Master Thesis

Cache Model Plugin for Memory Hierarchy Aware Programming

Author(s):
Fompeyrine, Patrick

Publication Date:
2020

Permanent Link:
https://doi.org/10.3929/ethz-b-000431675

Rights / License:
In Copyright - Non-Commercial Use Permitted
Cache Model Plugin for Memory Hierarchy Aware Programming

Master Thesis
Patrick Fompeyrine
May 12, 2020

Advisors: Prof. Torsten Hoefler, Dr. Tobias Gysi, Dr. Tobias Grosser
Department of Computer Science, ETH Zürich
Abstract

The cost of data movement is a non-trivial matter with today's memory hierarchies. It works behind the curtain of the compiler and computer architecture which makes a prediction difficult. Thus, code developers can often reason about the cost of computation but fail when it comes to memory access costs. Nevertheless, both factors contribute to the effective runtime and are subject of optimizations in high-performance computing. We present a cache model plugin that provides valuable memory access information for Visual Studio Code, a modern source code editor. This tool can help programmers throughout the development process to tune their code to the memory hierarchy. It is especially useful in the design phase where flaws can easily be fixed. We close thereby the gap between understanding the cost of arithmetic instructions and memory accesses. Different cache models and simulators have been developed to provide cache miss information. We utilize Haystack in our plugin as it is a lightweight cache model independent of the problem size. In practice, Haystack's response time is often in terms of a few seconds enabling our plugin to deliver on-the-fly information about data movement. Our plugin uses various visualization techniques for presenting these results. On the one hand, simple graphs put the information in a nutshell making cache effects accessible to novice programmers. On the other hand, advanced developers can inspect the raw data from the cache model for a more thorough analysis. Besides, a user can compare different code variants to track optimization progress or perceive factors that affect the memory access cost. As a result, we enable memory-hierarchy aware software development by providing an interactive tool to learn and understand cache effects.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> Introduction</td>
<td>1</td>
</tr>
<tr>
<td><strong>2</strong> Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Cache Misses</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Haystack</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Visual Studio Code</td>
<td>5</td>
</tr>
<tr>
<td>2.4 Use Cases</td>
<td>5</td>
</tr>
<tr>
<td><strong>3</strong> Implementation</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Overview</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Visual Studio Code Extension</td>
<td>8</td>
</tr>
<tr>
<td>3.2.1 Identifying SCoPs</td>
<td>10</td>
</tr>
<tr>
<td>3.2.2 Showing Data Reuse Potential</td>
<td>11</td>
</tr>
<tr>
<td>3.2.3 Raw Data of Cache Model</td>
<td>11</td>
</tr>
<tr>
<td>3.2.4 Comparing Different Code Variants</td>
<td>12</td>
</tr>
<tr>
<td>3.2.5 Highlighting Costly Accesses</td>
<td>13</td>
</tr>
<tr>
<td>3.2.6 Performance Model</td>
<td>14</td>
</tr>
<tr>
<td>3.2.7 Cache Miss Curve</td>
<td>18</td>
</tr>
<tr>
<td>3.2.8 Tuning the Tile Size</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Software Architecture</td>
<td>20</td>
</tr>
<tr>
<td>3.3.1 Clangd</td>
<td>20</td>
</tr>
<tr>
<td>3.3.2 Polly</td>
<td>21</td>
</tr>
<tr>
<td>3.3.3 Haystack</td>
<td>22</td>
</tr>
<tr>
<td>3.3.4 Repository Pattern</td>
<td>23</td>
</tr>
<tr>
<td><strong>4</strong> Evaluation</td>
<td>25</td>
</tr>
<tr>
<td>4.1 Optimization Story</td>
<td>25</td>
</tr>
<tr>
<td>4.1.1 Sobel Operator</td>
<td>25</td>
</tr>
</tbody>
</table>
The amount of data processed nowadays is increasing constantly. As an example, the size of Facebook’s social graph is many petabytes [12]. This requires efficient algorithms that are highly optimized not only in terms of computational complexity but also in terms of data movement. Most programmers are able to estimate the computational costs of their algorithms. Moreover, they have a good intuition for comparing different code variants and find the one which performs the least computations. Yet, the focus of optimizations shifts more and more toward data-locality and reducing the memory access time. A big part of the developers has problems when selecting an optimal tile size or when judging if loop fusion reduces the number of cache misses. Basically, we lack the awareness of the cache effects that affect the performance of data movement.

