High Performance Computing

Sequencial Code Optimizations

Στολτίδης Αλέξανδρος, Κουτσούκης Νικόλαος

November 1, 2021

Contents

1 Introduction

High performance computing is the ability to process data and perform complex calculations at high speeds. To achieve this we design Computer Clusters that take advantage of each computer's multiple CPUs and GPUs. Parallel and Concurrent Algorithms are considered the cornerstone of HPC since they are essential to harvest this processing power. Before we explore these algorithms we must first understand how also Sequential Code Execution can affect the performance of a system. In this homework we were assigned a Sequential Image Processing Algorithm that detects edges on an image with the use of a Sobel Filter and our task was to apply a few code changes that would hopefully optimize the code execution. Then we have to examine and analyze the performance gain for each of these optimizations.

2 System Characteristics

2.1 CPU and Caches

AMD Ryzen 9 3900X 12-Core Processor

1. L1 Instructions: 384 KiB
2. L1 Data: 384 KiB
3. L2: 6 MiB
4. L3: 64 MiB

2.2 Memory (RAM)

- Size: 16 GiB
- Frequency: 3200 MHz

2.3 Operating System

- OS Version: Ubuntu 20.04.2 LTS
- Kernel Version: Linux 5.11.0-38-generic

2.4 Compiler

- Compiler: icc (ICC) 2021.4.0 20210910

3 Code Optimizations

3.1 Loop Interchange

Arrays and Multi-Dimensional Arrays in C are stored horizontally. When a Memory Address is accessed (Read or Write) the CPU searches L1, L2 and L3 Caches and then the Main Memory (RAM or in the worst case scenario the Disk). If the requested data is not on the Cache then many more CPU Cycles are needed to access the data in the Main Memory. When the data is retrieved from Main Memory it is stored into the Cache in order to be retrieved more quickly next time. This is called a Cache Miss, since the data was not in the Cache and a Main Memory access was necessary. On a Cache Miss the CPU also retrieves a block of data that was next to the requested data since this block might be used later (Cache Locality). On the other hand a Cache Hit can occur and the data can be easily retrieved from Cache. A Cache Hit is always faster than a Cache Miss. So, should you iterate multi-dimensional arrays vertically or horizontally? If you iterate arrays vertically many Cache Misses will occur since there will be a lot less Cache Locality (a Cache Miss will pull a block of the following Columns into the Cache but the program tries to access a whole new Row which might not be into the Cache). If you iterate arrays horizontally fewer Cache Misses will occur since data is already in

Cache. An example of vertical versus horizontal iteration shows the Cache Misses and Hits with Red and Green color respectively.

Vertical Iteration

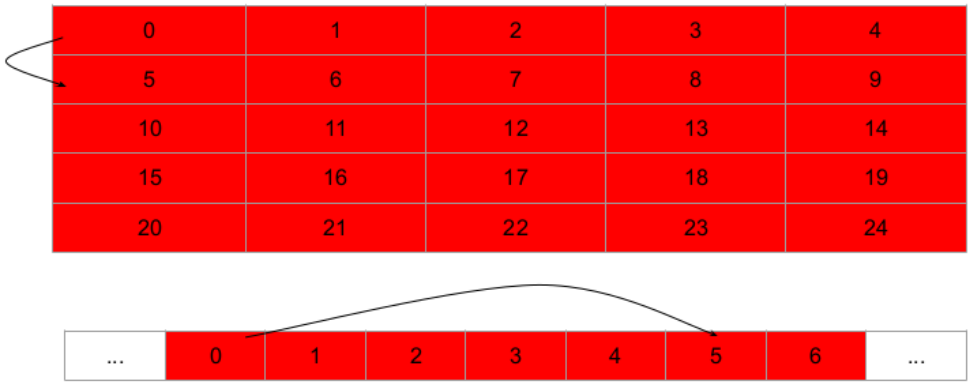

Figure 1: 2D Array in Memory with Vertical Iteration (Jumb in Memory $\Longrightarrow$ $CacheMisses$)

