TER report

Optimizing application performance through optimizing compilation

Prepared by
Francois Flandin

Supervised by
Pr Sid Touati

Master Informatique
at Université Côte d'Azur
First semester of year 2024-2025

Contents

1 Introduction

The tutoring project titled "Optimizing Application Performance through Compilation Optimization" explores how compilers improve the generated code using various optimization levels: `-O0` , `-O1` , `-O2` , `-O3` , and `-Os` .
The different optimization levels goes from `-O0` , which means no optimization, to `-O3` , which enables the most optimizations, but also `-Os` , which compiles to have the most compact code.
This project is divided into two main phases:

- Experimentation: Compile two programs with four different compilers (`gcc` , `icx` , `clang` , and `ccomp`) at each optimization level, measure execution times over 12 iterations, and plot the data using `R`.
- Optimization Analysis: Study the optimization options enabled by each compiler at each level, to better understand their internal mechanisms.

These steps allow for the evaluation of compilers not only in terms of speed but also by identifying the specific characteristics of each optimization level.

Part I
The experience

Experience's environnement

In this part will be presented the environnement of the experimentations, on which hardware it was done, on which software, etc.

Hardware

In this part, the hardware part of the computer will be explored.
To explore the hardware of a computer on linux, the folder `/cpu` can be explored, it gives us the *topology* of the cpu, like which threads are on each cores, which caches are on each cores, discover what is the cache line size etc...
But there are also a lot of linux tools that allows to check the hardware's specifications of a computer, and basically doing the job for us.
There are graphical tools such as `likwid`, which provides the command `likwid-topology`, or the tool `hwloc`, which provides `lstopo`, a command that gives a beautiful GUI of the computer's hardware's specification.

Model name : 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz

Adress size : 39 bits physical, 48 bits virtual

Cache line size : 64 bytes

Cores : 4

Graphical Topology				
Coeurs	0 4	1 5	2 6	3 7
Cache L1	48 kB	48 kB	48 kB	48 kB
Cache L2	1MB	1MB	1MB	1MB
Cache L3	8 MB			

Table 1: Computer's topology

Software

For the software part, the experiments were done on a lightweight configuration of the computer, where all unnecessary services were disabled, including the graphical interface, this allows the computer to run as baremetal as possible.
Here is a description of all the software elements used in this benchmark :

OS Fedora Linux v40 WorkStation

gcc version 14.2.1 20240912

icx version 2024.2.1.20240711

clang version 18.1.8

ccomp version 3.14

2 Method of experiment

To compare the performance of the different compilers, two programs will be used, the first `matrix_multiply`, is a `C` program that multiplies two matrices to put the result in a third matrix, it is an interesting challenge for optimization because of the presence of 3 nested loops for the calculation. This program will be compiled using `gcc` , `clang` , `icx` and `ccomp` .
The second program is `dijkstra` algorithm using `C++`, the goal behind this is to compare the compilers optimizations when the program has a lot of memory usage, because it is in `C++`, the program will only be compiled with the `C++` versions of `gcc` , `clang` and `icx` , due to the lack of `C++` compilation from `ccomp` .
Next, we can mesure time taken by the program by placing the c-function `gettimeofday()` around the function that we need to calculate, initialisation mesurement is not the main goal, but it can be mesured too. The programs are executed 12 times for each optimization level to make an interesting violin plot using `R`.