Data-locality is very important to attain peak performance. A single memory access can take from 100ns to 300ns if data resides in DRAM but only 0.5ns if the data is in the L1 cache. Therefore, developers have to efficiently use the cache hierarchy to benefit of such time savings. However, memory optimizations are often applied at the end of the development cycle. Getting rid of fundamental design flaws may be an arduous task at this stage. We believe a tool that visualizes memory access information in real-time can guide developers to avoid such shortcomings already in the design phase. In addition, the results of the tool could be used for accurate model-driven memory tuning.

In this work, we present a cache model plugin for Visual Studio Code [9] (VS Code), a modern source code editor. It provides programmers with details about the memory access pattern during the development process. This allows them to react to cache effects as early as possible. We use existing features of VS Code to supply the developers with basic information. For more details that are beyond the basic functionality of the editor, we utilize a webview provided by VS Code. With all the tools that are available for web


1. Introduction

application developers, we implement responsive and simple visualizations. The data for the VS Code annotations and visualizations is provided by Haystack [5], an analytical cache model. Haystack computes cache information of statically analyzable programs such as cache hits and misses. We have integrated Haystack into the LLVM Project [8] which is a collection of modular and reusable compiler and toolchain technologies. The most popular compiler for C, C++ and Objective-C code that is built on LLVM is Clang [2]. It is leveraged by the easily into VS Code integrated Clangd [3] which uses a Language Server Protocol (LSP). LLVM also contains Polly [4], a high-level loop and data-locality optimizer. Polly scans the source code for Static Control Parts (SCoP) which are the basis of the cache miss analysis done by Haystack.

We also show an example application of the VS Code extension by optimizing an implementation of the Sobel filter [7]. First, we come up with a naive variant that ignores data-locality and has a high run-time. Bit by bit, we improve the algorithm by using various optimization techniques such as loop fusion or tiling. At each step we analyze the memory access pattern with our plugin to reason about the performance gain by comparing the data to the results of the previous implementation. Moreover, we let us guide by the tool during the process of finding new improvements. In the end, we present an optimized Sobel operator for the underlying hardware. This exemplary optimization story illustrates the application and usefulness of on-the-fly cache miss information during the development process.

To sum up, our key contributions in this work are:

- We describe the integration of Haystack in the LLVM project.
- We illustrate the design of a Visual Studio Code extension featuring memory access information.
- We outline various visualization techniques that reflect the underlying data as well as possible. Then, we reason about their importance in the day-to-day workflow of a performance engineer.
- We evaluate our plugin and illustrate its application as guidance for optimizations. We implement a naive version of an image processing algorithm and continually improve the performance using our extension.
Chapter 2

Background

In this chapter, we start by explaining the different types of cache misses (§ 2.1). Afterward, we present the used cache model (§ 2.2) and source code editor (§ 2.3) for which we implemented an extension. At last, we demonstrate the different use cases (§ 2.4) for such an extension.

2.1 Cache Misses

A cache miss is an unsuccessful attempt to read or write a data block in a cache. If the memory hierarchy consists of several layers of caches, a miss causes that the data is fetched from the cache of the next level. This process repeats until either a cache line contains the requested data or the main memory has to deliver it. The result of a cache miss is a longer memory access time making them an interesting subject for performance optimizations.

There are different kinds of caches used in today’s Central Processing Units (CPU) such as instruction caches, data caches or unified caches. In our work, we focus on the data caches as the programmer has a direct influence on them by changing the code. For simplicity, we use the term cache when we talk about data caches.

According to Hill [6], there are three types of cache misses that can be distinguished. We shortly discuss each one of them.

Compulsory Misses

This kind of cache misses is caused by accessing a cache line for the first time. They are always present when handling data and are oftentimes impossible to prevent. However, algorithms sometimes use helper data structures that store intermediate results. In this case, the number of compulsory misses can be reduced by omitting such data structures. Although this decreases memory usage as well, it often makes the code more complex and increases
2. Background

the computation time. That is why they are usually added in the first place. For these reasons, compulsory misses are rarely a subject of optimizations that reduce the memory access costs.

Capacity Misses

If a cache line is discarded because the cache cannot contain all data for program execution, a capacity cache miss happens at the next memory access to the same data block. Usually, optimizations target to reduce the capacity misses as they are the easiest to avoid. There are several strategies on how to prevent this type of misses. They aim either to decrease the number of data accesses altogether or to improve the reuse of cache lines. Especially in loops, where the same data is demanded several times, the potential for reuse is huge. An efficient algorithm tries to process the data in chunks that nicely fit in the cache. If done properly, this can eliminate capacity misses.

