Lecture 09

Optimization, debugging, profiling, testing.

Advanced Programming - SISSA, UniTS, 2025-2026

Pasquale Claudio Africa

25 Nov 2025

Outline

1. Optimization
2. Debugging
3. Profiling
4. Testing

The *Three versions* principle - Joe Armstrong

> **Every non-trivial program needs to be written three times.**
>
> First make it work, then make it beautiful, then if you really, really have to, make it fast.

1. **Make it work**
   - Focus on correctness and understanding the problem, no optimization
   - Implement straightforward, readable solutions
2. **Make it right and beautiful**
   - Refactor for clarity and maintainability, write tests to ensure correctness
   - Fix bugs, add proper error handling
3. **Make it fast**
   - Profile to find actual bottlenecks
   - Apply optimizations where needed, maintaining correctness through testing

Premature optimization wastes time on code that may be rewritten anyway!

Optimization

Code optimization

Code optimization is the process of enhancing a program's performance, efficiency, and resource utilization without changing its functionality. It involves improving execution speed, reducing memory usage, and enhancing overall system responsiveness.

Optimization techniques:

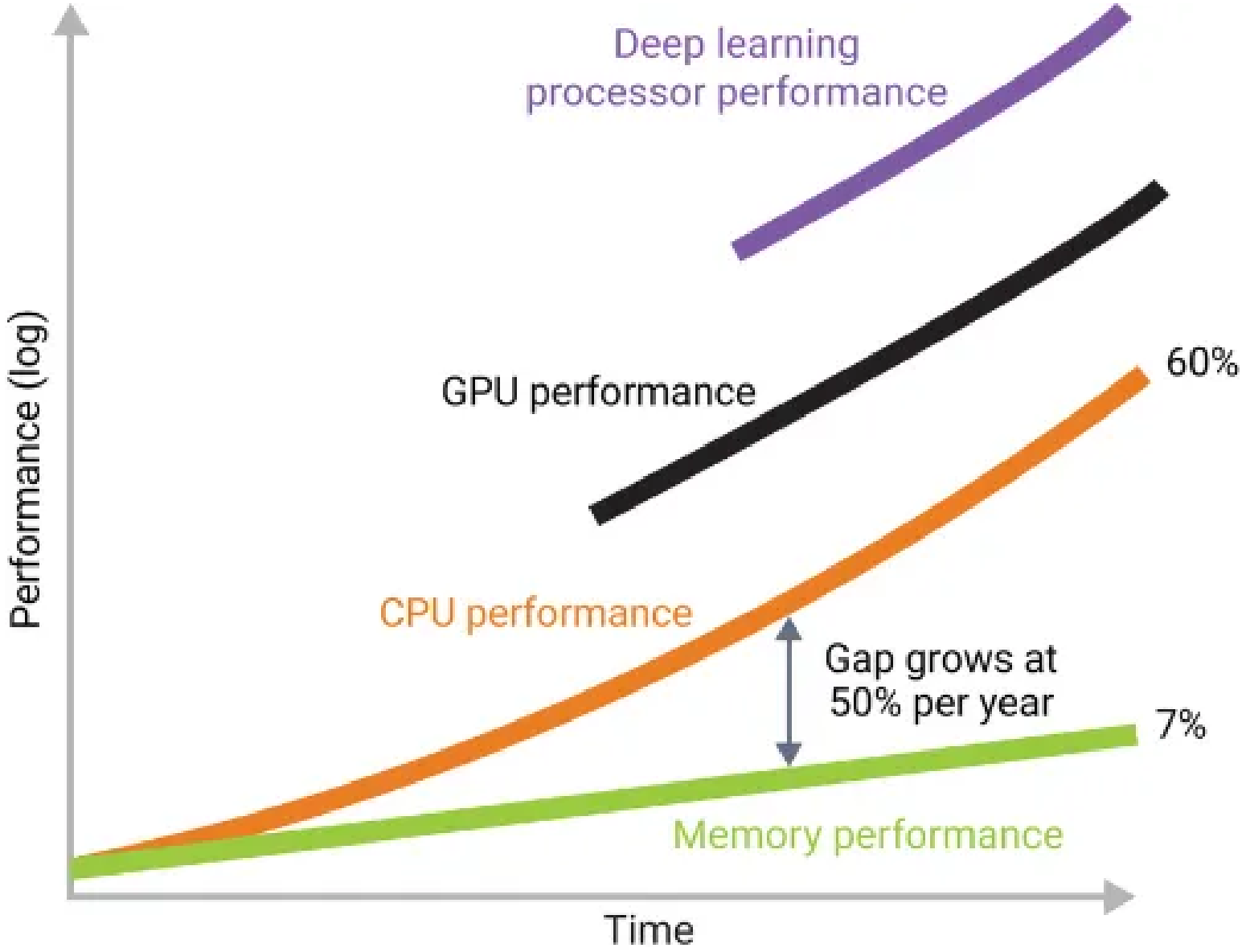
- **Compiler optimizations:** Utilize compiler features to automatically enhance code during compilation.
- **Algorithmic optimization:** Improve the efficiency of algorithms and data structures.
- **Manual refactoring:** Restructure code for better readability, maintainability, and performance.
- **Profiling and analysis:** Use profiling tools to identify and optimize performance bottlenecks.
- **Hardware-aware optimization:** Leverage CPU features like SIMD, cache hierarchy, and branch prediction.

