

Good Coding Practices

Thomas Madlener

FH Sustainable Computing Workshop

Nov 07, 2024

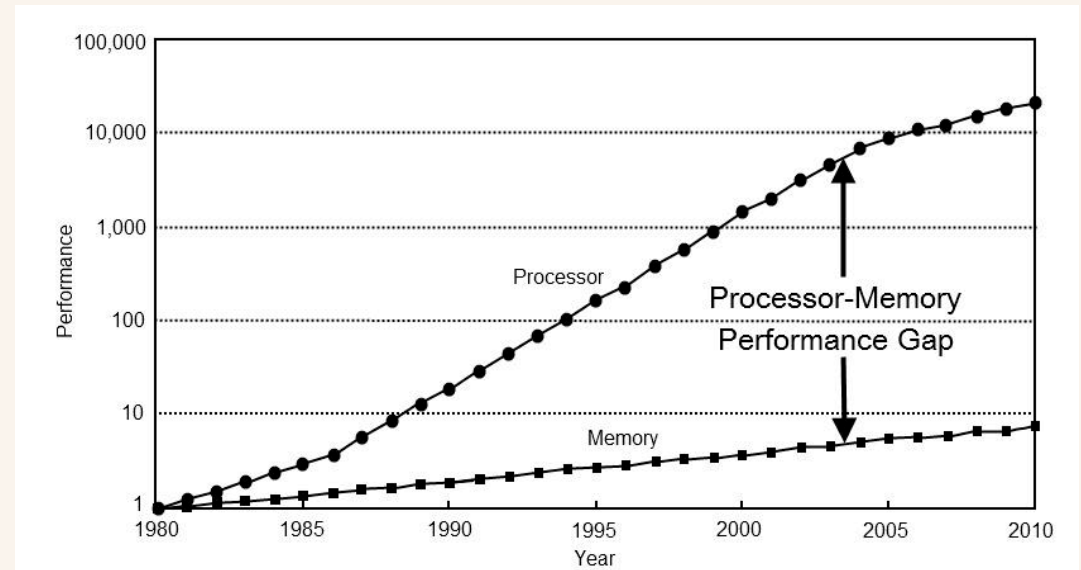
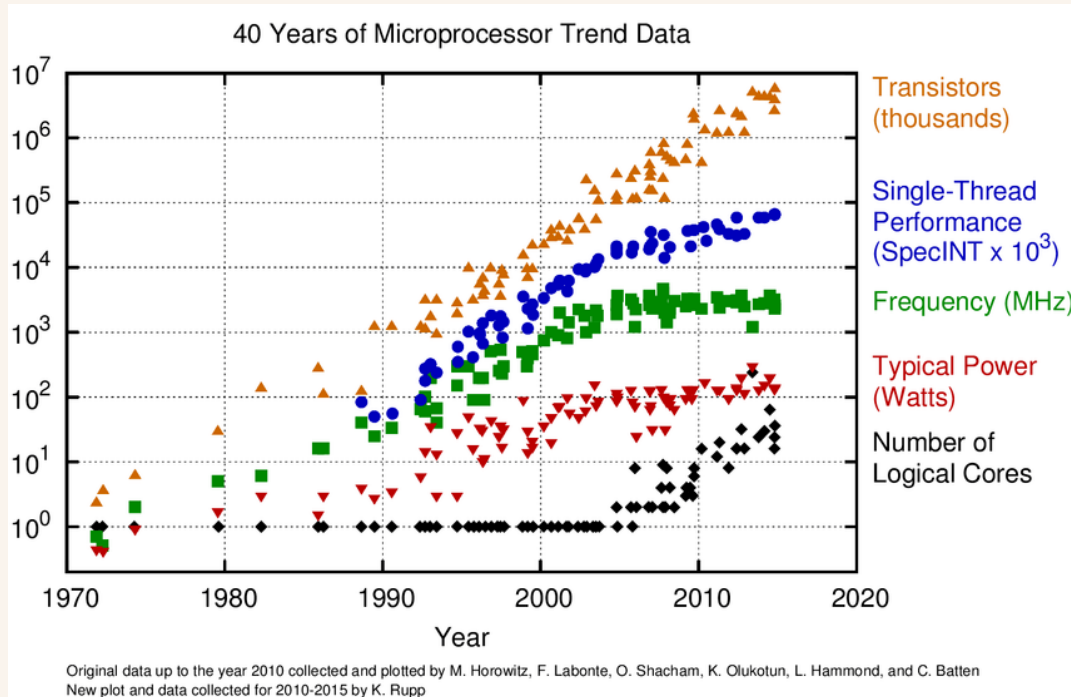
Menu for today