Conflict Misses

Lastly, a conflict miss can only happen in direct-mapped or set-associative caches. This type of cache maps the data to a specific cache line avoiding the search through the whole cache when reusing it. Even though it is efficient and runs on cheaper hardware, it comes with a price. If two different chunks of data map to the same cache line, we have a conflict miss even if the cache is empty otherwise. This can be countered by dividing the cache into different sets where the data gets mapped to. If the first set has already filled a cache line, the second one can accommodate the data. As a result, there is a trade-off between latency and the number of cache misses.

Conflict misses could be avoided but they are ignored in our VS Code extension. This is since the utilized cache model uses fully associative caches where conflict misses do not appear. Nevertheless, conflict misses are less often a subject of optimizations than capacity misses.

2.2 Haystack

We use Haystack for acquiring cache miss information. It is a lightweight cache model for fully associative caches that use a least recently used (LRU) replacement policy. The evaluation time is independent of the program runtime as it counts the cache misses symbolically and not by executing the code or enumerating all memory accesses explicitly. This symbolic counting is executed twice, once for deriving the stack distance for each memory access and then again to count the accesses with a larger stack distance than the cache size. Theoretically, this approach seems infeasible because of the non-linearities after the initial round of counting. However, Gysi et al. have
2.3. Visual Studio Code

shown that these counting problems are sufficiently linear in practice. The modeling error is evaluated to be below 0.6% on real hardware.

Haystack provides for each memory access the number of compulsory and capacity misses on each level in the cache hierarchy. As it models fully associative caches, conflict misses do not exist which is a weakness of the model. But as we have already stated, this type of cache misses is harder to eliminate and is therefore ignored by our evaluation. Additionally, the tool returns the total number of cache misses and individual accesses. From this information, one can compute the cache miss rate, hit ratio et cetera. Moreover, the cache model shows for each access the preceding reads and writes from which data is or can be reused. If the stack distance between these accesses is too large, capacity cache misses slow down the execution. This may be avoided by rearranging the code or using loop fusion. Overall, Haystack supplies all the analysis needed for understanding the data movement.

2.3 Visual Studio Code

Visual Studio Code is a source code editor supporting a variety of features such as debugging, embedded Git control, syntax highlighting, code completion and code refactoring for most common programming languages. It has been developed by Microsoft for Windows, Linux and macOS and is based on the Electron framework that is used to create Node.js web applications. An outstanding feature is to add support for new languages by creating an extension for VS Code. These extensions can perform static code analysis and code linting using the Language Server Protocol. The result is a free and open-source developing tool that is highly customizable and therefore popular among programmers. A Developers Survey [10] in 2019 has shown that it is the top-rated development environment across the board. Thus, it is a perfect choice to integrate Haystack and its cache miss information.

2.4 Use Cases

There are mainly two use cases for making memory access patterns comprehensible: high-performance computing and education. Both try to understand how the memory hierarchy functions and how code changes affect memory access costs. However, even for advanced developers it is currently hard to get an intuition because the mechanisms are well hidden behind the compiler and computer architecture. Moreover, comparing different code versions for the cache effects is time-consuming as you either have to implement additional code that count cache misses or run a simulation with an external tool. Both approaches can have a high execution time because of their dependence on the problem size. All these reasons complicate grasping the memory access patterns.
2. Background

One of the best ways to understand a mechanism is to visualize it or its effects. It is even better if the effects can interactively be tested and compared. Therefore, real-time visual feedback is a powerful learning tool. Especially a novice programmer can benefit from it since it helps to get a feeling of the basic concepts of the memory architecture. Additionally, they can try out different code versions and can, for example, perceive the effect of the size of an array on the cache misses when looping over it.

On the other hand, high-performance code developers may know how to estimate the number of cache misses of an algorithm. But each time they use different hardware, they need to adapt their calculations to match the specifications. Building up an intuition is therefore quite challenging unlike for the computational complexity which is hardware-independent. This makes building the bridge over the gap between memory performance and processor performance, which is the main goal of a cache, an even more demanding task. Optimizations often base on reducing the cache miss rate in such systems. Visualizing the memory access pattern is helping to fill in the gap and can even give information on whether the program is compute- or memory-bound. The latter is particularly important when determining what improvement is likely to succeed.
We begin by providing an overview (§ 3.1) over the implementation with its key actors. Then, we show the implementation of the different features in the Visual Studio Code extension (§ 3.2). In the end, we explain the software architecture (§ 3.3) in more detail and what changes we have performed in the LLVM project.