Optimization options

The compiler enhances performance by optimizing CPU register usage, expression refactoring, and pre-computing constants.

- Disable optimization during debugging.
- Pass the `-O{n}` (`n={0,1,2,s,3}`) flag to the compiler to control optimization level, with `-Os` for space optimization and `-O3` for maximum optimization. Here a detailed list of optimizations enabled with each flag.
- Defining the `-DNDEBUG` preprocessor variable, standard assertions are ignored, resulting in faster code.
- `-Ofast` enables `-O3` optimizations plus fast math optimizations that may violate strict IEEE compliance.
- For debugging, use `-Og` which optimizes while maintaining good debugging experience (better than `-O0`).

Source


Deep learning
processor performance
GPU performance
60%
CPU performance
Gap grows at
50% per year
7%
Memory performance
Performance (log)
Time

Loop unrolling

It is beneficial to unroll small loops. For example, transform:

```
for (int i = 0; i < n; ++i)
  for(int k = 0; k < 3; ++k)
    a[k] += b[k] * c[i];
```

to:

```
for (int i = 0; i < n; ++i) {
  a[0] += b[0] * c[i];
  a[1] += b[1] * c[i];
  a[2] += b[2] * c[i];
}
```

Compiler may unroll loops with `-funroll-loops`, but better performance isn't guaranteed. **Note:** Loop unrolling increases code size. For large loops or loops with large bodies, unrolling may degrade performance due to instruction cache pressure. Always profile!

Multiply vector entries: two strategies compared

```
double multiply(const std::vector<double> &data) {
  double result = 1;
  for (const auto &v : data)
    result *= v;
  return result;
}

double multiply_with_unrolling(const std::vector<double> &data) {
  double result = 1; double a0, a1, a2, a3;
  size_t i;

  for (i = 0; i + 3 < data.size(); i += 4) {
    a0 = data[i];
    a1 = data[i + 1];
    a2 = data[i + 2];
    a3 = data[i + 3];

    result *= a0 * a1 * a2 * a3;
  }
  for (; i < data.size(); ++i)
    result *= data[i];
  return result;
}
```

Which one is faster?

The number of floating point operations is the same in both cases!

The answer is not straightforward: it depends on the computer's architecture.

On my laptop (Intel(R) Core(TM) Ultra 7 155H CPU @ 4.80GHz), `multiply_with_unrolling` **is approximately 3-4 times faster than** `multiply` with `size = 1e6`! (see `examples/optimization/loop_unrolling.cpp`).

Why? The Streaming SIMD Extensions (SSE2) instruction set of the CPU allows for parallelization at the microcode level. It's a super-scalar architecture with multiple instruction pipelines to execute several instructions concurrently during a clock cycle. The unrolled code better exploits this capability.