Horizontal Iteration

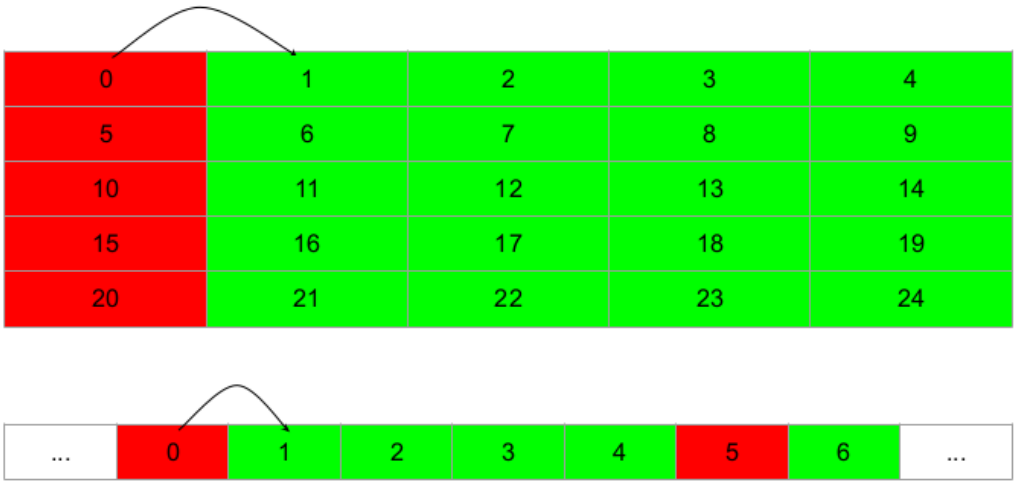

Figure 2: 2D Array in Memory with Horizontal Iteration (Jump to Nearby Address $\Longrightarrow$ $CacheHits$)

3.1.1 Code Changes

Changing Array Iteration by Interchanging Loops (Interchange i with j). These changes are applied in the functions below.

1. **convolution2D()**: In the Input-Operator Arrays Convolution Loop
2. **sobel()**: In Main Computation Nested Loop

With these changes there are no jumps in Memory for each iteration hence a lot less Cache Misses. This results in an immense performance gain since data is retrieved in fewer CPU Cycles from Cache instead of Main Memory.

3.2 Loop Unrolling

When Loop Unrolling is applied the body of the loop contains many more instructions but the iterations are a lot less. This might cause a lot more Cache Misses in the L1 Instruction Cache. On the other hand, in each iteration there is at least one conditional statement that needs to be checked in order for the loop to be executed. By reducing the iterations the conditional statements decrease as well. A conditional statement is somewhat "expensive" since each one takes at least 2 CPU Cycles to complete (Assuming that Branch Prediction is Disabled or False).

3.2.1 Code Changes

Loop Unrolling was applied in the functions below to eliminate some conditional statements and hopefully decrease the execution time. Loop unrolling also helps the compiler optimize more instructions since it knows more about the body of the loop.

1. **convolution2D()**: Since the body of the loop would only be executed 9 times the loop could be completely replaced with just the body of the loop. To do this we must replace i and j with their "fixed" values that would occur in each iteration. In this case there are no conditional statements since there is no loop.

2. **sobel()**: The Array has 4096^2 positions and the loop goes from 1 to 4095 (4094 iterations).

   (a) Initializing output[][]: By increasing the step of the iterator from one to two the iterations are reduced by half. In each iteration we initialize the output[i][0], output[i][SIZE - 1], output[i+1][0] and output[i+1][SIZE - 1] and then this repeats from output[(i + step)][...] etc.

   (b) Main Computation (Nested Loops): Here we tried a few different things. At first, we tried to increase both the step of the i and the j iterators. With this approach we examine 4 positions on the array ([i][j], [i][j+1], [i+1][j] and [i+1][j+1]). This results in a lot of Cache Misses when trying to jump from memory address at [i][j+1] to [i+1][j]. To counter this we examine only two positions ([i][j] and [i][j+1]) which increases Cache Hits and Performance. Here is a figure for the performance of these two methods.