Figure 3.1: An overview of the key actors.

3.1 Overview

An overview of the key actors is given in Figure 3.1. The VS Code extension is the front end, which a user interacts with, and therefore the starting point. When the user starts the analysis, the extension sends a command to the Clangd. After the command is processed, Polly is run on the specified file to find the different SCoPs in the code. The returned results are then fed into Haystack together with configurations like cache sizes. When Haystack finishes, the cache miss information is collected and transformed.
3. Implementation

into a JSON string by the clangd. In the end, the data is returned to the extension where it is distributed to the different features visualizing the cache miss information.

Figure 3.2: Screenshot of the VS Code extension with the source code editor on the left side and a webview in the right editor tab with a detailed analysis.

3.2 Visual Studio Code Extension

In this section, we present the implemented features of the Visual Studio Code extension and how they can be used to analyze code in terms of cache miss information. A screenshot of the VS Code extension is given in Figure 3.2.

On the left side, the source code editor is located. Below, we show a list of the different features integrated into it:

- Overview ruler for identifying SCoPs
- Annotations with contextual actionable information
- Hover displaying basic cache miss information for one access
3.2. Visual Studio Code Extension

- Code highlighting to mark relevant accesses and operations

In the right editor tab, a webview provides numerous visualizations of the memory access pattern. The following list presents the features in the webview from top to bottom:

- Cache miss curve for varying cache sizes or loop parameter values to tune a tile size
- Performance model evaluating the bottleneck and estimating the workload of the processor compared to the memory hierarchy levels
- User interface to add loop parameters and a slider to easily test different values
- Raw data from the cache model

Most of the features in the webview allow comparing various code variants or different parts of the code. We can use this comparison to reason about the success of an optimization or to find costly loops. In the next sections, we describe each feature in detail.

```c
void sobel() {
    for (int i = 1; i < HEIGHT - 1; i++) {
        for (int j = 1; j < WIDTH - 1; j++) {
            int g_x = -A[i - 1][j - 1] + A[i - 1][j + 1] + 2 * A[i + 1][j - 1] + 2 * A[i + 1][j + 1];
            if (g_x < 0) g_x = 0;
            if (g_y < 0) g_y = 0;
            int val = g_x * g_x + g_y * g_y;
            if (val < 255) val = 0;
            if (val > 255) val = 255;
            G[i][j] = val;
        }
    }
}
```

Figure 3.3: Editor decorations for identifying a SCoP: the green overview ruler, the Code Lenses and the code highlighting.
3. Implementation

3.2.1 Identifying SCoPs

The first features visualize which part of the code is currently processed by the cache model. For example in PolyBench [11], a lot of the code initializes, deallocates, prints debug information and measures the performance. However, our tool can only provide an analysis for SCoPs which need to be made visible to the user. We have several components contributing to the SCoP visualization. They are shown in Figure 3.3.

The extension adds a green overview ruler on the right side which marks these sections. This makes it easier to spot SCoPs in a long file. Furthermore, Code Lens is a feature of VS Code that adds actionable contextual information right in the source code. Essentially, they are links in the editor performing actions and displaying extra info. On the one hand, we use it to rapidly show an analysis of a part of the code in the webview which is explained later. On the other hand, they annotate every line with a memory access of a SCoP yielding a more fine-grained outline.

Lastly, we highlight the accesses and operations in the code that are currently analyzed in the webview. This can of course be done on any range in the code but a developer is usually interested in the cache miss information of one SCoP. Thus, these editor decorations are helping to spot them as well.

```
16    | | SCoP 1 summary | G_X[i][j]
    | G_X[i][j] = g_x > 0 ? g_x : -g_x;
17    | | SCoP 1 summary | G_Y[i][j]
    | G_Y[i][j] = g_y > 0 ? g_y : -g_y;
18
19   }
20   for (int i = 1; i < HEIGHT - 1; i++) {
21       for (int j = 1; j < WIDTH - 1; j++) {
22                   | | SCoP 1 summary | G_X[i][j] | G_Y[i][j]
23       int val = G_X[i][j] + G_Y[i][j];
24       if (val < 0)
25           val = 0;
```

Figure 3.4: Highlighting of accesses for possible reuse of data.
3.2. Visual Studio Code Extension

3.2.2 Showing Data Reuse Potential

Another feature presents all reads and writes that operate on the same data in memory as a selected access. If these memory accesses are not close enough to the selected one, others may overwrite the cache lines thereby nullifying any reuse potential. This results in capacity cache misses that could be avoided by reordering the memory accesses. This feature is an essential tool to track candidates for possible optimization. An example is shown in Figure 3.4.

