High Performance Linear Algebra

Lecture 3: Intra-node parallelism and starting with BLAS

Ph.D. program in High Performance Scientific Computing

Fabio Durastante **Pasqua D'Ambra** **Salvatore Filippone**

November 17, 2025 — 14.00:16.00

Dipartimento di Matematica
Università di Pisa

Summary of previous lecture

- Taxonomy of computer architectures
- Performance metrics: FLOP/s, speedup, efficiency, scalability
- Performance modeling: weak and strong scaling

Table of Contents

2 The roofline model

Modern Memory Hierarchy

- Computer architectures organized around a **memory hierarchy**
- Designed to balance speed, capacity, and cost

Memory Hierarchy Levels

1. Registers and cache (L1, L2, L3) — extremely fast
2. Main memory (RAM) — moderate speed
3. Secondary storage (SSD/HDD) — slower
4. Tertiary storage — archival

Key parameter: Memory bandwidth — rate of data transfer between memory and processor

The Memory Wall

The Problem

Processor speeds have grown much faster than memory bandwidth improvements

- **Memory wall:** memory latency and bandwidth become the primary bottleneck
- Need tools to understand and visualize this limitation
- Enter: the *Roofline Model* [7]

The Roofline Model: Concept

Definition

A visual performance model relating computational throughput to memory bandwidth

Key hardware characteristics:

- Peak floating-point performance: Perf (FLOP/s)
- Peak memory bandwidth: BW (Bytes/s)

Key application characteristic:

- Operational Intensity (OI): FLOP/Byte
- Ratio of floating-point ops to bytes accessed from memory

Roofline Model: The Relationship

Fundamental equation

$$\text{Perf} = \frac{\text{FLOP}}{s} = \frac{\text{FLOP}}{\text{Byte}} \cdot \frac{\text{Byte}}{s} = \text{OI} \cdot \text{BW}$$

- Performance depends linearly on both OI and BW
- Plotted as log-log graph: performance vs operational intensity
- Creates a characteristic “roofline” shape

Roofline Model: Visual Representation

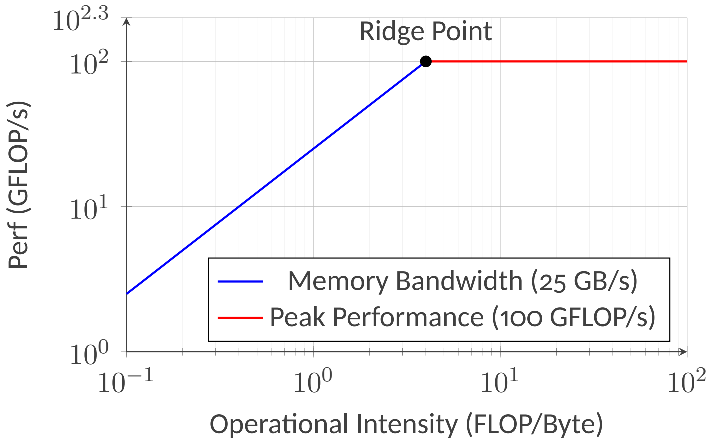


Figure: Roofline model: 100 GFLOP/s peak, 25 GB/s bandwidth

Understanding the Roofline Plot

Two regions:

1. Memory-bound (left)
   - Linear increase with OI
   - Limited by bandwidth
2. Compute-bound (right)
   - Horizontal line
   - Limited by peak FLOP/s

Ridge point:

- Intersection of two regions
- Minimum OI to reach peak performance
- In example: 4 FLOP/Byte

Using the Roofline Model

Applications:

- Analyze kernel performance on given architecture
- Identify performance bottlenecks
- Guide optimization efforts

Optimization strategy

Compare kernel’s OI to ridge point:

- Below ridge → memory-bound → improve data locality
- Above ridge → compute-bound → optimize computations

Roofline and Linear Algebra Evolution

- Algorithmic optimization improves OI and data locality
- Example: Evolution of BLAS (Basic Linear Algebra Subprograms)
  - Level 1: vector operations (low OI)
  - Level 2: matrix-vector operations (medium OI)
  - Level 3: matrix-matrix operations (high OI)
- Higher-level BLAS operations:
  - Reuse data in fast memory
  - Reduce memory traffic
  - Approach compute-bound regime

More details on BLAS in upcoming lectures

Measuring Memory Bandwidth: STREAM

STREAM Benchmark [4, 5]

Measures sustainable memory bandwidth (GB/s) for simple vector kernels

Four kernels:

COPY	Copy vector from one location to another
SCALE	Scale vector by constant factor
SUM	Add two vectors
TRIAD	Scaled vector addition

- Simple, easy to understand
- Provides reliable bandwidth measure
- Widely used in HPC community

Info: `http://www.cs.virginia.edu/stream/`

Measuring Memory Bandwidth: Example

Let us run try the STREAM benchmark on your machine:

- Download the STREAM benchmark from `http://www.cs.virginia.edu/stream/`

```
mkdir -p stream && cd stream
wget -r -np -nH --cut-dirs=2 -e robots=off -R "index.html*" \
    https://www.cs.virginia.edu/stream/FTP/Code/
```

- There is a `Makefile` provided; you can compile with `make`
- The standard configuration requires `g77`, but you can edit the `Makefile` to use `gfortran`, or any other compiler you have available:

```
FF = gfortran
FFLAGS = -O2
```