Note: Modern compilers with `-O3 -march=native` may automatically perform similar optimizations (sometimes, giving it a hand is beneficial). **Counting operations doesn't necessarily reflect performance.**

Prefetching constant values

Prefetch constant values inside the loop for further optimization. For example, transform:

```
for (int i = 0; i < n; ++i) {
  a[0] += b[0] * c[i];
  a[1] += b[1] * c[i];
  a[2] += b[2] * c[i];
}
```

to:

```
for (int i = 0; i < n; ++i) {
  auto x = c[i];
  a[0] += b[0] * x;
  a[1] += b[1] * x;
  a[2] += b[2] * x;
}
```

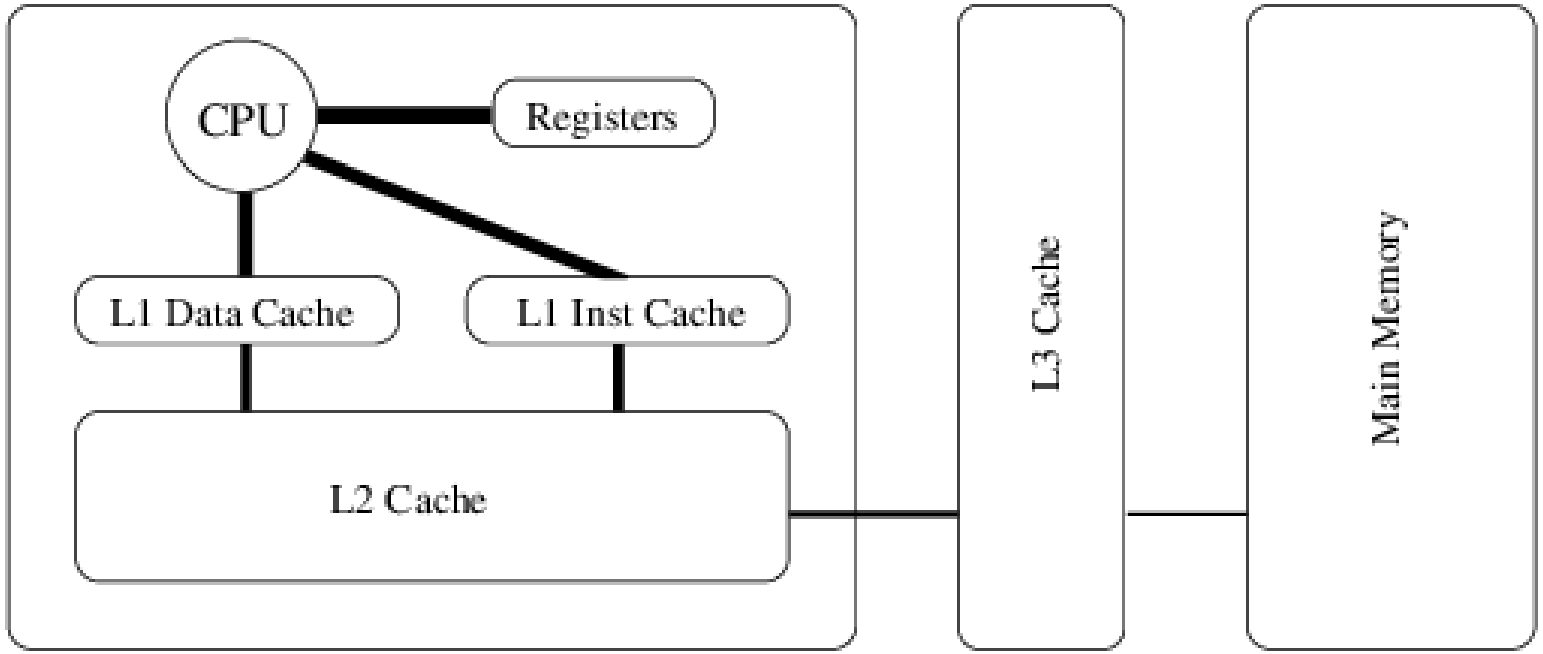
Avoid `if` inside nested loops

`if` statements, especially in nested loops, can be costly. Consider these improvements:

```
for(int i = 0; i < 10000; ++i) {
  for (int j = 0; j < 10; ++j) {
    if(c[i] > 0)
      a[i][j] = 0;
    else
      a[i][j] = 1;
  }
}
// Better:
for(int i = 0; i < 10000; ++i)
  const int value = (c[i] > 0) ? 0 : 1;
  for(int j = 0; j < 10; ++j)
    a[i][j] = value;
}
```

Note: Modern CPUs have branch predictors. Branches are less costly if the pattern is predictable.