3.2.3 Raw Data of Cache Model

In certain situations, we want to have a look at the raw data from Haystack. One of them is finding potential candidates to improve the reuse of data which we have explained in Section 3.2.2. For this task, we need to analyze the number of capacity cache misses and weigh them to the compulsory misses. The more capacity misses we have compared to compulsory misses, the more optimization potential is achievable. Another example is to find out the percentage of the total cache misses that happen in the analyzed code range. This helps to focus on the improvement of promising parts in the source code. It is the easiest to just show the data in these cases instead of providing a fancy visualization.

![Figure 3.5: An enhanced hover providing cache miss information for an access.](image)

We implemented two features presenting raw cache miss information. The first one is integrated into the Hover widget of Visual Studio Code. A sample of this component is given in Figure 3.5. It shows basic details such as the volume of data per memory level, the time consumption and the cache hit ratio. With this knowledge we can already filter potential data reuse candidates. However, this upgraded hover can only be displayed for a single access or operation and not for a range of code.

The second feature is placed in the webview. A table shows the following

<table>
<thead>
<tr>
<th>Index</th>
<th>Data Volume</th>
<th>Hit Ratio</th>
<th>Time Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>15.86MB</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>L2</td>
<td>15.86MB</td>
<td>99.95%</td>
<td>277.16μs</td>
</tr>
<tr>
<td>DDR</td>
<td>7.94KB</td>
<td>100%</td>
<td>903.11ns</td>
</tr>
</tbody>
</table>

We implemented two features presenting raw cache miss information. The first one is integrated into the Hover widget of Visual Studio Code. A sample of this component is given in Figure 3.5. It shows basic details such as the volume of data per memory level, the time consumption and the cache hit ratio. With this knowledge we can already filter potential data reuse candidates. However, this upgraded hover can only be displayed for a single access or operation and not for a range of code.

The second feature is placed in the webview. A table shows the following
3. Implementation

raw data from the cache model for each memory level:

- Time estimate that the memory needs to return data
- Provided data volume
- Number of cache hits
- Cache hit ratio
- Percentage of the total cache misses of the current code range
- Number of compulsory cache misses
- Number of capacity cache misses

<table>
<thead>
<tr>
<th></th>
<th>Time Estimate</th>
<th>Data Volume</th>
<th>Hits</th>
<th>Hit Ratio</th>
<th>% of Total Misses</th>
<th>Compulsory Misses</th>
<th>Capacity Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>N/A</td>
<td>127.88MB</td>
<td>0</td>
<td>0%</td>
<td>100%</td>
<td>1.05M</td>
<td>1.05M</td>
</tr>
<tr>
<td></td>
<td>(-20%)</td>
<td>(-100%)</td>
<td>(0%)</td>
<td>(0%)</td>
<td>(0%)</td>
<td>(0%)</td>
<td>(0%)</td>
</tr>
<tr>
<td>L2</td>
<td>2.23ms (0%)</td>
<td>127.88MB</td>
<td>523.52K</td>
<td>24.99%</td>
<td>100%</td>
<td>1.05M</td>
<td>523.78K</td>
</tr>
<tr>
<td></td>
<td>(+100.1%)</td>
<td>(0%)</td>
<td>(+12.5%)</td>
<td>(0%)</td>
<td>(0%)</td>
<td>(0%)</td>
<td>(+33.4%)</td>
</tr>
<tr>
<td>DDR</td>
<td>11.18ms (-14.3%)</td>
<td>95.92MB</td>
<td>1.57M</td>
<td>(-14.3%)</td>
<td>(0%)</td>
<td>(0%)</td>
<td>(0%)</td>
</tr>
<tr>
<td>Computation</td>
<td>1.96ms (0%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.6: Raw cache miss data presented in the webview table.

Furthermore, it presents the computation time required by the highlighted operations. An example is presented in Figure 3.6. All this information is already very informative but at the end of the day, we want to find out if our optimizations actually have an effect. This feature supports comparing different code versions, which is explained in more detail in the next section, to show the improvement or deterioration. To easily detect positive, negative or no change, we use different colors in the table.