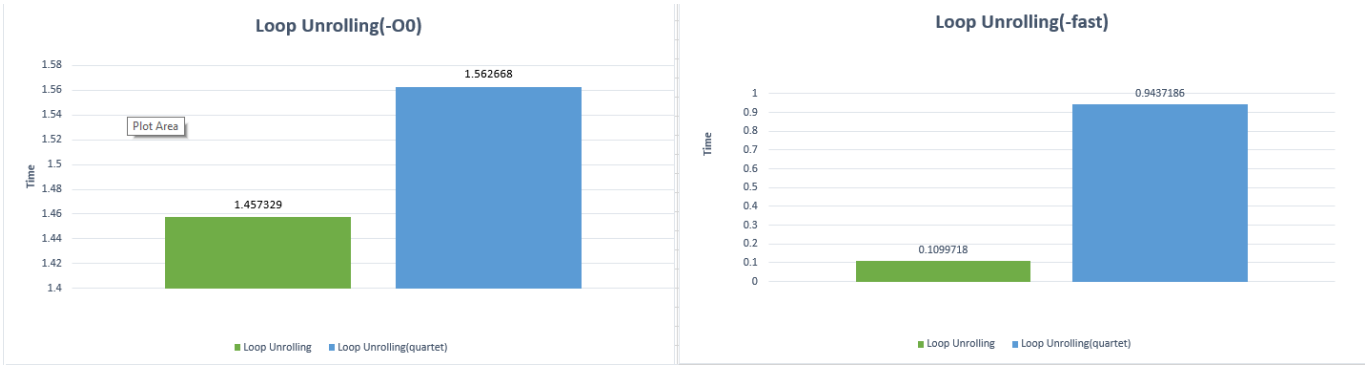

Figure 3: Quartet is when accessing 4 Cells of the Array

3.3 Loop Fusion

Just as Loop Unrolling, Loop Fusion puts a lot more load on the L1 Instruction Cache. In addition, the loops that were merged might need to access different blocks of data that are not in the cache. If large chunks of data are pulled into the Cache then the program might spend more time retrieving data than actually doing calculations. On the other hand, Loop Fusion skips a lot of conditional statements which might give a small performance gain if we are careful with our memory management.

3.3.1 Code Changes

Loop Fusion was applied in Sobel Function in order to decrease execution time.

1. **sobel()**: The Main Computation and PSNR loops were merged together since they are both nested loops starting from i = 1 and j = 1, going till SIZE - 1 with the same step. PSNR must be calculated after both res from position [i][j] and [i][j + 1] are calculated. This lets both input[][] (to calculate res) and output[][] (to calculate PSNR) stay in Cache without a lot of Cache Misses.

3.4 Function Inlining

In order to increase performance even more we try to skip some low-level operations that the compiler does that we as programmers don't see. Each time we call a function a Stack Frame is build in memory. After the end of the function the Stack Frame is removed and the code continues execution after the function call. Building and destroying Stack Frames, pushing function parameters on stack, moving instructions from memory to the L1 Instruction Cache and jumping back to caller after the end of the function takes a lot of time. By replacing the function call with the function content itself the execution skips this unnecessary overhead.