Memory layout



Cache friendliness

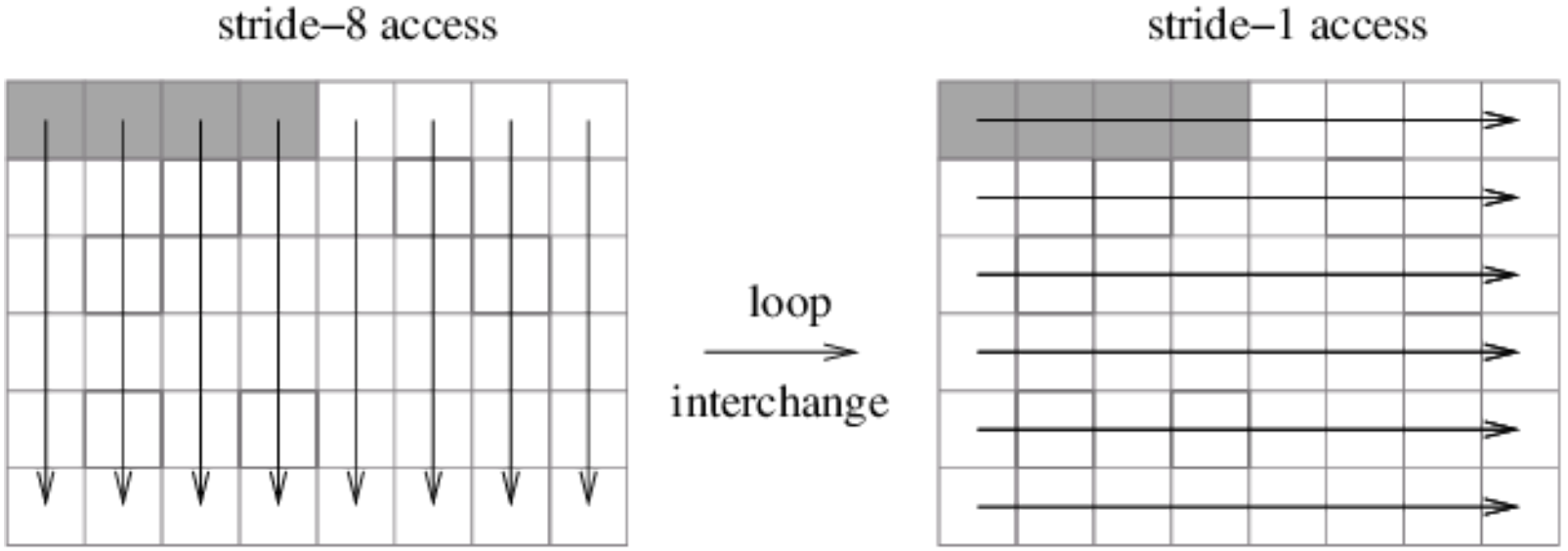
Modern CPUs have cache hierarchy (L1/L2/L3). Cache line size is typically 64 bytes (8 words). Accessing memory sequentially (i.e., contiguously) allows prefetching and reduces cache misses dramatically.

Example performance difference: If `mat` is a dynamic matrix organized **row-wise**, column-major vs row-major access time can differ by 10-100x!

```
// Not cache-friendly, inefficient.
for (j = 0; j < n_cols; ++j)
  for (i = 0; i < n_rows; ++i)
    sum += mat(i, j);

// Cache-friendly, thus more efficient.
for (i = 0; i < n_rows; ++i)
  for (j = 0; j < n_cols; ++j)
    sum += mat(i, j);
```