3.2.4 Comparing Different Code Variants

Our features provide a lot of details about the cache hierarchy effects. Yet, the best way to assess the success of an optimization is to put the information into context. We implemented a big part of our analysis into a webview that is shown in an editor tab. The webview supports HTML and JavaScript code thereby opening the door to a whole new range of possible visualizations. Thus, almost all the features in the webview support comparing different results. The user can click on a button to set the analysis currently shown in the webview as a baseline. With the Code Lenses, mentioned in Section 3.2.1, one can open the webview for a SCoP or a single access to put the outcomes into perspective. Moreover, the user can also select just a part of the source
3.2. Visual Studio Code Extension

Figure 3.7: Graph presenting parts of the cache model information for different code variants.

code and run the cache model command to compare it with the baseline. An example graph with such a comparison is shown in Figure 3.7.

```typescript
function getHeatmapIndex(weightedMisses: number, totalWeightedMisses: number, thresholds: number[]): number {
    let ratio = weightedMisses / totalWeightedMisses;
    if (ratio === 0) {
        return 0;
    } else if (ratio < thresholds[0]) {
        return 1;
    } else if (ratio < thresholds[1]) {
        return 2;
    } else if (ratio < thresholds[2]) {
        return 3;
    } else {
        return 4;
    }
}
```

Listing 3.1: TypeScript code of heat map categorization for one access.

3.2.5 Highlighting Costly Accesses

Sometimes there are only a few reads or writes causing most of the cache misses in a program. Applying optimization on these hotspots is the best way to improve a code’s performance. With the features so far one could look at the raw data for each access but this is a time-consuming task. Thus,
3. Implementation

we implemented a heat map for the memory accesses. It has five different categories depending on the cost of the read or write. We calculate the cost as the sum of the cache access times of all memory hierarchy levels. A detailed explanation how we compute these access times is given in Section 3.2.6. With this method, we better reflect that, for example, L2-cache misses are more expensive than L1-cache misses.

Figure 3.8: Heat map that highlights costly accesses.

The lowest cost category is reserved for accesses with no cache miss at all and is not highlighted. Then as reads or writes get more expensive compared to the total costs, we color them accordingly starting at yellow and turning to red. This categorization process can be seen in Listing 3.1 where we input the weighted cost of one access and the total weighted cost of all accesses. The thresholds for the categories depend on the number of memory accesses in the source file such that we use a fine-grained classification for a high number and coarse-grained for just a few. We use the values $[5\%, 10\%, 25\%]$ for more than six accesses and $[10\%, 25\%, 50\%]$ otherwise. In Figure 3.8, we provide an example of such a heat map.

3.2.6 Performance Model

Another important aspect when tuning code is to know the potential benefit and priority of optimizations. Two main aspects that can bound performance are computation and memory time. If one of them alone is the limiting factor, it rarely makes sense to just improve the other. So far we have only implemented features analyzing the memory hierarchy effects. But as
we have said, it is important to know if optimizing memory is the way with the biggest impact.

The Roofline model [14] is an insightful visual performance model that provides information about the hardware limitations. A basic version of this model plots the floating-point performance as a function of a machine’s peak processor throughput, memory bandwidth and the arithmetic intensity. In our case, we cannot run the program and measure the floating-point performance. This is why we designed a performance model inspired by the Roofline model. We estimate the execution time assuming a measured bandwidth and plot the results split into its components. Usually, this is an undervaluation of the real computation and data-movement time since we measure the bandwidths for one component by making it the bottleneck. Thus, we assume that all components have a full workload at all times and do not have to wait on each others results. However, we do not estimate the L1-cache transfer time because we do not know how much data is transferred to and from the registers making an approximation rather difficult. The problem is that Haystack cannot properly model registers since the number of accesses is dependent on whether single values are loaded or a whole vector. Moreover, higher optimization levels reduce L1-cache hits by storing more data in the registers. This mechanism can be captured by Haystack but then we have problems in the VS Code extension to map memory accesses to the actual code. As a result, we cannot compute reliable L1-cache transfer times for our performance model.

![Figure 3.9: Performance model indicating whether the program is memory- or compute-bound.](image)

The computation time can be estimated by first finding all unary and binary operations in the source code. Then, we sum up the compute cycles it takes to perform the operations. This depends on the hardware thus our exten-
3. Implementation

Implementation supports two configurable values: the number of cycles for a simple operation such as addition or multiplication and the number of cycles for division and modulo. In the end, the number of cycles can be divided by the processor clock frequency, which can be configured as well, to get the computation time.