- CPU and memory basics for performance
- Sustainability aspects (including human resources)
- Avoiding common performance pitfalls in C++
- Some exercises (and food for thought)

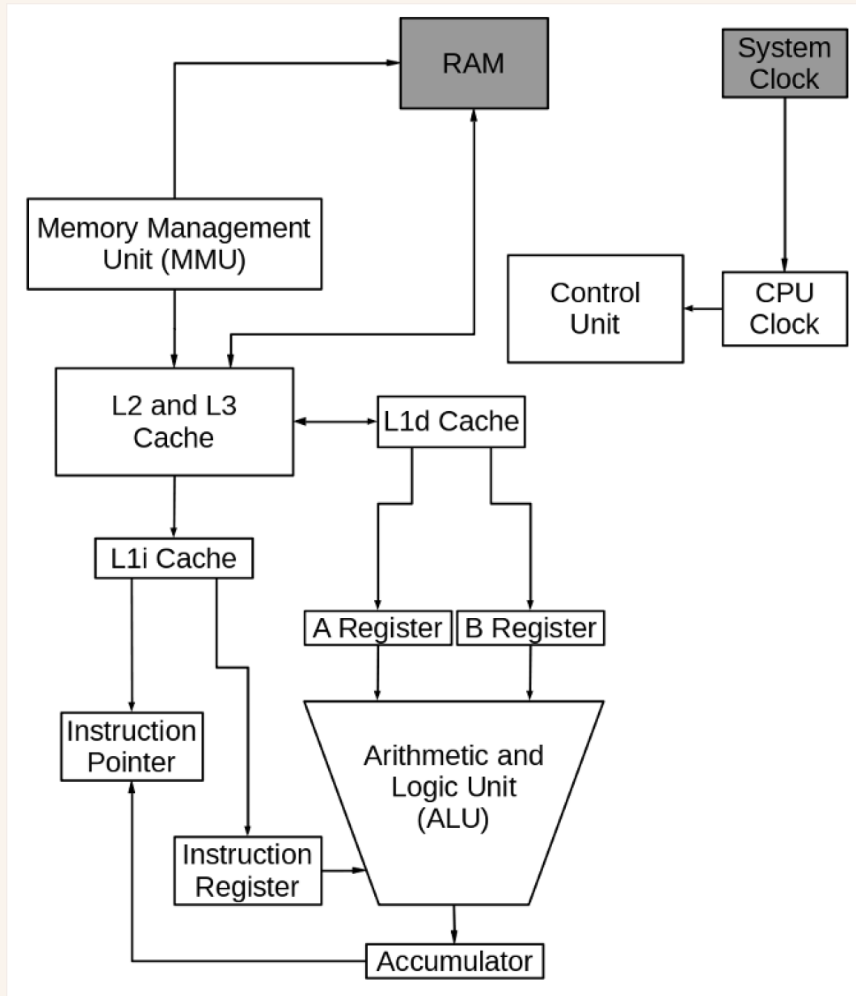
What not to expect

- Introduction to c++ / python from scratch
 - See the HSF Training Courses for that
- GPU / heterogeneous resources
- In depth discussion of leveraging CPU features
- Profiling
- "Proper" benchmarking