Cache friendliness (for a row-major matrix)




```
1: double sum;
2: double a[n, n];
3: // Original loop nest:
4: for j = 1 to n do
5:     for i = 1 to n do
6:         sum+ = a[i, j];
7:     end for
8: end for
```

```
1: double sum;
2: double a[n, n];
3: // Interchanged loop nest:
4: for i = 1 to n do
5:     for j = 1 to n do
6:         sum+ = a[i, j];
7:     end for
8: end for
```

Data structure alignment and padding

```
class MyClass
{
  char a;       // 1 byte.
  short int b; // 2 bytes.
  int c;        // 4 bytes.
  char d;       // 1 byte.
};
```

How data is *not* stored

Size of 1 block = 1 byte

Size of 1 row = 4 byte

a	b	b	c
c	c	c	d

How data is actually stored

Size of 1 block = 1 byte

Size of 1 row = 4 byte

a	padding	b	b
c	c	c	c
d	padding	padding	padding

Common optimization pitfalls (1/2)

1. **Premature optimization**
    - *"Premature optimization is the root of all evil"* - Donald Knuth
    - Profile first, optimize second
    - Focus on algorithmic improvements before micro-optimizations
2. **Over-optimization**
    - Don't sacrifice readability for marginal gains
    - Compiler often outsmarts manual micro-optimizations
    - Always value code maintainability

Common optimization pitfalls (2/2)

3. **Not measuring**
    - Always benchmark before and after
    - Use consistent test conditions
    - Consider multiple architectures

Golden rule: Measure, optimize, verify!

Examples

The folder `examples/optimization/` contains three examples:

1. **`data_alignment`** compares the memory occupation of two objects containing the same data members but with different data alignment/padding.
2. **`loop_unrolling`** implements a function that multiplies all elements in a `std::vector` by looping over all its elements and returns the result. The executable compares the performance with those obtained exploiting loop unrolling.
3. **`static`** implements a function that allocates a `std::vector` and, taking an index as input, returns the corresponding value. The executable compares the performance with those obtained by declaring the vector `static`.

Debugging

Static analysis vs. debugging (1/2)

Static analysis:

- **Nature:** Examines code without executing it.
- **Purpose:** Identifies potential issues (e.g., detecting null pointer dereferences, unused or uninitialized variables) and coding standards violations.
- **Tools:** Code linters, security scanners, and complexity analyzers.
- **Integration:** Often part of development workflows or continuous integration.

Debugging:

- **Nature:** Inspects and troubleshoots code during runtime.
- **Purpose:** Locates and resolves bugs, runtime errors, and unexpected behavior (e.g., why a specific portion crashes or produces wrong results).
- **Tools:** Debuggers with features like breakpoints and variable inspection.
- **Integration:** Interactive process during development or post-runtime.

Static analysis vs. debugging (2/2)

Key differences

- **Timing:** Static analysis is pre-runtime; debugging is during or post-runtime.
- **Focus:** Static analysis emphasizes code quality; debugging resolves runtime issues.
- **Use cases:** Static analysis is proactive; debugging is reactive.
- **Automation:** Static analysis tools can be automated; debugging is more interactive.
- **Complementarity:** Both are complementary, with static analysis preventing issues and debugging addressing runtime problems.

Static analysis

Static analysis tools analyze source code by inspecting it for potential issues, vulnerabilities, or adherence to coding standards. Common ones include:

- cppcheck
- cpplint
- clang-tidy

Some of the checks they perform:

- Automatic variable checking.
- Bounds checking for array overruns.
- Unused functions, variable initialization and memory duplication.
- Invalid usage of Standard Template Library functions and idioms.
- Missing `#include`s.
- Memory or resource leaks, performance errors.

Other useful tools

- Cling is an interactive C++ interpreter, built on LLVM and Clang. It's part of the ROOT project at CERN and can be integrated into a Jupyter workspace (see here). While experimental, an interpreter aids in code prototyping.
- Compiler Explorer to check how code translates into assembly language. Useful for understanding optimization effects and assembly code generation.
- C++ Insights allows viewing source code through a compiler's eyes.

Debuggers

Debuggers are software tools that enable developers to inspect, analyze, and troubleshoot code during the development process. They provide a set of features for identifying and fixing errors in programs.