- Run the benchmark by doing: `./stream_f.exe`

Example output of STREAM benchmark

```
-------------------------------------------------------------
 Double precision appears to have 16 digits of accuracy
 Assuming 8 bytes per DOUBLE PRECISION word
-------------------------------------------------------------
 -------------------------------------------------------------
 STREAM Version $Revision: 5.6 $
 -------------------------------------------------------------
 Array size =    2000000
 Offset     =          0
 The total memory requirement is   45 MB
 You are running each test  10 times
 --
 The *best* time for each test is used
 *EXCLUDING* the first and last iterations
 -------------------------------------------------------------
```

Example output of STREAM benchmark

```
 ----------------------------------------------
 Printing one line per active thread....
 ----------------------------------------------------
 Your clock granularity/precision appears to be      1 microseconds
 ----------------------------------------------------
Function      Rate (MB/s)  Avg time    Min time   Max time
Copy:        19300.7949     0.0017      0.0017      0.0019
Scale:       16737.4645     0.0019      0.0019      0.0020
Add:         20691.3250     0.0024      0.0023      0.0025
Triad:       19599.5514     0.0025      0.0024      0.0025
 ----------------------------------------------------
 Solution Validates!
 ----------------------------------------------------
```

How to obtain correct results from STREAM

- Ensure the array size is large enough to exceed cache sizes
- Compile with optimizations enabled (e.g., `-O2` or higher)
- Run multiple iterations and take the best time
- Validate results to ensure correctness

Note

Reported bandwidth may vary based on system load, compiler optimizations, and other factors. Always run multiple trials for reliable measurements.

How to obtain correct results from STREAM: Example

We can extract the right way to perform the test by looking at the size of the level 3 cache of our machine and ensuring that the array size is large enough to exceed it. This number can be found by running the command:

```
lscpu | grep "L3"
```

On my machine, this returns:

```
L3 cache:                              36 MiB (1 instance)
```

So I should set the array size to be larger than 36 MiB. Since each double-precision number takes 8 bytes, I can calculate the minimum number of elements needed:

```
MIN_SIZE=$(echo "36 * 1024 * 1024 / 8" | bc)
echo $MIN_SIZE
```

This gives me 4,718,592 elements. To be safe, I can set the array size to 5,000,000 elements in the STREAM benchmark code before compiling and running it

How to obtain correct results from STREAM: Modifying the Makefile

Using awk

A nice way to automate the modification of the array size in the STREAM benchmark code is to use `awk` to edit the source file directly from the command line.

```
L3CACHE=$(lscpu | awk -F: '/L3 cache/ {match($2, /[0-9]+/); print
↪  substr($2, RSTART, RLENGTH)}')
MIN_SIZE=$(echo "${L3CACHE} * 1024 * 1024 / 8" | bc)
echo $MIN_SIZE
```

Then, you can modify the `FFLAGS` variable in the Makefile to use the new array size:

```
FFLAGS="-O3 -march=native -mtune=native
↪  -DSTREAM_ARRAY_SIZE=${MIN_SIZE}"
```

Measuring Peak Performance

2 The roofline model

Estimation formula

$$\text{Peak FLOP/s} = \text{Cores} \times \text{Clock (GHz)} \times \text{FLOP/Cycle}$$

Example: x86 processor with AVX2

- 8 double-precision FLOP per cycle
- 4 cores at 3 GHz
- Peak: $4 \times 3 \times 8 = 96$ GFLOP/s

Note

This is theoretical peak; actual performance may be lower due to: bandwidth limitations, cache misses, other overheads. It always best to get this number from the manufacturer datasheet when possible.

Table of Contents

3 Intra-node parallelism

From Moore’s Law to Parallelism

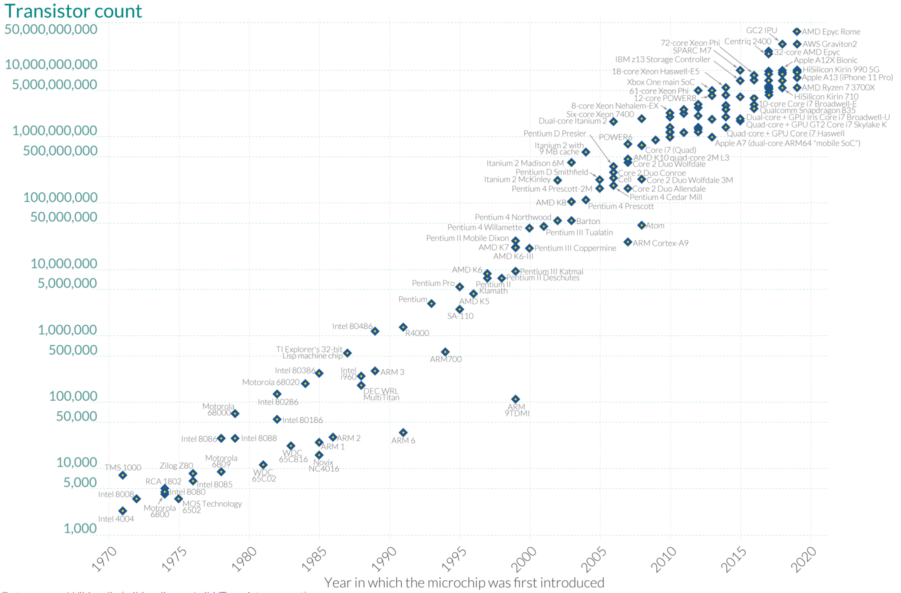
- For decades, performance grew via Moore’s Law
  - Higher clock frequencies
  - Instruction Level Parallelism (ILP): pipelining, out-of-order execution, branch prediction
- Early 2000s: this trend hit fundamental limits