The data movement time can be estimated by incorporating the cache model analysis with the access time of the used hardware. This again is accomplished by configuring the hardware specifications in the extension’s settings. As we only model the throughput, the user can set the bandwidth for each level in the memory hierarchy. Then, we can simply divide the data volume going through a cache or DDR by its bandwidth. However, the results are usually an underestimation of the actual memory access time as we assume a measured maximum throughput but, for example, no latency at all. A visualization of the performance model is shown in Figure 3.9.

Like for the raw cache model data table, we can compare the results of the performance model for different code versions. An example can be found in Figure 3.7. With it, we can easily observe what impact an optimization has on the computation and memory access costs. Furthermore, it wonderfully visualizes the difference of the memory access time between the hierarchy levels. The performance model is therefore an important feature for judging the success of a certain code change.

```c
#define D 4194304
#define n 100

int A[D];
int B[D];
int C[D];

initializeArray(A, B, C);

auto start = std::chrono::steady_clock::now();
for (int k = 0; k < n; k++) {
    for (int i = 0; i < D; i++) {
        C[i] = A[i] + B[i];
    }
}

auto end = std::chrono::steady_clock::now();
double diff = std::chrono::duration<double, std::milli>(end - start).count();
printf("Runtime: %.fms\n", diff);
```

Listing 3.2: C++ implementation of the benchmark measuring the main memory bandwidth.
Configuration of Hardware Values

To get fitting approximations in the performance model, we need to configure reasonable values for the used hardware in the extension’s settings. We demonstrate how to use a set of micro-benchmarks to learn the performance of the system. The key idea is to make one component the bottleneck and measure the runtime for different workloads. An example implementation in C++ for the main memory bandwidth can be seen in Listing 3.2. All benchmarks have this basic structure changing only \(D, n\) and line 13 in the innermost loop. We choose \(n\) for each benchmark such that the runtime is in the order of several milliseconds to reduce the effects of external influences.

As a start, we want to find the number of cycles used for arithmetic operations. We set \(D\) to 512 and run it twice for \(n_1 = 100,000\) and \(n_2 = 200,000\). At each loop iteration, we use 19 times the same operation on the same data making it compute-bound. The number of cycles \(C_S\) used for one operation can be approximated by

\[
C_S \approx \frac{19 \times D \times (n_2 - n_1)}{t_2 - t_1} \cdot \frac{1}{f}
\]

where \(t_i\) stands for the measured time for \(n_i\) and \(f\) for the CPU frequency. As a result, we have 0.11 cycles per simple operation and 1.6 cycles per division and modulo.

Next, we want to find a good value for the bandwidth of the main memory. The used benchmark can be seen in Listing 3.2. The array size \(D\) is 4,194,304 to ensure that the arrays do not fit in the L3-cache which has a size of 8MB on our test system. This is memory-bound as we have three main memory accesses and only one simple arithmetic operation. We run the benchmark for \(n_1 = 100\) and \(n_2 = 200\). The bandwidth \(B_{DDR}\) can be approximated by

\[
B_{DDR} \approx \frac{4 \times A \times D \times (n_2 - n_1)}{t_2 - t_1}
\]

where \(A\) is the number of arrays and \(t_1\) and \(t_2\) are the measured execution time for \(n_1\) and \(n_2\) respectively. We multiply the data size by 4 as we have integers and they have a size of 4 bytes. With this method, we find that 10.5 GB/s is a good estimate for the maximum main memory bandwidth.

Last but not least, we measure the bandwidth of the L2-cache. At each loop iteration, we add the array element to a total sum. We run it for \(n_1 = 10,000\) and \(n_2 = 20,000\) and set \(D\) to 32,768 to provoke L2-cache hits. In the end, we can approximate the bandwidth with the formula used for the main memory which results in 60 GB/s.
3. Implementation

3.2.7 Cache Miss Curve

A cache miss curve shows the absolute number or the rate of cache misses as a function of cache size. One can read off the cache size needed to accommodate the whole data. Usually, the line has one or several steps at which the data or part of it, like one array, fits nicely into the cache. The minimum y-value of the graph is either the number of compulsory misses if the data is accessed for the first time, or zero if it is reused.

The cache miss curve reflects how big the benefit is if the data fits into the cache. It gives the knowledge whether a memory optimization like, for example, tiling is sensible. In addition, it shows which part of the memory hierarchy can profit by such a code alteration because we annotate the currently configured sizes of the caches. This allows a user to estimate the theoretical performance gain.