Development of CPUs & Processor-memory gap

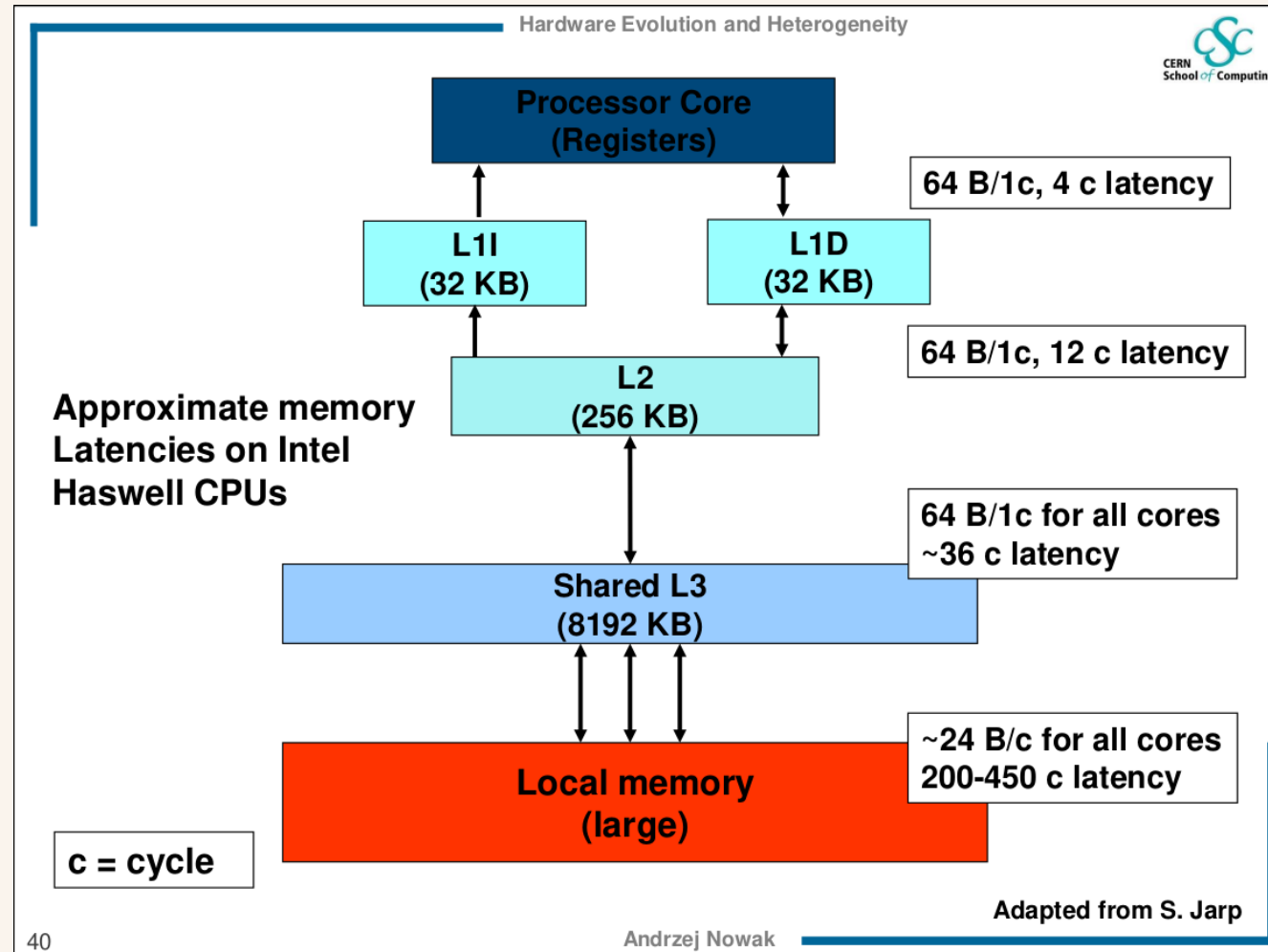


A modern CPU is a complicated beast



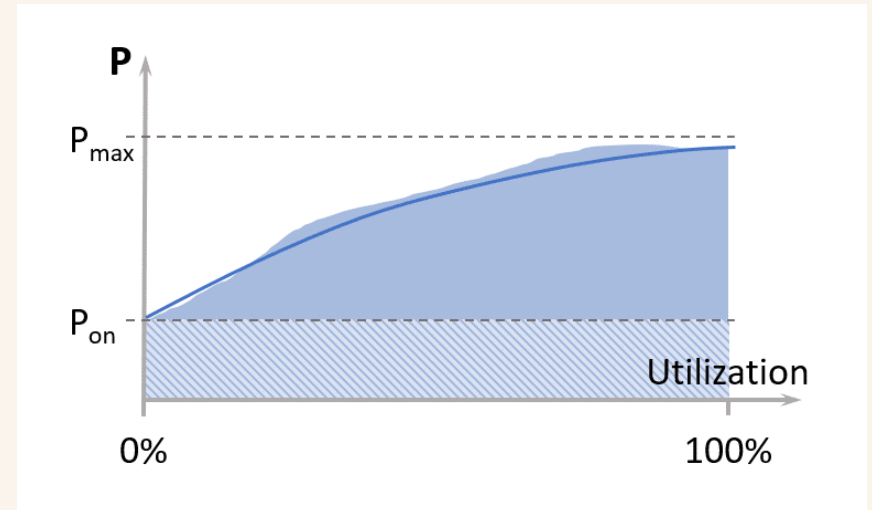
- Multithreading
- Hyperthreading
- **Caching on multiple levels**
- Instruction pipelining
- Speculative execution / branch prediction
- Vectorization

Memory is king



Sustainability & Performance

- Execution time $\propto$ energy consumption
arXiv:2410.05460
 - Better performance $\rightarrow$ more sustainable
- A stalled CPU still consumes power!
 - Infrastructure as well
- FLOPS / Watt numbers assume **full CPU utilization**
 - (100 % only achievable in theoretical scenarios)



Practical advice

- Make data contiguous and cache friendly
 - Avoid pointers & virtual functions where possible
- Make data requests cache-friendly and **predictable**
- Design with data flow in mind
 - "Natural" in many cases in HEP
- **Write simple code**
 - Easier to maintain and understand
 - Compiler might have an easier job optimizing it

What does *Cache friendly* even mean?

- Data that is accessed together is close by in memory
 - CPU can "guess" which data are needed next
 - (Pre)fetches them into caches to make them quickly available

```
// Actual Data will live scattered throughout memory
std::vector<Data*> ptrVec;
// Access might be slow due to "pointer chasing"
```

```
// All Data will be stored contiguously in memory
std::vector<Data> valueVec;
// Access likely very quick since the CPU knows where the next