Moore’s Law

Number of transistors on a microchip doubles approximately every two years, leading to increased computational power and decreased relative cost (Gordon E. Moore, 1965)

Moore's Law: The number of transistors on microchips has doubled every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

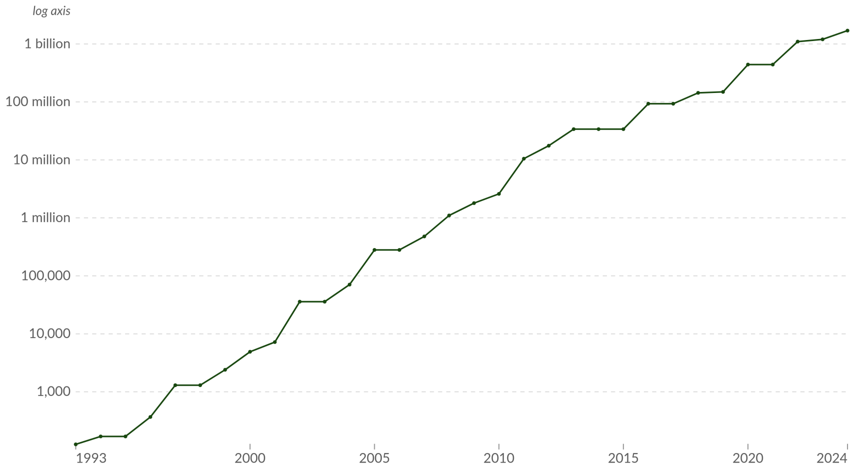


Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

Computational capacity of the fastest supercomputers

The number of floating-point operations[1] carried out per second by the fastest supercomputer in any given year. This is expressed in gigaFLOPS, equivalent to 10^9 floating-point operations per second.



Data source: Dongarra et al. (2024)

OurWorldinData.org/technological-change | CC BY

1. **Floating-point operation** A floating-point operation (FLOP) is a type of computer operation. One FLOP represents a single arithmetic operation involving floating-point numbers, such as addition, subtraction, multiplication, or division.

Hard Limits to ILP

Concurrence Limit

- ILP techniques are sophisticated but limited
- Modern processors: max 4-5 instructions per cycle
- Available concurrence is much larger

Power Limit

- Power consumption $\propto$ frequency^3
- Critical for mobile devices (battery life)
- Critical for supercomputers (operational costs)
- Top500 systems: $\sim$30 MW (small town!)

The Shift to Thread Level Parallelism (TLP)

- Industry shifted from ILP to TLP techniques
- Birth of **multicores** / **Chip Multi-Processors (CMP)**
- Multiple independent cores on the same die
- Each core handles different instructions and data streams

Key Advantages

- Higher concurrence levels
- Power consumption $\propto$ number of cores (linear)
- Lower frequency + more cores = better performance + less power

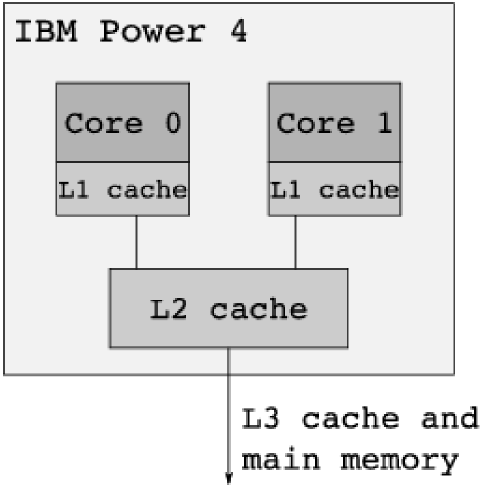
The Multicore Era

- Multicore processors are now ubiquitous
- Evolution driven by increasing core counts per chip
- Paradigm shift: parallel programming is essential

**Performance no longer comes from faster cores,
but from more cores working together**

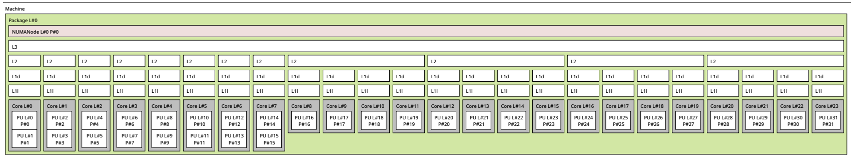
POWER4: first mainstream multicore (2001)

- Two general-purpose cores on the same die
- Per-core private L1 caches
- Shared L2; off-chip shared L3
- Cores access DRAM via a shared memory bus
- Template for many subsequent multicore designs



Modern CPU examples

- AMD EPYC 9655P (2023): 96 cores, 192 threads
- Intel Xeon w9-3595X (2024): 60 cores, 120 threads
- Intel i9-14900HX (hybrid, 24 cores / 32 threads)
  - 8 P-cores (16 threads), each with dedicated L2
  - 16 E-cores (16 threads), L2 shared across clusters of 4
  - L3 shared among all cores; L1 private per core



