Code-Optimization

By Thomas Lienbacher (12113690) and Frederick Knauder (12005333)

Table of contents

- Introduction
- Branchless code
- Loop unrolling
- OpenMP
- Single Instruction Multiple Data (SIMD)
- Guideline for practical examples
- Sources

Introduction

Code optimization is the process of improving the performance of a computer program by making changes to the source code that do not alter its functionality. This can include techniques such as reducing the number of instructions executed, reducing memory usage, and improving algorithm efficiency. The goal of code optimization is typically to make the program run faster, use less memory, or both. This report focuses on four key strategies: branchless code, loop unrolling, OpenMP, and Single Instruction Multiple Data (SIMD) parallelization.

Branchless code

Branchless code is a technique that involves removing branches from the code and replacing them with mathematical operations in order to improve performance. Branches in code can be expensive, as they require the processor to predict the outcome of the branch and then execute the appropriate instructions. When the prediction is incorrect, the processor must flush the pipeline and start over, leading to a performance penalty known as a branch misprediction.

For example, instead of using the branch instruction `if (x < 0) y = -1; else y = 1;`, the branchless equivalent would be `y = (x >> 31) | 1;`. This shifts the sign bit of `x` into the least significant bit and ORs it with 1, resulting in -1 for negative `x` and 1 for non-negative `x`. This eliminates the branch instruction and improves performance.

While branchless code can improve performance, it is not always the best choice. The code can become more difficult to read and understand, and the performance benefits may not be significant in all cases. Additionally, some compilers are able to optimize branches and improve performance, so the use of branchless code should be carefully considered on a case-by-case basis.

In conclusion, branchless code is a technique that can be used to improve performance by eliminating branches from the code and replacing them with mathematical operations. The bitwise operations and ternary operator are some common ways to achieve this. However, it is important to note that the use of branchless code should be weighed against the potential negative impact on code readability and the potential for compilers to optimize branches.

Loop unrolling

Loop unrolling is a technique that increases the amount of parallelism in a loop by duplicating the loop body and incrementing the loop index by a larger value. This can improve performance by

reducing the number of branches and cache misses that occur during the execution of the loop.

When a loop is executed, the processor must repeatedly fetch the instructions for the loop body from memory and execute them. This can lead to cache misses, as the instructions may not be in the cache when they are needed. Loop unrolling can reduce the number of cache misses by increasing the amount of instructions that are executed before the loop index is incremented, thereby reducing the number of times the instructions need to be fetched from memory.

In addition, unrolling a loop can also reduce the number of branches that are executed. For example, a loop that is executed with a step of one will have a branch instruction at the end of each iteration to check if the loop index has reached the end. Unrolling the loop by a factor of four, for example, will reduce the number of branches by a factor of four.

However, loop unrolling also has some drawbacks. It increases the size of the code and can also cause register pressure, which can lead to a decrease in performance. Additionally, it's not always easy to determine the optimal unrolling factor for a specific loop, as it depends on the specific characteristics of the loop and the target architecture.

In conclusion, loop unrolling is a technique that can be used to improve performance by increasing the amount of parallelism in a loop and reducing the number of branches and cache misses. However, it also has some drawbacks, such as increased code size and register pressure. It's important to carefully consider the trade-offs before deciding to use loop unrolling on a specific loop, and determine the optimal unrolling factor for the specific case.

OpenMP

OpenMP (Open Multi-Processing) is a set of compiler directives and library routines that can be used to parallelize code in a straightforward manner. It is designed to work seamlessly with the C, C++, and Fortran programming languages and allows developers to add parallelism to their code without the need for explicit thread management or low-level memory management.

OpenMP provides a set of directives that can be used to specify how a loop or section of code should be parallelized. For example, the `parallel for` directive can be used to parallelize a loop, while the `parallel sections` directive can be used to parallelize a section of code. OpenMP also provides a set of library routines, such as `omp_get_num_threads()` and `omp_set_num_threads()`, that can be used to query and set the number of threads that are used for parallel execution.

It has several advantages over other parallel programming models. It is easy to use and does not require explicit thread management or low-level memory management. It also provides a simple and intuitive programming model, as all threads share the same memory space.

In conclusion, OpenMP is a set of compiler directives and library routines that can be used to parallelize code in a straightforward manner.

Single Instruction Multiple Data (SIMD)

SIMD, or Single Instruction Multiple Data, is a technique used in computer architecture and programming to process multiple data elements in parallel using a single instruction. This allows for significant improvements in performance, particularly in applications such as video processing, image processing, and scientific simulations. SIMD is implemented in modern processors with special processor instructions and registers that can operate on multiple data elements at once. These instructions and registers are often called SIMD or vector units. The most common SIMD instruction sets are MMX, SSE, AVX, and AVX-512.

One of the key advantages of SIMD is that it allows for a significant increase in performance by reducing the number of instructions required to process a large amount of data. This is because SIMD instructions can operate on multiple data elements at once, rather than having to process each element individually. This can greatly reduce the number of clock cycles required to complete a task, resulting in a significant performance boost.

Another advantage of SIMD is that it can help to reduce the number of memory accesses required to process data. This is because SIMD instructions can operate on data stored in registers, rather than having to access memory for each data element. This can reduce the number of cache misses, leading to a further increase in performance.

To take advantage of SIMD, you need to write specific code that can be vectorized. This means that the code needs to be written in a way that allows the compiler to recognize and take advantage of the parallelism in the data. Some programming languages such as C++ and Fortran have built-in support for vectorization, and libraries such as OpenMP and OpenCL can also be used to vectorize code.

In conclusion, SIMD is a powerful technique that can significantly improve the performance of applications by allowing the processor to operate on multiple data elements in parallel using a single instruction. This can lead to significant performance gains, particularly in applications such as video processing, image processing, and scientific simulations. However, the programmer needs to write specific code that can be vectorized to take advantage of this technique.

Guideline for practical examples

This repository contains four CMake projects. One for each of our topics. These projects can easily be opened using Jetbrain CLion. All projects were compiled using GCC 12 on a 11th gen Intel i7 running Fedora Linux 36. Your CPU must support up to AVX512 in order to compile these projects and your host system should have the OpenMP library installed.

Use this command to check what extensions your CPU supports.

```
$ cat /proc/cpuinfo
```

If you are using Fedora or other RPM based distros you can use the following command to install OpenMP libraries:

```
$ dnf install libgomp
```

Please remember to add `-DCMAKE_BUILD_TYPE=Release` when compiling.

Sources

- https://dev.to/jobinrjohnson/branchless-programming-does-it-really-matter-20j4

- https://en.algorithmica.org/hpc/pipelining/branchless/
- https://jaredgorski.org/notes/branchless-programming/
- https://www.geeksforgeeks.org/loop-unrolling/
- https://de.wikipedia.org/wiki/Loop_unrolling
- https://de.wikipedia.org/wiki/OpenMP
- https://tildesites.bowdoin.edu/~ltoma/teaching/cs3225-GIS/fall17/Lectures/openmp.html
- http://jakascorner.com/blog/
- https://en.wikipedia.org/wiki/Single_instruction,_multiple_data

- https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
- https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf
- https://doc.rust-lang.org/stable/core/arch/x86/index.html
- https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
- https://godbolt.org/