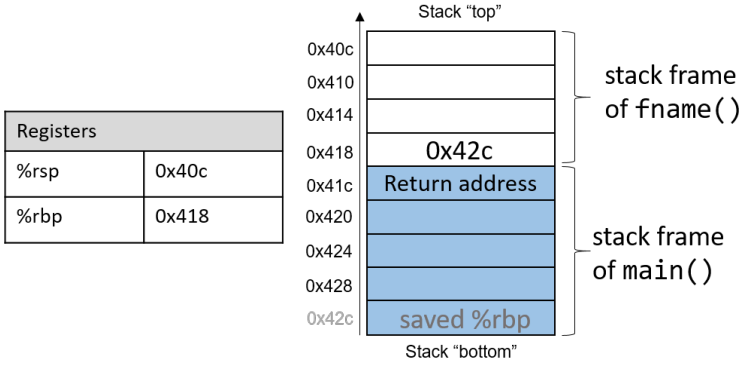

Figure 4: Stack in x64

3.4.1 Code Changes

To skip the unnecessary overhead we completely remove the convolution2D() function call from sobel(). To achieve what convolution2D() does we hard-code its body every time there is a convolution2D() function call. With this approach, we skip unnecessary $rip jumps and memory accesses. On sobel() the changes are shown below.

1. **sobel()**: Whenever there is a conolution2D() call the code is replaced by the body of the convolution2D() with
   (a) **posy = i**
   (b) **posx = j**
   (c) **input = input**
   (d) **operator = horiz_operator** or **vert_operator**

3.5 Loop Invariant

On a loop some variables should be set only once since on each iteration since their value does not change. So these variables must be outside the loop body. If the variables that are independent of the loop are removed from the body then many unnecessary instructions and operations can be skipped

3.5.1 Code Changes

In order to boost performance even more we can skip some operations that always return the same result. These changes can be applied in sobel().

1. **sobel()**: The Array has 4096^2 positions and the loop goes from 1 to SIZE - 1 = 4095.
   (a) Initializing output[][]: In each loop SIZE - 1 is recalculated to be checked. However SIZE - 1 is always the same and can be calculated only once.
   (b) Main Computation (Nested Loops): Here SIZE - 1 is calculated both in the outer and in the inner loop. Recalculating the same value for 4095^2 times is an unnecessary overhead that can be easily skipped if the SIZE - 1 is calculated only once.

3.6 Common Subexpression Elimination

The idea here is the similar to Loop Invariant, same operations that are calculated more than once with the same result can be calculated just once to be used later. This is very useful in sobel() since position on array is calculated more than once to access a certain cell on the array.

1. **sobel()**: A list of expressions that can be calculated once per iteration are shown below
   (a) $row = i * SIZE$
   (b) $previous\_position = row - SIZE + j$
   (c) $current\_position = row + j$
   (d) $next\_position = row + SIZE + j$

3.7 Strength Reduction

The CPU doesn't take the same CPU Cycles for all Arithmetic Calculations. Some operations are faster than others. If we replace these operations with cheaper ones we increase performance by also decreasing the load on the ALU of each core.

3.7.1 Code Changes

Replacing pow() will speed up the execution of our program since pow() does a lot more arithmetic operations under the hood than a simple multiplication of a number with itself.

1. **sobel()**: Each time there is a pow(number, 2) function call we replace it with number * number[0]. We also tried to write a quicker sqrt() that finds the approximate square root of a number in order to optimize code even more. A figure of the already written sqrt() versus our sqrt() is shown below

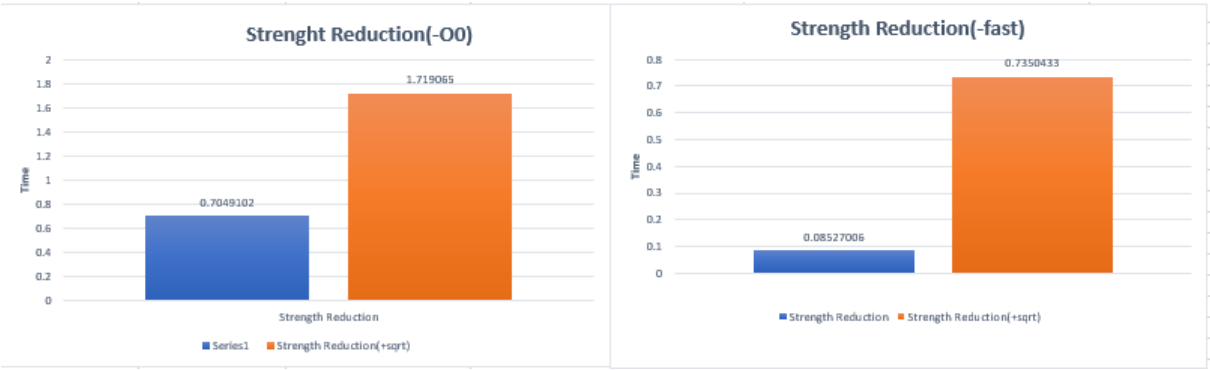

Figure 5: DIY Square being a lot slower

3.8 Helping the Compiler

Some keywords can help the compiler to decide how to treat each variable based on some low-lever hardware principles. Such keywords are:

1. **const**: Content of memory address won't change by the function call.
2. **register**: Put this variable on a register in order to reduce memory r/w operations.
3. **restrict**: Only a single pointer points to this address so compiler can skip some error checking.

These keywords where used for

const	register	restrict
input[][]	PSNR t i j p res size_invariant row position previous_position current_position next_position	Unused

[0]This can also fit into the category of Function Inlining but we thought that pow() does a lot more than a simple multiplication under the hood, so we assume that pow() is just an expensive and more extensive implementation of a multiplication

4 Performance Analysis

We approached performance optimizations from two scopes. We started by trying to optimize code by not applying compiler optimizations. When we finished optimizing we observed that some optimizations made the code even slower when enabling compiler optimizations. We then excluded the optimizations that made the code slower.

4.1 Optimizations

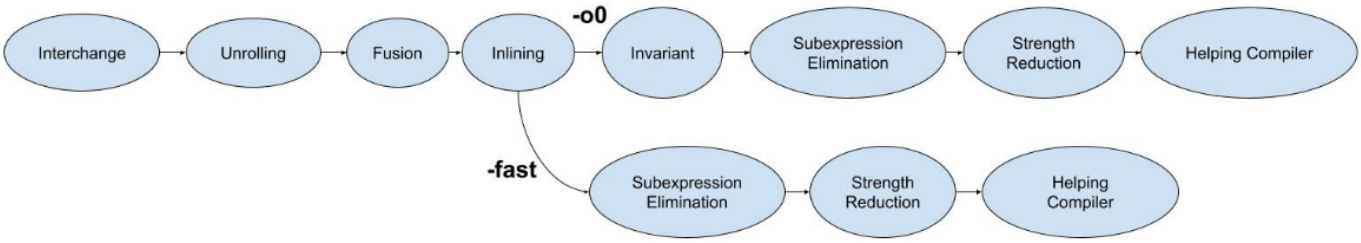

Figure 6: When Compiler Optimizations are Enabled Loop Invariant is Disabled

4.2 Statistics

4.2.1 Optimizations when Compiler Optimizations are Disabled (-o0)

Here we **optimized for the -o0 version** without inspecting the -fast stats. Even though Loop Invariant Code Motion improves the -o0 build, the -fast build runs 80% slower. This might be due to compiler replacing SIZE - 1 with it's exact value instead of reading it from memory or register (we stored SIZE - 1 in a variable).

