over cells?

P1	P2	P3	P4	P1	P2	P3	P4	P1	P2	P3	P4	P1

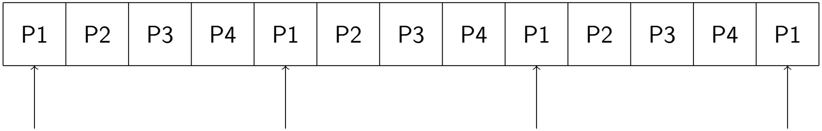
Derived types

- What do Fortran derived types look like in memory?
- Organised as an array of structures.
- What happens when we vectorise our loop over cells?



Derived types

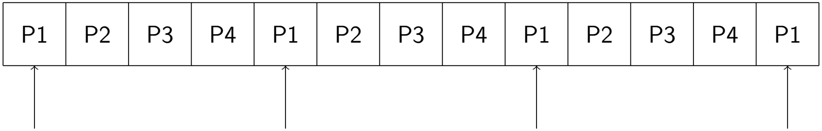
- What do Fortran derived types look like in memory?
- Organised as an array of structures.
- What happens when we vectorise our loop over cells?



- The `property1` values are gathered into a vector register.

Derived types

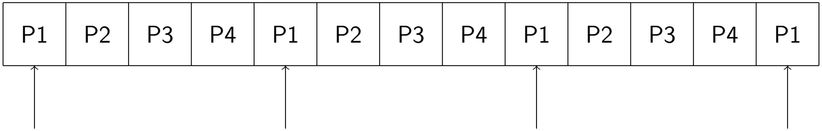
- What do Fortran derived types look like in memory?
- Organised as an array of structures.
- What happens when we vectorise our loop over cells?



- The `property1` values are gathered into a vector register.
- After the computation, the results are scattered back into memory.

Derived types

- What do Fortran derived types look like in memory?
- Organised as an array of structures.
- What happens when we vectorise our loop over cells?



- The `property1` values are gathered into a vector register.
- After the computation, the results are scattered back into memory.
- A cache line is 64 bytes, so only the first two values are on the first cache line.
- Must read two cache lines to fill the vector up.

Structure of arrays

Switch type around to have an array per property.

```
type grid
  real(kind=8), allocatable :: property1(:,:)
  real(kind=8), allocatable :: property2(:,:)
  real(kind=8), allocatable :: property3(:,:)
  real(kind=8), allocatable :: property4(:,:)
end type

do j = 1, ny
  do i = 1, nx
    grid%property1(i,j) = update_1()
    grid%property2(i,j) = update_2()
    grid%property3(i,j) = update_3()
    grid%property4(i,j) = update_4()
  end do
end do
```

Structure of arrays

- Order of data in memory has changed.

P1	P1	P1	P1	P1	···	P2	P2	P2	P2	P2	···	P3

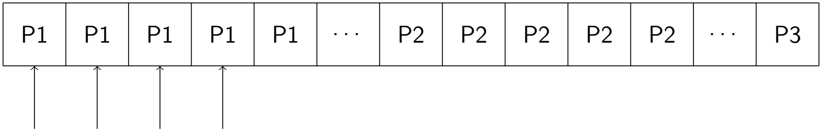
Structure of arrays

- Order of data in memory has changed.
- What happens when we vectorise?

P1	P1	P1	P1	P1	$\cdots$	P2	P2	P2	P2	P2	$\cdots$	P3

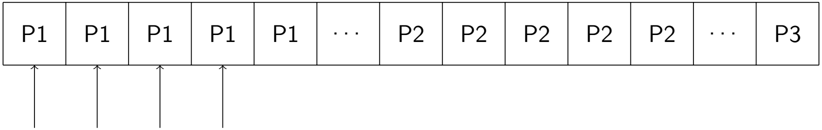
Structure of arrays

- Order of data in memory has changed.
- What happens when we vectorise?



Structure of arrays

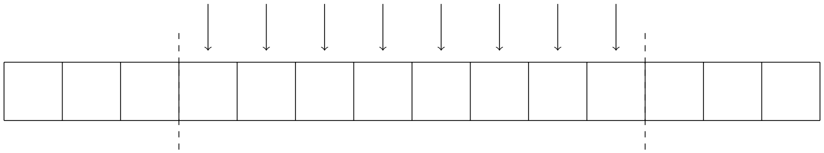
- Order of data in memory has changed.
- What happens when we vectorise?



- Coalesced memory accesses are key for high performance code.
- Adjacent vector lanes read adjacent memory locations.
- A cache line is 64 bytes, so can fill the vector from a single cache line.
- More efficient vectorisation.