// element lives in memory
```

Considerations for software design

- Necessary efforts depend on several factors
- (Expected) lifetime of the code you are writing?
- (Potential) users other than you?
 - Keep in mind *future you*!
- Software changes constantly
 - Divide into independent pieces when possible
 - No "spooky action at a distance"
- Take time to refactor if new requirements come up
- **(Automated) testing is part of the process**
- **Documentation is part of the process**

Building blocks for software design

- **Functions**
 - Avoid code repetition
 - Reduce variable scope / improve readability
 - Isolation of dependencies
- **class / struct**
 - Group data together
 - Ensure preservation of invariants
- **Naming**
 - Good naming reduces need for comments

General considerations

- **No mutable global state!**
- Immutable global variables / configuration OK
 - Keep as small as possible
- Avoid manual memory management
 - `std::unique_ptr` is a thing
- Use containers over C-style arrays
 - `std::vector` is almost always the right choice
 - Store values not pointers
- **Functions, functions, functions, ...**

Considerations for functions

- Split large functions into smaller ones
- Write "pure" functions
 - Easier to test
 - No side-effects to keep in mind
 - Pass arguments by `const&` by default
- Keep number of arguments low
 - Group input arguments into `class`es if necessary
- Try to avoid in-out parameters
 - Return multiple values
 - Group return value into a `class`