CPU cache hierarchy (L1, L2, L3)

- Caches use SRAM (fast, low latency, small, costly)
- DRAM in main memory is larger but slower
- Multi-level design balances speed, capacity, and cost
- L1: smallest/fastest, usually split I/D caches, per-core
- L2: larger/slower than L1, per-core or per-cluster
- L3: largest on-chip, shared across cores
- Miss path: L1 → L2 → L3 → DRAM (increasing latency)

Exercise: topology

Use `lscpu` and the following command to inspect your CPU topology:

```
lstopo --no-attrs --no-factorize --no-collapse --no-cpukinds --no-legend
↪  topology.pdf
```

Memory-bound vs compute-bound workloads

Memory-bound

- Low arithmetic intensity; little/no data reuse
- Performance limited by memory bandwidth
- Parallel speedups saturate early
- Examples:
  - SpMV: $\mathcal{O}(\mathrm{nnz})$ FLOPs on $\mathcal{O}(\mathrm{nnz})$ data
  - BLAS-1: $\mathcal{O}(n)$ FLOPs on $\mathcal{O}(n)$ data
  - BLAS-2: $\mathcal{O}(n^2)$ FLOPs on $\mathcal{O}(n^2)$ data

Compute-bound

- High arithmetic intensity; strong data reuse
- Performance limited by peak FLOP/s
- Scales well across cores (cache-friendly)
- Examples:
  - BLAS-3 (e.g., GEMM): $\mathcal{O}(n^3)$ FLOPs on $\mathcal{O}(n^2)$ data
  - Dense factorizations leveraging BLAS-3

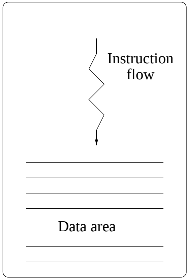
Operational intensity and the memory wall

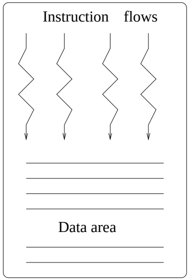
- Example (EPYC 9655P): peak 710 GFLOP/s vs 614 GB/s bandwidth
- Roofline knee: 710/614 ≈ 1.16 FLOP/byte
  - OI < 1.16: memory-bound (bandwidth limits performance)
  - OI > 1.16: compute-bound (FLOP/s limits performance)
- As core counts grow, static bandwidth limits memory-heavy kernels
- Remedies: improve cache reuse, increase bandwidth, or both

Multiprogramming and processes

- Modern systems support multiprogramming: many programs appear to run concurrently.
- Microscopic view: you **cannot** execute **more programs than available cores**.
- Macroscopic view: time sharing makes many programs seem concurrent.
- A process is a running *instance of a program* **plus** its *data*.
- Processes are dynamic; multiple processes can run the same program.
- **Each process has a private address space** (its data are private).

Processes and threads: visual





- Code + private memory + execution context.
- OS schedules processes on cores.
- No shared memory by default.

- Execution streams within a process.
- Share address space and program data.
- Own stack and registers; often one per core.

Programming tools for threads

- POSIX threads (pthreads): low-level API, fine-grained control, portable.
- OpenMP: high-level, directive-based, widely used in C/C++/Fortran.
- Typical workflow: start with OpenMP; use pthreads only when necessary.

Will start describing some **OpenMP basics**, and decline it in the context of linear algebra routines.

OpenMP: overview

- De-facto standard API for shared-memory parallel programming.
- Languages: Fortran, C, C++; introduced in 1997.
- Maintained by the OpenMP Architecture Review Board (openmp.org).

OpenMP: overview

- De-facto standard API for shared-memory parallel programming.
- Languages: Fortran, C, C++; introduced in 1997.
- Maintained by the OpenMP Architecture Review Board (openmp.org).

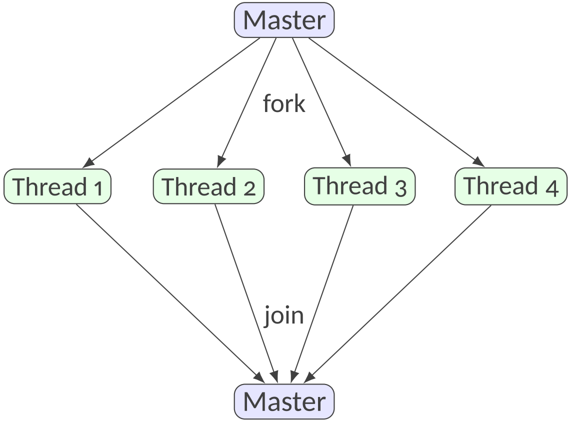
Components:
- Compiler directives (pragmas)
- Run-time library routines
- Environment variables

Directives behave as:

1. Actual instructions for OpenMP-aware compilers
2. Comments for non-supporting compilers (keeps serial behavior)

Fork-join execution model

- Serial region executed by a single master thread.
- Hitting a parallel region: fork into multiple threads.
- Threads share address space; may coordinate via shared data.
- End of region: threads join back to one thread.



Original focus: loop parallelism

- Split workload of loops (e.g., `do`) across threads.

1. Enter a loop/region: activate multiple threads and partition iterations.
2. Threads may communicate via shared variables/memory.
3. On completion: synchronize; deactivate all but one thread and continue serially.

Original focus: loop parallelism

- Split workload of loops (e.g., `do`) across threads.