3 Results

3.1 Matrix Multiply Results

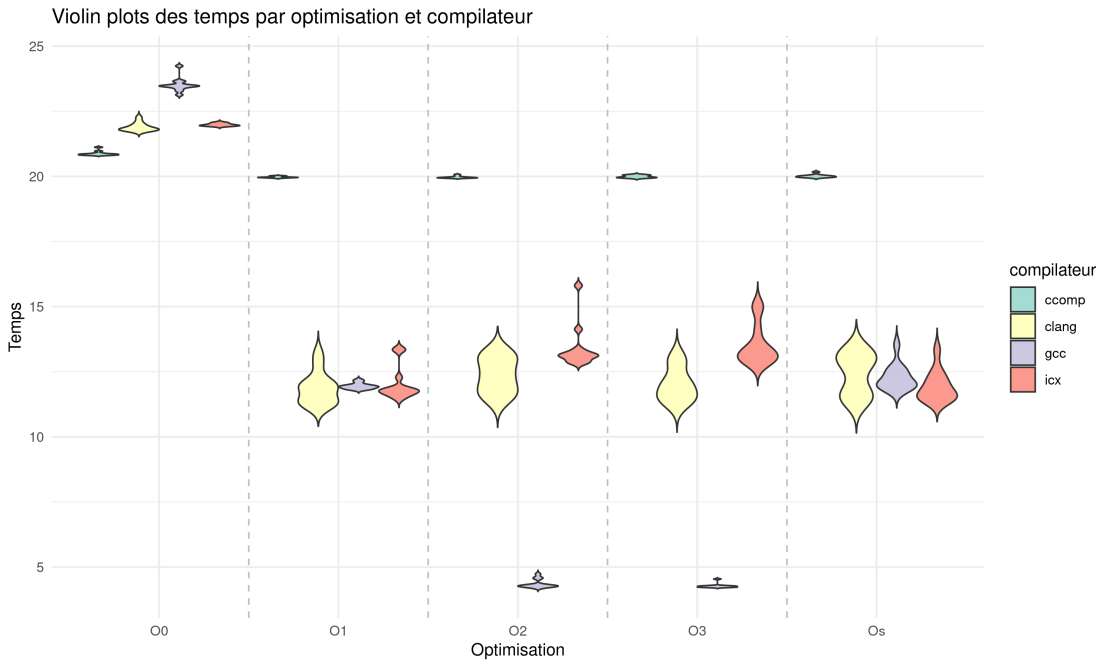

Figure 1: Evolution of the execution time of the program mat_mult.c as a function of compiler and optimization level.

What we notice here is that `gcc` gives the most optimized programs of them all, even by starting as the worst one, `ccomp` is the worst one and can even notice that its execution times are really constant, the violin plots looks the same across all optimizations.

`clang` get really consistent maximum and minimum time across all optimizations from `-O1` to `-Os` , but the violins are really differents one from another. It doesn't offer great performances compared to `gcc` , but it gives a good upgrade from `-O0` , and is slightly better than `icx` .
`icx` is the second worst compiler across the 4 that we analyze, its performances are even degrading with the optimizations levels.

3.2 Dijkstra Results

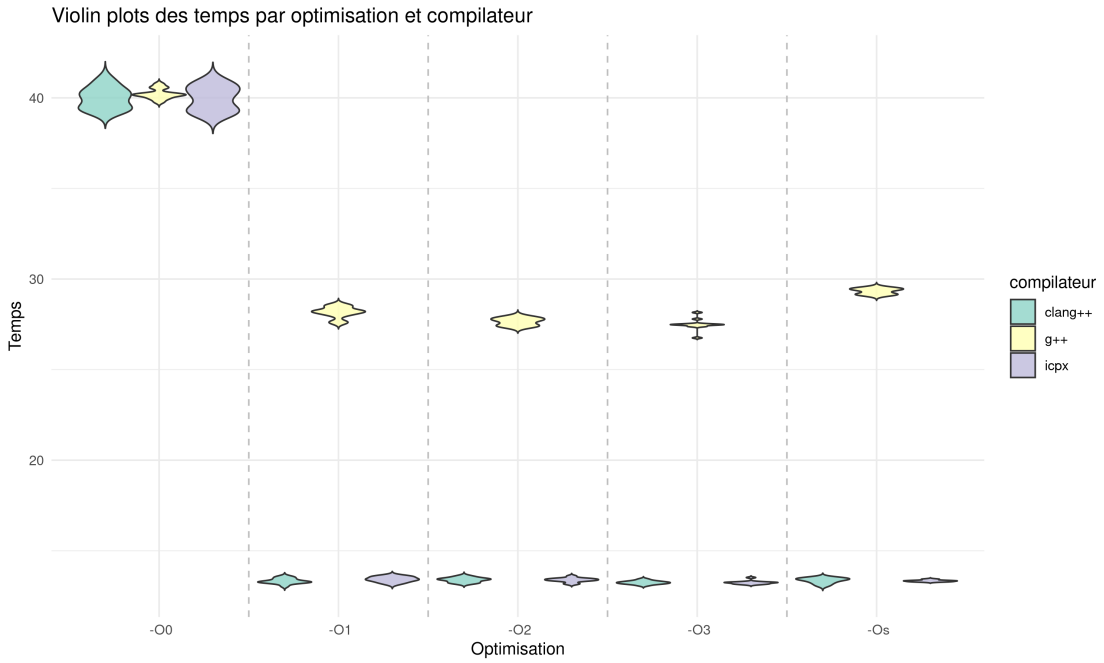

Figure 2: Evolution of the execution time of the program dijkstra.cpp as a function of compiler and optimization level.