Key features

- **Breakpoints:** Pauses program execution at specified points to inspect variables and code.
- **Variable inspection:** Allows developers to examine the values of variables during runtime.
- **Step-through execution:** Enables line-by-line execution for precise debugging.
- **Call stack analysis:** Displays the sequence of function calls leading to the current point in code.

Debugging

During code development, debugging allows step-by-step execution. To use a debugger, compile with `-g` (which implies no optimization). `-g` adds information for locating source lines in machine code.

Two debugging types:

- **Static debugging:** Analyze core dump if code aborts.
- **Dynamic debugging:** Execute through a debugger, breaking at points to examine variables.

Two common debuggers are `gdb` and `lldb`. See, e.g.:

- GDB cheat sheet
- GDB tutorial

Debugging levels

Debugging can be at different levels, and using `-g -O` together is allowed. `-g` tells the compiler to provide extra information for the debugger. However, line-by-line debugging reliability decreases with optimization. `-g` implies `-O0` by default.

Debugging levels and special optimization options linked to debugging:

- `-g0`: No debugging information.
- `-g1`: Minimal information for backtraces.
- **The default debugging level is 2.**
- `-g3`: Extra information, including macro definitions.
- `-Og`: Special optimization option. Enables optimizations without interfering with debugging.

Main commands of `gdb` / `lldb`

- `run` : Run the program.
- `break` : Set a breakpoint at a line/function.
- `where` : Show location and backtrace.
- `print` : Display variable/expression value.
- `list n` : Show lines around line n.
- `next` : Go to the next instruction, proceeding through subroutines.
- `step` : Go to the next instruction, entering called functions.
- `continue` : Continue executing.
- `backtrace` : Print memory stack after program aborts.
- `quit` : Exit the debugger.
- `help` : Displays help information.

Other debugging tools

`valgrind`, a suite of tools for debugging and profiling. It can find memory leaks, unassigned variables, or check memory usage:

Find memory leaks:

```
valgrind --tool=memcheck --leak-check=yes --log-file=file.log executable
```

Check memory usage:

```
valgrind --tool=massif --massif-out-file=massif.out --demangle=yes executable
ms_print massif.out > massif.txt
```

`massif.txt` indicates memory usage during the program execution.

Debugging strategies

Scientific debugging method:

1. Reproduce the bug consistently.
2. Isolate the problem (binary search through code).
3. Form a hypothesis.
4. Test the hypothesis.
5. Fix and verify.

Useful techniques:

- Print statements (simple but effective).
- Assertions (`assert()`, `static_assert()`).
- Use version control to bisect (`git bisect`).
- Reduce to minimal reproducible example.

Examples

The content of `examples/debug/` was inspired by this repository and shows basic techniques for debugging as well as an introduction to `gdb`.

Debug vs. Release builds

- Always test in both configurations.
- Bugs may appear only in one (e.g., uninitialized variables).

Further readings

- Defensive programming and debugging.
- C++ undefined behaviour 101.
- Shocking undefined behaviour in action.

Profiling

Profilers

A **profiler** in software development is a tool or set of tools designed to analyze the runtime behavior and performance of a computer program. It provides detailed information about resource utilization, execution times, and function calls during the program's execution.

Key objectives

- **Performance analysis:** Profilers offer insights into how much time a program spends in different functions, helping identify performance bottlenecks.
- **Resource usage:** They measure memory consumption, CPU utilization, and other system resources, aiding in optimizing resource-intensive operations.
- **Function call tracing:** Profilers track the sequence of function calls, enabling developers to understand the flow of execution.

`gprof`

`gprof` is the GCC simple profiler. In order to use it, compile the code with the `-pg` option at both the compilation and linking stages.

When executing the code, it generates a file called `gmon.out`, which is then utilized by the profiler:

```
gprof --demangle executable > file.txt
```

Then `file.txt` will contain valuable information about the program execution.

Limitations

- Uses statistical sampling (may miss fast functions).
- Overhead from instrumentation.
- Limited support for multithreaded programs.