1. Enter a loop/region: activate multiple threads and partition iterations.
2. Threads may communicate via shared variables/memory.
3. On completion: synchronize; deactivate all but one thread and continue serially.

- Programming model: threads with shared logical address space.
- Natural fit for shared-memory systems; not mandated by the standard.
- Attempts to map the same model to distributed-memory exist, but limited success in practice.

OpenMP today

- Standard evolves regularly; 6.0 recently released, 5.2 widely supported.
- Key additions:
  - Irregular and data-driven workload dispatching
  - Transformations to improve memory hierarchy usage and work sharing
  - Support for SIMD extensions and accelerators

OpenMP in practice

- Will show concrete OpenMP code next.
- Often combined with MPI for hybrid/nested parallelism.
- Further reading: [1, 2, 3, 6]

OpenMP example: let us start from an hello world

The standard Fortran hello world program:

```fortran
program hello
    use, intrinsic :: iso_fortran_env,
    ↪  only: output_unit
    write (output_unit, '("Hello,
    ↪  world!")')
end program hello
```

which can be compiled and run as:

```
gfortran -o hello hello.f90
./hello
```

Getting the output:

```
Hello, world!
```

OpenMP example: let us start from an hello world

The standard Fortran hello world program:

```
program hello
    use, intrinsic :: iso_fortran_env,
    ↪   only: output_unit
    write (output_unit, '("Hello,
    ↪   world!")')
end program hello
```

which can be compiled and run as:

```
gfortran -o hello hello.f90
./hello
```

Getting the output:

```
Hello, world!
```

We now want to implement the same program using OpenMP, and getting an output from each thread.

```
program hello
    use, intrinsic :: iso_fortran_env,
    ↪   only: output_unit
    use omp_lib
    integer :: tid, nthreads
    nthreads = omp_get_max_threads()
    !$omp parallel private(tid)
    tid = omp_get_thread_num()
    write (output_unit, '("Hello, world!
    ↪   from thread ", I0)') tid
    !$omp end parallel
end program hello
```

Compiling the OpenMP hello world

To compile the OpenMP version, we need to add the '-fopenmp' flag:

```
gfortran -o hello hello.f90 -fopenmp
./hello
```

Getting the output (on my Laptop):

```
Hello, world! from thread 3
Hello, world! from thread 20
Hello, world! from thread 31
Hello, world! from thread 1
Hello, world! from thread 2
Hello, world! from thread 5
Hello, world! from thread 7
⋮
Hello, world! from thread 29
Hello, world! from thread 16
```

- Each thread prints its ID.
- Order of output may vary due to thread scheduling.
- By default, uses all available threads.
- Control number of threads via `OMP_NUM_THREADS=<num>` environment variable.

Compiling the OpenMP hello world

To compile the OpenMP version, we need to add the ‘-fopenmp‘ flag:

```
gfortran -o hello hello.f90 -fopenmp
./hello
```

Getting the output (on my Laptop):

```
Hello, world! from thread 3
Hello, world! from thread 20
Hello, world! from thread 31
Hello, world! from thread 1
Hello, world! from thread 2
Hello, world! from thread 5
Hello, world! from thread 7
⋮
Hello, world! from thread 29
Hello, world! from thread 16
```

- Each thread prints its ID.
- Order of output may vary due to thread scheduling.
- By default, uses all available threads.
- Control number of threads via `OMP_NUM_THREADS=<num>` environment variable.
- Let us have a better look at the code, line by line.

OpenMP hello world: code walkthrough

```
program hello
    use, intrinsic ::
    ↪   iso_fortran_env, only:
    ↪   output_unit
    use omp_lib
    integer :: tid, nthreads
    nthreads =
    ↪   omp_get_max_threads()
    !$omp parallel private(tid)
    tid = omp_get_thread_num()
    write (output_unit, '("Hello,
    ↪   world! from thread ",
    ↪   I0)') tid
    !$omp end parallel
end program hello
```

- `use omp_lib`: imports OpenMP functions/constants
- `nthreads = omp_get_max_threads()`: gets max available threads
- *`!$omp parallel private(tid)`*: starts parallel region; each thread has private `tid`
- `tid = omp_get_thread_num()`: each thread gets its unique ID
- *`!$omp end parallel`*: ends parallel region; threads synchronize

Compilation flag for other compilers

- **GCC / GFortran**: `-fopenmp`
- **Intel ICC / IFORT**: `-qopenmp` or `-openmp`
- **Clang / Flang**: `-fopenmp` (requires OpenMP library)
- **PGI / NVIDIA HPC SDK**: `-mp`

Note

Ensure the compiler supports OpenMP and is properly configured.

Mixing compilers

There exist a few cases where mixing compilers is possible (e.g., Intel and GCC), but in general it is not recommended to mix different compilers when dealing with OpenMP code.

Using CMake to build a Fortran project

As we have seen from the previous slide, and from the question on managing different compilers in the previous lecture, it is often useful to use a **build system** to manage the complexity of building a project.

Using CMake to build a Fortran project

As we have seen from the previous slide, and from the question on managing different compilers in the previous lecture, it is often useful to use a **build system** to manage the complexity of building a project.
There exists several build systems:

Make / GNU Make / Autotools:	classic, widely used, but low-level
	🔗 https://www.gnu.org/software/make/
CMake:	popular, cross-platform, higher-level
	🔗 https://cmake.org/
Ninja:	fast, modern, often used as a backend for CMake
	🔗 https://ninja-build.org/
Meson:	high-level, fast, modern
	🔗 https://mesonbuild.com/

Using CMake to build a Fortran project

To build a project with CMake, the first step is represented by the creation of a `CMakeLists.txt` file in the root directory of the project.