What we can notice from these 2 graphs, is how differently `gcc` compares with `clang` and `icx` according to the language used, `C` or `C++`. When using `C`, `gcc` produces way more optimized codes than `clang` and `icx` , whereas in `C++`, `gcc` is the least performant one.
Few experiments were done get some answers to why is it like this.

First, the experiments were made on the `C++` programs, at the compilation level of `-O3` , this will give us some data to work with, and, by using the command `perf record` and `perf report`, we can use material performance counters to get a lot of data on the percentage of time used by a function, we could use profiling applications such as `gprof`, but `perf` allowed us to have more precise data and avoided us to recompile entirely the program.

The results were the following:

	`gcc`	`clang`	`icx`
total time taken	136.37s	146.98s	155.65s
dijkstra usage	$\approx 26s$	$\approx 13s$	$\approx 17s$
init/free usage	$\approx 109s$	$\approx 133s$	$\approx 138s$

What we can get as results from here is that `gcc` is the most optimized in terms of initialisation and memory management, but is the worst one in terms of pure dijkstra calculation.
If we push further the analysis, we can see there are some points in the program where types inference were used, such as `for (auto edge : temp_neighbors)`. `gcc` might not be able to manage this type system as well as `clang` or `icx` , hence the lack of optimizations.

Part II
Compilers

In this part will be presented all the compilers used in this tutorship project, as well as deepening their optimization options for each optimizations levels.

4 gcc

`gcc` is the most popular `C` compiler, it comes in the `GNU Compiler Collection`, which includes compilers for `Fortran`, `C++`, `Go`...

4.1 Optimization options

4.1.1 Method of obtention

`gcc` is really helpful regarding getting informations about what he does, because the compiler has an option for this sole purpose, which is

```
gcc --help=optimizers -Q -On
```

4.1.2 Options introduced by `-O0`

`gcc -O0` was originally meant to disable every optimization flags to give the least optimized code, however, the competition with other compilers led to now have some optimization flags enabled, 53 to be more precise. Not all optimizations will be presented, because the "optimizations" are for the most, analysis options.
A part of the optimizations concerns loop optimizations, with the option `-faggressive-loop-optimizations`, `gcc` enables, as its name suggests, a lot of loop optimizations, such as **loop fusion** and **fission**, **loop unrolling**, **loop peeling**.
Next is the dead code elimination, `gcc` can also remove every line of code that isn't used, this doesn't impact the execution speed, but reduces the size of the generated code.
And finally are the peephole optimizations this option introduces optimizations on a reduced window, optimizing redundant register moves, redundant calculations, etc.
Other enabled options at `-O0` are either analysis options or add error-handling instructions.

4.1.3 Options introduced by `-O1`

`gcc -O1` optimizations add optimizations to the tree that the compiler builds in the intermediate level.
First, there is the dead code and redundancy elimination, with the options `-ftree-dce`, `-ftree-fre` and `-ftree-dse`, `gcc` removes all the **dead code**, computations that produce the same results as well as all **dead stores**, that means that the cpu has less operations to manage, and the code size is reduced.
Secondly, there is value and copy propagation, first, the option `-ftree-ccp` tells the compiler to propagate constant values across conditions when it's possible, it simplifies all the control flow and makes place for further optimizations, and can even help for `branch prediction`.
Secondly, the option `-ftree-copy-prop` propagates values with copy instructions, for example `b = a; c = b;` would be replaced with `c = a;`.
And finally, loop optimizations, there are two main options here to consider, the first, `-ftree-sink`

moves computations outside of the loop when it's possible, it allows to avoid computing the same thing over and over.
The second is `-ftree-sra` for *scalar replacement of aggregates*, it breaks structures and array into individual variable when needed, it can improve register usage and enable even further optimizations. An example of this would be a variable of the struct `Point {int x; int y}` would be transformed from `Point p;` to `int p_x, p_y;`.
At `-O1` , there are also other options to enable inlining of functions, that means that a function call will be replaced by the function's body, which allows for further optimizations.

4.1.4 Options introduced by `-O2`

`gcc -O2` introduces aligning of functions, jumps, labels, and loops, the aligning allows to ensures that the instructions fit at the beginning of cache lines, which can reduce cache misses. Also, aligning labels and jumps allows to decrease branch prediction misses.
`-O2` also enables `-funroll-loops`, which performs loop unrolling when iteration count is known, which allows the compiler to optimize as it wishes.