Main options of `gprof`

`gprof` offers a range of options. The main ones are:

- `--annotated-source[=symspec]`: Prints annotated source code. If `symspec` is specified, print output only for matching symbols.
- `-I dirs`: List of directories to search for source files.
- `--graph[=symspec]`: Prints the call graph analysis.
- `--demangle`: Demangles mangled names (essential for C++ programs).
- `--display-unused-functions`: As the name says.
- `--line`: Line-by-line profiling (but maybe better use `gcov`).

For production profiling: Consider `perf` (Linux) which uses hardware counters without code modification.

`callgrind`

`callgrind` is a tool of `valgrind` that you may call, for instance, as:

```
valgrind --tool=callgrind --callgrind-out-file=grind.out --dump-line=yes ./myprog
```

Compile the program with `-g` and **optimization activated**. The option `--dump-lines` is used for line-by-line profiling.

Afterward, post-process the binary file `grind.out`, e.g., using `kcachegrind`:

```
kcachegrind grind.out
```

It opens a graphical interface.

Profiling tools

When to use `gprof` vs. `callgrind`:

- `gprof`: Faster, good for quick profiling.
- `callgrind`: More accurate, cache simulation, call graphs.

There are alternative profilers, some useful in a parallel environment:

- perf: Lightweight CPU profiling. Best for production systems, minimal overhead.
- gperftools: Formerly Google Performance Tools.
- TAU (Tuning and Analysis Utilities): Profiling and tracing toolkit for parallel programs.
- Scalasca: Performance analysis for parallel applications on distributed memory systems.
- Likwid: A toolsuite of command line applications for performance oriented programmers. Supports threaded, parallel applications, and various arhitectures, including GPUs.

Profiling best practices

Setup:

- Profile with optimization enabled (`-O2` or `-O3`).
- Include debug symbols (`-g`) for readable output.

Analysis:

- Focus on *hot spots* (80/20 rule: 80% of time in 20% of code). Optimizing *cold code* wastes time.
- Profile both CPU time and memory usage.

Iterative process:

1. Profile → Identify bottleneck
2. Optimize → Re-profile and verify improvement
3. Repeat

Testing

Verification vs. validation

Verification: Ensuring correct implementation

Conducted during development, tests **individual components** separately. Specific tests demonstrate correct functionality, covering the code and checking for memory leaks.

Validation: Confirming desired behavior

Performed on the **final code**. Assesses if the code produces the intended outcome, such as convergence, reasonable results, and expected computational complexity.

Types of testing

- **Unit testing:** Testing individual components (functions, methods, or classes) to ensure each behaves as expected. It focuses on a specific piece of code in isolation.
- **Integration testing:** Verifying that different components/modules of the software work together as intended. It deals with interactions between different parts of the system.
- **Regression testing:** Ensuring recent code changes do not adversely affect existing functionalities. It involves re-running previous tests on the modified codebase to catch unintended side effects.
- **Performance testing:** Ensuring code meets performance requirements and doesn't regress.
- **Numerical accuracy testing:** Verifying numerical results against analytical solutions or reference implementations (crucial for scientific computing).
- **Stress testing:** Testing behavior under extreme conditions or edge cases.

Importance of testing

- **Early detection of bugs:** Testing allows early detection and fixing of bugs, reducing the cost and time required for debugging later in the development process.
- **Code reliability:** Testing ensures the code behaves as expected and provides reliable results under different conditions.
- **Documentation:** Test cases serve as documentation for how different parts of the code are expected to work/to be used. They help other developers understand the intended behavior of functions and classes.

Unit testing in C++

In C++, unit testing often uses frameworks like Google Test, Catch2, or CTest itself (from the CMake ecosystem).

Here's a simple example using `gtest`:

```
#include "mylibrary.h"
#include "gtest/gtest.h"

TEST(MyLibrary, AddTwoNumbers) {
  EXPECT_EQ(add(2, 3), 5);
}

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

In this example, we test the `add` function from the `mylibrary` module.

Testing numerical code: special considerations

Floating-point comparison:

```
EXPECT_EQ(compute_value(), 3.14159);          // ❌ Exact comparisons are bad practice.
EXPECT_NEAR(compute_value(), 3.14159, 1e-5); // ✅ Use tolerance-based comparison.