Using CMake to build a Fortran project

To build a project with CMake, the first step is represented by the creation of a `CMakeLists.txt` file in the root directory of the project.
Let us go step by step through a minimal example.

1. Create a folder for the project and enter it:

```
mkdir hello_openmp
cd hello_openmp
```

Using CMake to build a Fortran project

To build a project with CMake, the first step is represented by the creation of a `CMakeLists.txt` file in the root directory of the project.
Let us go step by step through a minimal example.

1. Create a folder for the project and enter it:

   ```
   mkdir hello_openmp
   cd hello_openmp
   ```

2. Create a git repository inside:

   ```
   git init
   git branch -m main
   ```

Using CMake to build a Fortran project

To build a project with CMake, the first step is represented by the creation of a `CMakeLists.txt` file in the root directory of the project.
Let us go step by step through a minimal example.

1. Create a folder for the project and enter it:
   ```
   mkdir hello_openmp
   cd hello_openmp
   ```
2. Create a git repository inside:
   ```
   git init
   git branch -m main
   ```
3. Create the Fortran source file `hello.f90` with the OpenMP code seen before.

Using CMake to build a Fortran project

To build a project with CMake, the first step is represented by the creation of a `CMakeLists.txt` file in the root directory of the project.
Let us go step by step through a minimal example.

1. Create a folder for the project and enter it:

   ```
   mkdir hello_openmp
   cd hello_openmp
   ```

2. Create a git repository inside:

   ```
   git init
   git branch -m main
   ```

3. Create the Fortran source file `hello.f90` with the OpenMP code seen before.
4. Create the `CMakeLists.txt` file:

   ```
   touch CMakeLists.txt
   ```

Editing the CMakeLists.txt file

The content of the `CMakeLists.txt` file should be as follows:

```
cmake_minimum_required(VERSION 3.23)

project(hello-openmp LANGUAGES Fortran)
find_package(OpenMP REQUIRED COMPONENTS
↪   Fortran)

# Executable from the single source file
add_executable(hello-openmp hello-openmp.f90)
# Link OpenMP
target_link_libraries(hello-openmp PRIVATE
↪   OpenMP::OpenMP_Fortran)
```

- Specify minimum CMake version
- Define project name and language
- Find OpenMP package for Fortran
- Add executable target
- Link OpenMP libraries to the target

The CMake instructions explained

The roject name and the programming language used, it also take further optional arguments:

```
project(<PROJECT-NAME>
        [VERSION
        ↪   <major>[.<minor>[.<patch>[.<tweak>]]]]
        [COMPAT_VERSION
        ↪   <major>[.<minor>[.<patch>[.<tweak>]]]]
        [SPDX_LICENSE <license-string>]
        [DESCRIPTION <description-string>]
        [HOMEPAGE_URL <url-string>]
        [LANGUAGES <language-name>...])
```

Specify:

- project name
- version
- compatible version
- license (SPDX format)
- description
- homepage URL
- programming languages used

The CMake instructions explained

Another important command is igure external packages or libraries that the project depends on.

```
find_package(<PackageName> [version] [EXACT]
↪  [REQUIRED]
             [QUIET] [COMPONENTS components...]
             [OPTIONAL_COMPONENTS components...]
             [NO_DEFAULT_PATH])
```

Specify:

- package name
- version
- whether it is required
- components to find
- whether to suppress messages
- whether to avoid default search paths

The CMake instructions explained

Another important command is igure external packages or libraries that the project depends on.

```
find_package(<PackageName> [version] [EXACT]
↪  [REQUIRED]
             [QUIET] [COMPONENTS components...]
             [OPTIONAL_COMPONENTS components...]
             [NO_DEFAULT_PATH])
```

You can pass suggestion on where to find the package using the `CMAKE_PREFIX_PATH` environment variable or the `-DCMAKE_PREFIX_PATH=<path>` option when invoking CMake.

Specify:

- package name
- version
- whether it is required
- components to find
- whether to suppress messages
- whether to avoid default search paths

The CMake instructions explained

The next command is `add_executable()`, which is used to define an executable target:

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE]
            [EXCLUDE_FROM_ALL]
            source1 source2 ... sourceN)
```

An executable target is a binary file that can be run on the system, it can be created from *one* or *more* source files.

Specify:

- target name
- platform-specific options
- whether to exclude from default build
- source files

The last command is `target_link_libraries()`, which is used to specify libraries to link against a target.