4.1.5 Options introduced by `-O3`

`gcc -O3` enables also further loops optimizations, such as `loop peeling`, `loop jamming`, `loop unrolling`, `loop interchange`, `loop splitting`.

loop peeling removes special cases in the loop and put them outside of the loop.

loop jamming combines two loops with the same scope/range into a single one, enabling more optimizations, and is generally combined with `loop unrolling`.

loop interchange swap two loops in a nested loop, this allows to put the most changing variable to the consecutive elements in memory, instead of having the most changing variable to the separate elements in memory, it's very useful in programs such as `matrix multiplication`.

loop splitting allows simplification of a loop by dividing it into several loops, it simplifies the loops with the pattern **for (int** $i = 0$**;** $i < 10$**;** $i++$**) if(**$i < 5$**)... else** into two loops, one with the range 0..4 and the other 5..9. It help in reducing dependencies.

4.1.6 Options enabled by `-Os`

`gcc -Os` enables the same options as `gcc -O2` , but removes the aligning optimizations to free some space.

5 icx

It's Intel's compiler for their x86 architecture, it is made to deliver extremely optimized programs on their processor architecture.

5.1 Optimization options

5.1.1 Method of obtention

Everything is available in the online documentation, it's relatively easy to find what we need.

5.1.2 Options introduced by `-O0`

Everything is disabled at `-O0` .

5.1.3 Options introduced by `-O1`

At this level of optimization, `icx` starts by introducing **data flow analysis**, to help gathering data about data flows throughout the whole program, allowing for optimizations.
Then, the compiler can also reorganize code as he wishes with **code motion**, which allows to move non-changing result operations out of loop to avoid recomputing it over and over, and **instruction scheduling**, which allows to change the order of instructions, to hide memory latency for exemple, or doing something while something heavier is done. It is done to reduce CPU stall.

Other optimizations are also enabled, **strength reduction**, that changes heavy operations for cheaper ones, like switching from a multiplication to a bit shift, **test replacement**, that simplifies loops' conditions for example :

```
if (x >= 0 && x < 100) { /* code */ } // base loop
if ((unsigned)x < 100) { /* code */ } // optimized loop
```

And, finally, **split-lifetime analysis**, can split the *lifetime* of a variable into smaller lifetimes, to reallocate the registers when the variable is not used anymore.

5.1.4 Options introduced by `-O2`

At this level of optimization, `icx` introduces more optimizations, like **constant propagation**, or **forward substitution**, which can replace variable with their expressions, for example :

```
int a = 5 + i
int b = a + 3
```

will be replaced by

```
int b = 5 + i + 3
```

These two options complete each other really well, because imagine that the variable `i` was a constant, it would be replaced by its value, hence further optimizing the program.

Additionnal optimizations are introduced, like **loop unrolling**, which reduces the number of iterations by executing multiple iterations within a single loop cycle,**peephole optimizations**, which examines small code fragments to identify and replace inefficiencies, such as redundant instructions or suboptimal sequences, improving code quality. Or even **optimized code selection**, replaces inefficient instructions or code sequences with more efficient alternatives.

There are also code-removing optimizations such as **dead-code elimination**, **dead-store elimination**, and **dead static function elimination** which does not really improves performance but can help reduce the weight put on the CPU.

5.1.5 Options introduced by `-O3`

It enables all the same optimizations as `-O2` but add more aggressive loop transformations such as **loop fusion**, **block-unroll-and-jam**, with these two options, the compiler expands and find the best fusions/jams in the loops and **collapsing-IF-statements**, where multiple successive if are combined into one to reduce branching therefore gaining performances.

6 ccomp

Certified Compiler
It's the only C compiler that is formally verified to produce a code described by the source code, optimized or not. By its formally verified nature, it is the one that produces the most inefficient code. It is generally used for safety critical programs where a bug introduced by compilation could lead to serious problems.

6.1 Optimization options

6.1.1 Method of obtention

Everything is available in the online documentation, it's relatively easy to find what we need.

6.1.2 Options introduced by `-O0`

`ccomp` doesn't enable anything at `-O0` , and the main optimization level `-O` gives the same optimizations as `-O1` , `-O2` , `-O3` , because they are just an alias of that optimization option.

6.1.3 Options introduced by `-O1` ,`-O2` ,`-O3`

There are not a lot of options so here is a list of them

-fcont-prop enables constant propagation, a process that replaces variables with their known constant values throughout the program.