Memory access patterns

```
do i = 1, N
  val = A(i)
end do
```

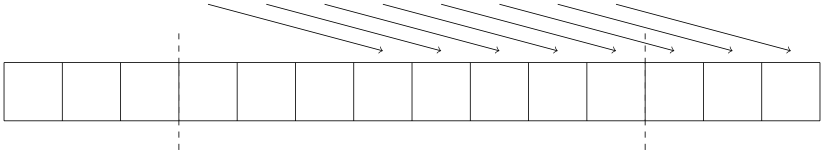


64 byte boundary

- Ideal memory access pattern.
- All access is coalesced.
- Vectors are aligned to cache line boundary.

Memory access patterns

```
do i = 1, N
  val = A(i+3)
end do
```

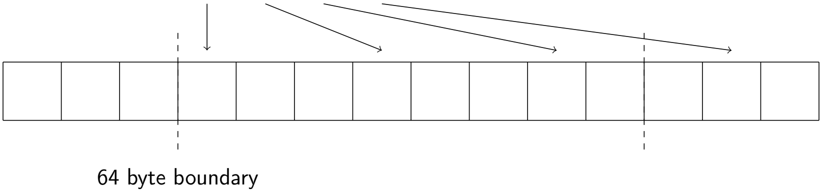


64 byte boundary

- OK memory access pattern.
- All access is coalesced, but split across cache lines.
- Still get good use of cache lines, but not as efficient as aligned version.

Memory access patterns

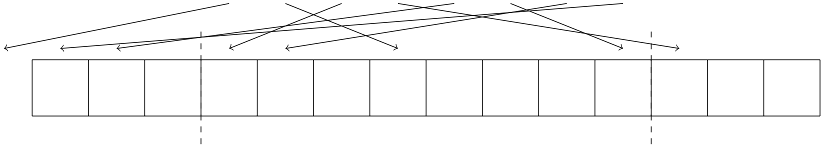
```
do i = 1, N
  val = A(j,i) ! equiv. A(j+3*i)
end do
```



- Strided access results in multiple memory transactions.
- Kills throughput due to poor reuse of cached data.
- Very easy to fall into this trap with multi-dimensional arrays.
- Check your strides!

Memory access patterns

```
do i = 1, N
  val = A(B(i))
end do
```



64 byte boundary

- Essentially random access to memory.
- Little reuse of cache lines.
- Unpredictable pattern, so hardware prefetchers won't work efficiently.
- Very challenging!

Alignment

- If we can align arrays, we get better vectorisation; specifically load/stores are faster.
  - Guarantee only one cache line needs updating and not split between two cache lines.
- Taking advantage of alignment is a two stage process:
  1. Align the memory on allocation.
  2. Tell the compiler the access is aligned.
- Aligned allocations in Fortran are (currently) unfortunately vendor specific.
- OpenMP can help with telling the compiler the data is aligned.
- Aligned allocations due in OpenMP 5.0.

Step 1: Aligning allocations

Generally focus on the Intel compiler. Only need to use one of these methods, whichever is most convenient.

- Align all allocations of arrays (not in derived types) with compiler flag: `-align array64byte`
- Use an Intel compiler directive on array definition:

```
real(kind=8), allocatable :: A(:,:)
!dir$ attributes align:64 :: A
```

- Allocate memory in C, and convert to Fortran `pointer`:

```
double * alloc(int *len) {
  return (double *)aligned_alloc(64,
   ↪  sizeof(double)*(*len));
}
```

```
real(kind=8), pointer :: A(:,:)
type(c_ptr) :: A_ptr
A_ptr = alloc(nx*ny)
call c_f_pointer(A_ptr, A, (/ nx, ny/))
```

Step 2: Telling the compiler

- Use OpenMP `simd aligned` clause:

```
  !$omp simd aligned(A:64)
  do i = 1, nx
    A(i,j) = A(i,j) + 1.0
  end do
  !$omp end simd
```

Step 2: Telling the compiler

- Use OpenMP `simd aligned` clause:

```
!$omp simd aligned(A:64)
do i = 1, nx
  A(i,j) = A(i,j) + 1.0
end do
!$omp end simd
```

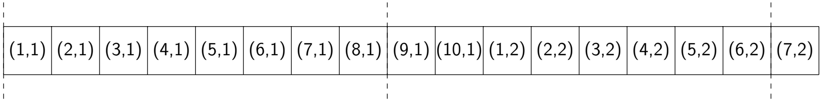
- Unfortunately often not sufficient.
- Often need to use Intel specific directives to say loop extent is divisible by vector length.

```
! 64 byte aligned / 8 byte data type means mod 8
!dir$ assume(mod(nx,8) .eq. 0)
!$omp simd aligned(A:64)
do i = 1, nx
  A(i,j) = A(i,j) + 1.0
end do
!$omp end simd
```

- Check the compiler report for aligned and unaligned access.