```
target_link_libraries(<target>
            <PRIVATE|PUBLIC|INTERFACE> <item>...
            [<PRIVATE|PUBLIC|INTERFACE> <item>...]...)
```

Specify:

- target name
- libraries to link
- linkage type

Private, Public, and Interface linkage

When using `target_link_libraries()`, you can specify the linkage type:

- **PRIVATE**: the library is used only for the target itself.
- **PUBLIC**: the library is used for both the target and any targets that link against it.
- **INTERFACE**: the library is used only for targets that link against the target, not for the target itself.

Example

```
target_link_libraries(my_executable
    PRIVATE libA
    PUBLIC libB
    INTERFACE libC)
```

In this example, `libA` is linked only to `my_executable`, `libB` is linked to both `my_executable` and any targets that link against it, and `libC` is linked only to targets that link against `my_executable`.

Configuring and building

To configure and build the project with CMake the steps are:

1. Create a build folder: `mkdir build`
2. Move to the build folder and launch the `cmake` program

```
cd build
cmake .. # You could also try doing ccmake .. for an interactive configuration
```

3. Build the project using the generated build system, for example:

Make	run `make`
Ninja	run `ninja`

this will compile the code and generate the executable in the `build` folder.

Make a commit

If everything works, we can make a commit of the results.

💡 it is a good idea to create a `.gitignore` file to avoid committing build *artifacts*.

For doing this, you run

```
touch .gitignore
```

and then with your favourite editor write inside it

```
build/
```

Everything which is listed here is going to be ignored by git.

Make a commit

If everything works, we can make a commit of the results.

- it is a good idea to create a `.gitignore` file to avoid committing build *artifacts*.

For doing this, you run

```
touch .gitignore
```

and then with your favourite editor write inside it

```
build/
```

Everything which is listed here is going to be ignored by git. Now we can add all the files and make a commit:

```
git add .
git commit -m "Initial commit: OpenMP hello world with CMake"
```

Continuous Integration (CI) with GitHub Actions

We can adapt our last example of continuous integration (CI) with GitHub Actions from the previous lecture to build and test our OpenMP project. We need to create a workflow file in the `.github/workflows` folder.

1. Create the folders:
   ```
   mkdir -p .github/workflows
   ```
2. Create the workflow file:
   ```
   touch .github/workflows/CI.yml
   ```
3. Edit the file (starting from the one seen in the previous lecture).

Editing the CI.yml file

The content of the `CI.yml` file should be as follows:

```
name: CI
on:
  push:
    branches:
      - main
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
      - name: Setup CMake (latest)
        uses: lukka/get-cmake@latest
      - name: Setup Fortran
        uses: fortran-lang/setup-fortran@v1.8.0
```

- Define workflow name and trigger on push to main branch
- Set up Ubuntu environment
- Checkout code, **set up CMake** and Fortran compiler

```
with:
  compiler: gcc
  version: 'latest'
  update-environment: true
```

Editing the CI.yml file

The content of the `CI.yml` file should be as follows:

```
- name: Configure (CMake)
  run: cmake -S . -B build
  ↪  -DCMAKE_BUILD_TYPE=Release
- name: Build (CMake)
  run: cmake --build build --config Release
  ↪  -- -j
- name: Run program
  env:
    OMP_NUM_THREADS: '4'
  run: |
    ./build/hello-openmp || (echo
    ↪  "Executable not found" && ls -la
    ↪  build && exit 1)
```

- **Configure** and **build** project using CMake
- Run the compiled OpenMP program with 4 threads

Table of Contents

4 Building Blocks for Dense Linear Algebra

Motivation: Cholesky factorization example

Symmetric Matrix

A matrix $A \in \mathbb{R}^{n\times n}$ is called symmetric if $A = A^\top$, meaning that it is equal to its transpose.

Eigenvalue and Eigenvector

Given a square matrix $A \in \mathbb{R}^{n\times n}$, a non-zero vector $\mathbf{v} \in \mathbb{R}^n$ is called an eigenvector of A if there exists a scalar $\lambda \in \mathbb{R}$ such that:

$$A\mathbf{v} = \lambda\mathbf{v}$$

The scalar λ is referred to as the eigenvalue corresponding to the eigenvector $\mathbf{v}$. All eigenvalues of a symmetric matrix are real.

Motivation: Cholesky factorization example

Positive Definite Matrix

A symmetric matrix $A \in \mathbb{R}^{n \times n}$ is called positive definite if for all non-zero vectors $\mathbf{x} \in \mathbb{R}^n$:

$$\mathbf{x}^\top A \mathbf{x} > 0$$

This implies that all eigenvalues of A are positive.

Examples of symmetric positive definite matrices

- Covariance/correlation matrices in statistics and machine learning.
- Normal equations: $A^\top A$ from least squares; SPD if A has full column rank.
- Gram/kernel matrices: $K_{ij} = k(x_i, x_j)$ with strictly PD kernels (e.g., Gaussian/RBF).
- Precision (inverse covariance) matrices in Gaussian Markov random fields.

Motivation: Cholesky factorization example

- The Cholesky factorization is a method for decomposing a positive definite matrix A into the product of an upper triangular matrix U and its transpose:

$$A = U^\top U$$

- It is useful for solving systems of linear equations, and inverting matrices.
- It is computationally efficient, requiring approximately $\frac{1}{3}n^3$ operations for an $n \times n$ matrix.

Theorem (Existence and uniqueness)

Every symmetric positive definite matrix A has a unique Cholesky factorization $A = U^\top U$, where U is an upper triangular matrix with positive diagonal entries.

Motivation: Cholesky factorization example

Consider the Cholesky factorization $A = U^\top U$:

Algorithm