-fcse activates the elimination of common subexpressions, reducing redundant calculations by reusing previously computed values.

-fif-conversion with this option, the compiler decides whether to replace simple if-then-else statements or the conditional operator (?:) with conditional assignements.
The heuristic-based approach selectively optimizes small and balanced expressions, provided the target architecture supports a suitable conditional move instruction.

-finline enables or disables the inlining of functions, potentially improving performance by replacing function calls with the actual function code.

-finline-functions-called-once specifically inlines functions that are called only once, reducing overhead while maintaining efficiency.

-fredundancy activates the elimination of redundant computations and unnecessary memory stores, improving execution time by avoiding repetitive operations.

-ftailcalls optimizes function calls in tail position, which can enhance performance by reusing the current function's stack frame.

-ffloat-const-prop 2 enables full propagation of floating-point constants, ensuring arithmetic is performed in IEEE double precision format with round-to-nearest mode.

6.1.4 Options introduced by `-Os`

The documentation doesn't give a lot of informations about it. Just the usual "optimizes for code size".

7 clang

It's a compiler made by the LLVM Developer Group, it can replace `gcc` by supporting most of the option flags of `gcc` .
The objective behind this compiler is to create a compiler that allowed: first, better diagnostics, which allows to debug more easily, second, to be seperated from the GNU licence, which forced softwares to be integrated to said licence and therefore having to opensource proprietary software, third, to have a nimble compiler that is simple and easy to develop and maintain.

7.1 Optimization options

7.1.1 Method of obtention

`clang` doesn't have a clear documentation about what it enables with each compilation level, however, it has 2 commands that can give what is enabled with each optimization level, there are 2 commands because of the way clang works.
The compiler is divided in two parts, `clang` , the part where the intermediate representation (IR) is made, and `opt`, the part that optimizes said IR. The first command is used to know which optimizations are enabled by `opt`.

```
llvm-as < /dev/null | opt -On -disable-output -debug-pass-manager
```

The second command gives the optimizations added by `clang` itself

```
diff -wy --suppress-common-lines \
    <(echo 'int;' | clang -xc - -o /dev/null -\#\#\# 2>&1 | tr " " "\n" | grep -v /tmp) \
    <(echo 'int;' | clang -xc -On - -o /dev/null -\#\#\# 2>&1 | tr " " "\n" | grep -v /tmp)
```

7.1.2 Options enabled at `-O0`

`clang` like the other compilers doesn't enable a lot of optimizations, the optimizations are called **AlwaysInlinerPass**, which inlines functions marked with the `always_inline` attribute, **CoroConditionalWrapper**, which is related to coroutines support in C++, which is a function that can be paused and resumed, enabling asynchronous and cooperative task management. Finally, **VerifierPass** ensures that the LLVM Intermediate Representation generated by the compiler is correct and coherent with the original program.

7.1.3 Options enabled at `-O1` , `-O2` , `-O3` , `-Os`

Like for `ccomp` , they enable all the same optimizations.

`clang` introduces a lot of optimizations on the functions.
It applies to functions the user-defined attributes such as `inline`, or automatically applies some attributes like `pure`, `const`, `noreturn`, etc. which will be helpful for other optimizations, such as inlining or dead code optimization.
`clang` also manipulate the functions arguments by removing *"dead"* arguments, which simplifies function calls and memory usage. It also analyzes function calls to propagate information about the possible values of arguments or return values. This enables more precise optimizations, especially for indirect calls.
It can also propagate function attributes (e.g., `noreturn`, `readonly`) through the call graph in reverse post-order, enabling interprocedural optimizations by analyzing functions and their dependencies. And, by analizing said call graph, it collects profiling information, which can be

used to guide inlining or other optimizations.

These optimization levels also introduces more common optimizations, the first is **ConstantMergePass**, which merges duplicate constant data, such as identical string literals or constant arrays, to save memory and reduce code size. The second one is **GlobalDCEPass** which removes unused global variables, functions, or other entities which reduces the size of the generated code. The last one is **GlobalOptPass**, which simplifies, removes, or transforms global constructs to improve runtime performance and reduce code size.

References

Intel Corporation. Intel® oneapi dpc++/c++ compiler developer guide and reference, 2024. URL https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2024-2/o-001.html.

Xavier Leroy. The compcert c verified compiler, documentation and user's manual. version 3.14, 2024. URL https://compcert.org/man/manual003.html#sec26.