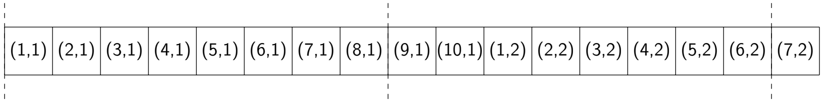
Aligning 2D arrays

- Aligning the memory only aligns the first entry.
- Multiples of the alignment factor will also be aligned.
- With 2D arrays you need to double check that access can be aligned.
- Example: 10-by-10 grid of FP64 numbers, aligned to 64 byte cache line:



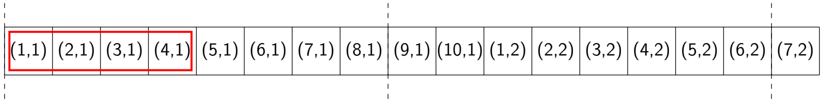
```
do j = 1, 10
  !$omp simd aligned(A:64)
  do i = 1, 10
    A(i,j) = A(i,j) + ...
  end do
  !$omp end simd
end do
```

Aligning 2D arrays



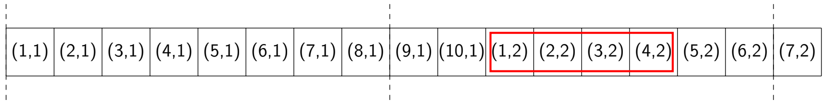
- The array is aligned to a 64-byte cache line.

Aligning 2D arrays



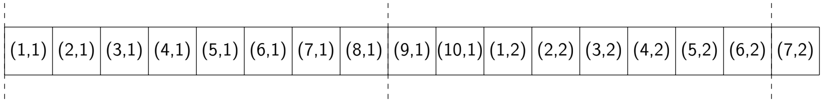
- The array is aligned to a 64-byte cache line.
- Accessing the vector `A(1:4,1)` is aligned.

Aligning 2D arrays



- The array is aligned to a 64-byte cache line.
- Accessing the vector `A(1:4,1)` is aligned.
- Accessing the vector `A(1:4,2)` is *not* aligned.

Aligning 2D arrays



- The array is aligned to a 64-byte cache line.
- Accessing the vector `A(1:4,1)` is aligned.
- Accessing the vector `A(1:4,2)` is *not* aligned.

- Need the inner stride to be a multiple of the alignment, and need to tell the compiler this is true (previous slide).
- Solution: pad the array, but beware of memory footprint.
- Example of why the `aligned` clause doesn't always ensure aligned load/stores.

Branches

- CPUs support speculative execution, GPUs tend not to.
- Branch instructions have high latency.
- GPUs hide this latency by fast context switching, CPUs by good branch predictors.
- In both cases, divergent execution within the vector unit reduces performance.
- Can use predication, selection and masking to convert conditional control flow into straight line code.

Removing branches

Conditional execution

- Only evaluate expression if condition is met

```
if (a .gt. b) then
  acc = acc + (a - b*c)
end if
```

```
if (a > b)
  acc += a - b*c;
```

Selection and masking

- Always evaluate expression and mask result

```
temp = a - b*c
mask = merge(1.0, 0.0, a
↪  .gt. b)
acc = acc + (mask * temp)
```

```
temp = a - b*c;
mask = a > b ? 1.0 : 0.0;
acc += mask * temp;
```

In practice, you may or may not see an improvement: the compiler may be doing something smart already.

Exercise

- Take your parallel 5-point stencil code and optimise it.
- Think about:
  - Memory access patterns
  - Vectorisation
- Note down the performance differences your optimisations make.
- Calculate the achieved memory bandwidth of your stencil code.
- Extension: consider these optimisaions for the Jacobi solver.

Summary

- Performance of cache hierarchy.
- Performance analysis with the Roofline model.
- Vectorisation:
  - Compiler auto-vectorisation.
  - OpenMP `simd` construct.
  - Memory access patterns.
  - Data alignment.

- Next sessions:
  4. NUMA and MPI interoperability.
  5. GPU programming with OpenMP.
  6. Tasks and Tools.