BEST CODING PRACTICES

Bohdan Dudar

FH Sustainable Computing Workshop | Jan 18, 2024

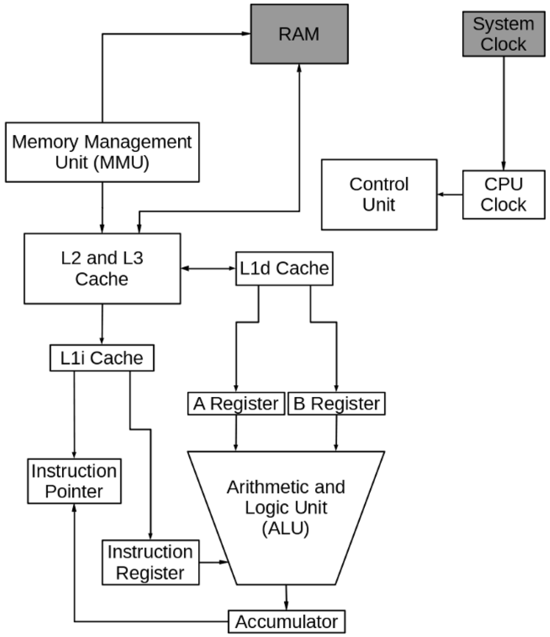
MENU FOR TODAY

- 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

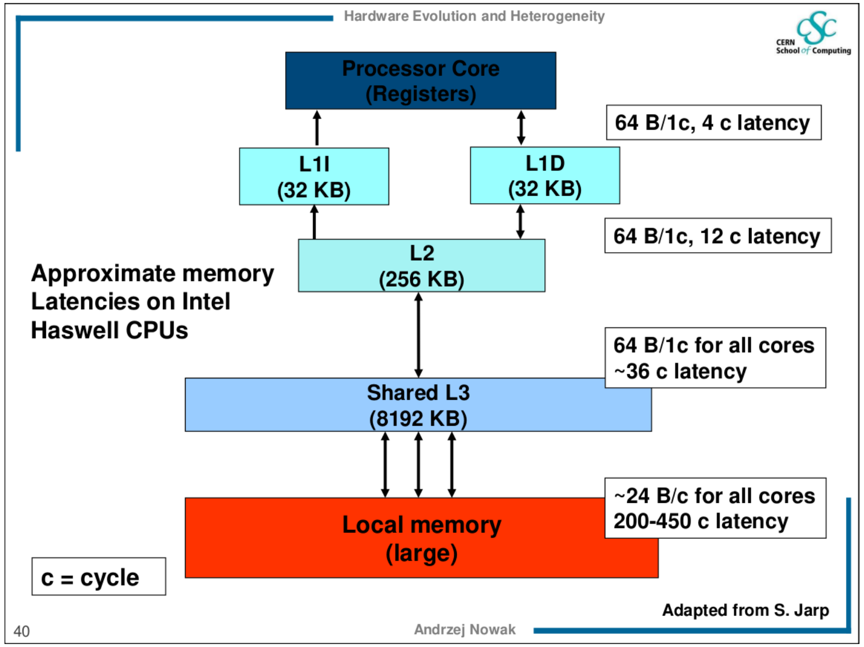
A MODERN CPU IS A COMPLICATED BEAST



FEATURES OF MODERN CPUS

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

MEMORY IS KING



Hardware Evolution and Heterogeneity

CERN School of Computing

Processor Core
(Registers)

64 B/1c, 4 c latency

L1I
(32 KB)

L1D
(32 KB)

64 B/1c, 12 c latency

L2
(256 KB)

Approximate memory
Latencies on Intel
Haswell CPUs

64 B/1c for all cores
~36 c latency

Shared L3
(8192 KB)

~24 B/c for all cores
200-450 c latency

Local memory
(large)

c = cycle

Adapted from S. Jarp

40

Andrzej Nowak

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 `classes` if necessary
- Try to avoid in-out parameters
  - Return multiple values
  - Group return value into a `class`

SPLIT LARGE FUNCTIONS INTO SMALLER ONES

```python
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)

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!

FINAL THOUGHTS (2 / 2)

- Error messages can be useful if read completely
- 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/CppCoreGuidelir

EXERCISES

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

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