```
1: for $j = 1$ to $n$ do
2:     for $i = 1$ to $j - 1$ do
3:         $u_{ij} \leftarrow \frac{1}{u_{ii}} \left( a_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj} \right)$
4:     end for
5:     $u_{jj} \leftarrow \sqrt{a_{jj} - \sum_{k=1}^{j-1} u_{kj}^2}$
6: end for
```

- Easy to translate to any language
- But…“reinventing the wheel”
- Similar patterns appear repeatedly
- Lots of code duplication

The key observation

Similar code patterns resurface over and over again
in linear algebra algorithms

Natural strategy

*“Define a set of operators such that any algorithm
can be expressed as their application to the data at hand.”*

- Some languages provide native operators (MATLAB, Fortran, Julia)
- Algorithms = sequences of primitive operator calls

Benefits of standardized building blocks

1. **Code reuse**
   - Write once, use many times
   - Amortize cost of high-quality implementation
2. **Standardized interfaces**
   - Explore alternative implementations
   - Preserve overall code behavior
3. **Architecture-aware optimizations**
   - Exploit cache hierarchies
   - Use block/submatrix operations (not just vectors)
4. **Portability across systems**
   - Same interface, optimized per platform

Scope of application

- Cholesky is just one example
- Same reasoning applies to:
  - Dense linear algebra (LU, QR, eigensolvers, ...)
  - Sparse linear algebra (SpMV, iterative solvers, ...)
  - Many other numerical algorithms

- Encapsulation enables:
  - Performance tuning without changing user code
  - Leveraging hardware accelerators (GPUs, vector units)
  - Evolution of implementations over time

This is the foundation of BLAS and LAPACK

The Basic Linear Algebra Subprograms (BLAS)

- Set of low-level routines for common linear algebra operations
- Designed to be efficient and portable
- Building block for higher-level libraries (LAPACK, ScaLAPACK, PSBLAS, PETSc)
- Available in many programming languages (C, Fortran, Python)

Focus of this section

Dense BLAS: routines for dense matrices and vectors

BLAS organization: three levels

Level 1: Vector operations

- Examples: dot product, vector addition, scaling
- Complexity: $\mathcal{O}(n)$
- Memory-bound

Level 2: Matrix-vector operations

- Examples: matrix-vector multiplication, rank-1 updates
- Complexity: $\mathcal{O}(n^2)$
- Memory-bound

Level 3: Matrix-matrix operations

- Examples: matrix-matrix multiplication (GEMM)
- Complexity: $\mathcal{O}(n^3)$
- Compute-bound (high data reuse)

Popular BLAS implementations

OpenBLAS:	Open-source implementation of BLAS and LAPACK
ATLAS:	Automatically Tuned Linear Algebra Software; open-source, self-optimizing
Intel MKL:	High-performance library optimized for Intel processors
cuBLAS:	GPU-accelerated BLAS for NVIDIA GPUs
BLIS:	Portable, high-performance, modern BLAS framework

Key takeaway

Same interface, different implementations $\Rightarrow$ performance portability

Finding BLAS with CMake

- CMake provides a built-in module to find BLAS libraries
- Use `find_package(BLAS REQUIRED)` to locate BLAS
- Link against the found BLAS library using `target_link_libraries(<target> PRIVATE ${BLAS_LIBRARIES})`
- Information are available on the webpage: FindBLAS module documentation.

Example CMake snippet

```
find_package(BLAS REQUIRED)
target_link_libraries(<target> PRIVATE ${BLAS_LIBRARIES})
```

Summary and Next Steps

- OpenMP is a widely used API for shared-memory parallel programming.
- It provides a simple and flexible way to parallelize code using compiler directives.
- CMake can be used to manage the build process of Fortran projects with OpenMP.
- The Basic Linear Algebra Subprograms (BLAS) provide standardized building blocks for dense linear algebra operations.
- Using BLAS enables code reuse, portability, and performance optimizations across different hardware architectures.

Summary and Next Steps

- OpenMP is a widely used API for shared-memory parallel programming.
- It provides a simple and flexible way to parallelize code using compiler directives.
- CMake can be used to manage the build process of Fortran projects with OpenMP.
- The Basic Linear Algebra Subprograms (BLAS) provide standardized building blocks for dense linear algebra operations.
- Using BLAS enables code reuse, portability, and performance optimizations across different hardware architectures.

Next Steps

- Explore more advanced OpenMP features (e.g., task parallelism, SIMD).
- Use Fortran and OpenMP features to look through BLAS implementations.

References

6 Bibliography

[1] G. J. Barbara Chapman and R. van der Pas. *Using OpenMP*. MIT Press, Cambridge, MA, 2007, p. 384. ISBN: 9780262533027.

[2] O. A. R. Board. *OpenMP Application Programming Interface Specification 5.2*. Ed. by B. de Supinski and M. Klemm. 2021. ISBN: 979-8497370195. URL: `https://www.openmp.org/specifications/`.

[3] O. A. R. Board. *OpenMP Application Programming Interface Specification 6.0*. 2024. URL: `https://www.openmp.org/specifications/`.

[4] J. D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25.

References
6 Bibliography

[5] J. D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. A continually updated technical report. http://www.cs.virginia.edu/stream/. Charlottesville, Virginia: University of Virginia, 1991-2007. URL: `http://www.cs.virginia.edu/stream/`.

[6] Y. (H. Timothy G. Mattson and A. E. Koniges. *The OpenMP Common Core*. MIT Press, Cambridge, MA, 2019, p. 320. ISBN: 9780262538862.

[7] S. Williams, A. Waterman, and D. Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: `10.1145/1498765.1498785`. URL: `https://doi.org/10.1145/1498765.1498785`.