// Or compute relative error:
const double result = compute_value(); const double expected = 3.14159;
EXPECT_LT(std::abs(result - expected) / expected, 1e-6); // Test with different tolerances.
```

Testing convergence:

- Test that iterative methods actually converge.
- Verify convergence rates match theoretical expectations.

Regression testing with reference solutions:

- Store reference results from validated runs.
- Compare new results against reference solutions.

Test-Driven Development (TDD)

TDD is a software development approach where tests are written before the actual code. The cycle is writing a test, implementing the code to pass the test, and refactoring.

Advantages

TDD encourages modular and testable code, ensuring all parts of the codebase are covered by tests. It also starts by thinking at how code should be used (bottom-up strategy), possibly guiding the design of the interface exposed.

Process

1. Write a test defining a function or improvements succinctly.
2. Run the test to ensure it fails, showing it doesn't pass.
3. Write the simplest code to make the test pass.
4. Run the test and verify it passes.
5. Refactor the code for better structure or performance.

TDD example

1. Write test:

```
TEST(VectorOps, DotProduct) {
  const std::vector<double> a{1.0, 2.0, 3.0}, b{4.0, 5.0, 6.0};
  EXPECT_DOUBLE_EQ(dot_product(a, b), 32.0);
}
```

2. Run test → ❌ Fails (function doesn't exist).
3. Implement

```
double dot_product(const std::vector<double>& a,
                   const std::vector<double>& b) {
  //...
}
```

4. Run test → ✅ Passes.
5. Refactor (e.g., use `std::inner_product` for better performance),

Continuous Integration (CI) and testing

- **CI:** Frequently integrating code changes into a shared repository. Automated builds and tests ensure new changes don't break existing functionalities.
- **Benefits:**
  - Early detection of integration issues.
  - Regular validation of code against the test suite.
  - Confidence in the stability of the codebase.
- **Popular CI Tools:**
  - Jenkins
  - Travis CI
  - GitHub Actions
  - GitLab CI/CD

Coverage

Code coverage is a metric used in software testing to measure the extent to which source code is executed during the testing process. It provides insights into which parts of the codebase have been exercised by the test suite and which parts remain untested.

Key concepts

- Code coverage is often expressed as a percentage of **lines of code** that have been executed by tests. You should aim for 70-90% coverage (100% often unrealistic/unnecessary).
- Code coverage can also analyze **branches** or **execution paths** (`if` statements) within the code. This provides a more detailed analysis of the code's behavior. Critical paths should have close to 100% coverage.
- Coverage **doesn't** guarantee any form of correctness (you can have 100% coverage with poor tests).

Coverage with gcov

`GCC` supports program coverage with `gcov`. Compile with `-g -fprofile-arcs -ftest-coverage` and **no optimization**. For shared objects with `dlopen`, add the option `-Wl,--dynamic-list-data`.

Run the code, producing `gcda` and `gcno` files. Use `gcov` utility:

```
gcov [options] source_file_to_examine [or executable]
```

Text files with code and execution counts for each line are created.

Main options of gcov:

- `--demangled-names`: Demangle names, useful for C++.
- `--function-summaries`: Output summaries for each function.
- `--branch-probabilities`: Write branch frequencies to the output file.

`lcov` and `genhtml`: nice graphical tools for `gcov`

The `gcov` output is verbose. With `lcov` and `genhtml`, you get a graphical view:

Compile with `gcov` rules, then:

```
lcov --capture --directory project_dir --output-file cov.info
genhtml cov.info --output-directory html
```

`project_dir` is the directory with `gcda` and `gcno` files. In the `html` directory, open `index.html` in your browser.

Summary

1. **Optimization:** Profile first, optimize hot spots, let the compiler help.
2. **Debugging:** Use static analysers proactively and debuggers reactively.
3. **Profiling:** Essential for finding actual bottlenecks (not guessing).
4. **Testing:** Invest in automated tests; they save time and ensure correctness.

Best practices:

- Measure, don't guess.
- Optimize algorithms before micro-optimizations.
- Use tools appropriately (each has its strengths).
- Test, test, test. Early and often.
- Document assumptions and edge cases.

Remember: Premature optimization is evil, but so is premature pessimization!

Introduction to Python.