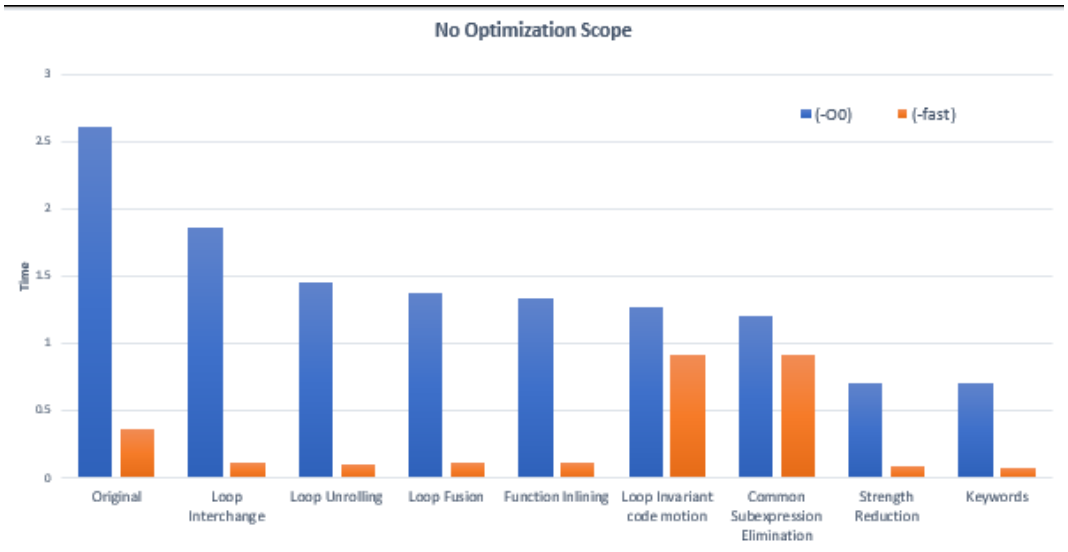

Figure 7: Loop Invariant 80% slower on -fast so it must be excluded

4.2.2 Optimizations when Compiler Optimizations are Enabled (-fast)

We now **optimized for the -fast version** without inspecting the -o0 stats. Even though -fast decreases execution time the -o0 doesn't improves as much as previously.

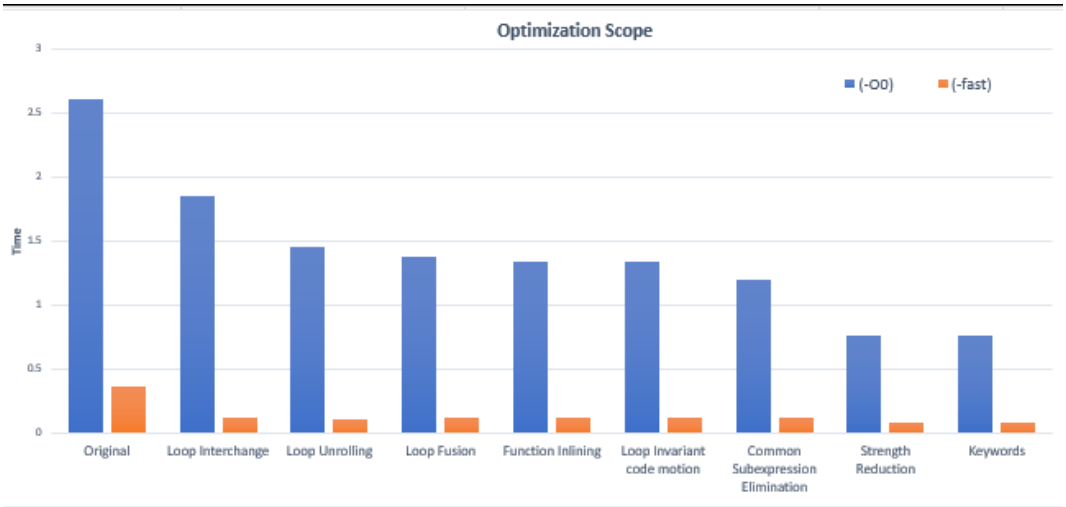

Figure 8: Loop Invariant was not applied since it reduced -fast performance

5 Optional Optimizations

5.1 Helping the preprocessor

By copying function declarations and definitions that are located in header files like stdio.h, math.h etc we improve performance. Since include statements are removed the preprocessor doesn't have to copy large chunks of code in the main file. This saves a lot of time.

5.2 Approximate Value of Square Root

Finding a square root is a very expensive operation. Most times though we do not care about the exact square root of a number. We can find an approximation of a square root with an algorithm like the inverse of Quake III.

5.3 Threading

Finally each independent iteration of a loop can be executed by a different processor. Taking advantage of Multi-Threading might boost performance even more.

6 Contact Information

Στολτίδης Αλέξανδρος (2824): stalexandros@uth.gr
Κουτσούκης Νικόλαος (2907): nkoutsoukis@uth.gr