Split large functions into smaller ones

```
def complicated_function(args):  
    """This long function has all the lines"""  
    # step 1: read data  
    # ... very involved procedure to read data ...  
  
    # step 2: filter data  
    # ... do some stuff to filter out some things ...  
  
    # extract result 1  
    # ... complicated procedure to get some result ...  
  
    # extract another result  
    # ... entirely independent procedure for another result ...
```

- Common pattern
- Halfway there to functions
 - Even naming is solved already

Split large functions into smaller ones

```
def complicated_function(args):  
    """This long function has all the things but not the lines"""  
    data = read_data(args)  
  
    filtered_data = filter_data(data)  
  
    result_1 = get_result_1(filtered_data)  
  
    indep_res = get_independent_result(filtered_data)
```

- Common pattern
- Halfway there to functions
 - Even naming is solved already (to a certain point)
- There are even tools to help with this!

Passing function arguments in C++

```
// Pass by value (do this for small objects)
// --> Copy the inputs
// --> No changes visible outside (automatically threadsafe)
void process_1(vector<Data> inputs);

// Pass by reference (this should almost never be necessary!)
// --> No copy
// --> Function CAN mutate inputs (NOT threadsafe!)
void process_2(vector<Data>& inputs);

// Pass by const reference (do this for large objects)
// --> No copy
// --> Function CANNOT mutate inputs (threadsafe)
void process_2(const vector<Data>& inputs);
```

Avoid in-out parameters

```
bool process(vector<Data> const& inputs,
             vector<Data>& output,
             double& efficiency);

// ===== Usage =====
vector<Data> output{};
double procEff;
if (process(inputs, output, procEff)) {
    // do something
}
```

- Complicates const-correctness
- "Noisy"

Avoid in-out parameters

```
std::tuple<bool, vector<Data>, double>
process(vector<Data> const& inputs);

const auto& [success, output, procEff] = process(inputs);
if (success) {
    // do something
}
```

- Use structured bindings
- Introduce a simple `struct` or `class` if applicable
- Consider `std::optional`

Const correctness in C++

- C++ has the `const` keyword
 - Mark variables, function parameters and member functions as immutable
- Allows compiler to more aggressively optimize
- Communicates intent to users / developers
- **Since C++11 a `const` member function is assumed to be *thread-safe*!**
- Unfortunately not the default in C++

Basics of Testing

- Different levels of tests
- Small (pure) functions make writing unit tests easier
- Write tests in parallel to other code
- Also check "unhappy" paths
- Every language has (unit) testing frameworks
- Make tests quick to run
- Run them as part of the development cycle
 - **A bug that is caught by a test doesn't need debugging!**
- Automate running tests (CI)
- (Tool assisted) refactoring much easier if you have tests

Final thoughts (1 / 2)

- Use an editor that works with you not against you
 - Syntax highlighting, autocomplete, code browsing, documentation, ...
 - VS Code is a good starting point
- ChatGPT (and friends) are great but not always right
 - Treat them as "better autocomplete" and check what they produce!
 - Keep in mind ToS and what they say about your inputs

Final thoughts (2 / 2)

- Error messages can be useful if read completely
 - Starting at the top is usually a good idea
- Enable compiler warnings and treat them as errors by default
 - `-Werror` for enforcement by the compiler
- Jupyter notebooks are great for prototyping
 - Not so much for storing (and versioning!) your code

Resources & useful links

- [HSF Training website](#) - material for various languages and tools
- cppreference.com - reference page for c++ & STL
- godbolt.org - "compiler explorer", online c++ compiler
- isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Exercises

- Pick and choose
- Solutions / inspiration included
- c++ exercises
 - Easy performance gains / pitfalls, writing const correct code
 - Refactoring an existing analysis
- python exercises
 - Unit testing introduction

Exercises repositories

gitlab.desy.de/fh-sustainability-forum/sustainable-coding-tutorial/

- **software-exercises** (main exercises for c++ and python)
- **python-unittesting** (intro to `pytest` and unittesting with python)
- **cpp-unittesting** (intro to unittesting with c++)