![Figure 3.10: Example cache miss curve for an algorithm using different tile sizes.](image)

Like other features, the cache miss curve supports comparing two different versions of a program. Thus, it helps to track the progress of an optimization process. Moreover, it indicates when we can no longer decrease the number of cache misses by some improvement techniques like reordering memory accesses. In Figure 3.10, an example of such a graph is presented.

3.2.8 Tuning the Tile Size

Let us assume that we found out that tiling might be a valid optimization technique for a specific algorithm. After implementing such a version of the code, we have to tune the tile size to fully take advantage of this im-
3.2. Visual Studio Code Extension

![Parameter slider and interface](image1)

**Figure 3.11:** Parameter slider and interface to reset existing ones or to add more.

This can be a cumbersome task especially if there are several dimensions that can be tiled. Our VS code extension provides features that partly automates this process. Firstly, the user can input loop parameters and change their value by using a slider to rapidly check a range of possible tile sizes. Furthermore, they can reset the parameter to the initial value if no performance gain is apparent. Figure 3.11 demonstrates such a slider.

![Plot of cache misses](image2)

**Figure 3.12:** Plot of the cache misses for an increasing tile size.

Moreover, the tool can run Haystack for different tile sizes at once reducing the labor of manually changing the parameter. One can provide a minimum and maximum tile size and it plots the cache misses for several values in the range. Oftentimes it makes sense to compute the misses only for a tile size that is the power of two. This is due to the cache capacities that have generally been sized in powers of two. Out of this reason, the user can choose whether a parameter has only binary values and then only such numbers are utilized when plotting. An example is shown in Figure 3.12.
3. Implementation

3.3 Software Architecture

LLVM is a project featuring a collection of modular and reusable compiler and toolchain technologies. It is designed around a language-independent intermediate representation (IR) which can be seen as a high-level assembly language. The IR is optimized over multiple passes by different transformations. This allows to develop a front end for any programming language and a back end for any instruction set architecture.

For our use case, we need a front end that supports C and C++ and is easily integrable into Visual Studio Code. Clang is one of the most popular compilers for C, C++ and Objective-C code that are built on the LLVM infrastructure. Moreover, there exists already a VS Code extension in the LLVM project repository which integrates Clangd, an implementation of the Language Server Protocol leveraging Clang. This allows us to easily integrate Haystack into an existing compiler infrastructure supported by a modern source code editor such as VS Code.

Furthermore, LLVM contains Polly, a high-level loop and data-locality optimizer. It evaluates and optimizes the memory access pattern of a program by using an abstract mathematical representation based on integer polyhedra. We use it to find Static Control Parts which are code regions that the polyhedral model can work on. Haystack needs such SCoPs to provide its analysis.

In the following sections, we outline the changes performed on a fork of the LLVM source code. Additionally, we present how we configured and integrated Haystack. Lastly, we describe a key design pattern utilized in the VS Code extension to manage the data flow.

3.3.1 Clangd

First of all, we defined three new commands that Clangd accepts. The following list shows the purpose of each command:

- Run a full analysis using the cache model for a code file
- Compute the capacity cache misses for a given array of cache sizes for a SCoP
- Compute the capacity cache misses for a given array of values of a loop parameter for a SCoP

The first command always needs to be executed before the others as it assigns an id to each SCoP. This id can later be used to run the other commands on the corresponding region of code. The reason is that Haystack performs the analysis on one SCoP instead of the whole file.
3.3. Software Architecture

When a command is received by the Clangd server, it is parsed and a hash of the file content is created. The file hash is used to check if Polly has already been executed for this particular algorithm. As Polly operates on the IR, the compilation has to be triggered beforehand which can be slow especially for long files. Thus, caching the SCoP information is one measure to speed up the response time.

If Polly does not find any SCoP, Clangd reports it back to the extension which in turn shows a warning to the user. Otherwise, we call Haystack for the given configuration. As soon as Haystack finishes its calculations, we write the results to a JSON string and send them to the VS Code extension via a notification. Unfortunately, we have to use asynchronous notification to the front end instead of directly replying to the command with the data. This is due to a timeout that is triggered when a command takes too long.

3.3.2 Polly

Polly is not enabled in LLVM per default therefore we added the appropriate flags when invoking. An especially important flag is -polly-position=early since it accomplishes that we get a non-optimized SCoP. Without it, we would provide information about the memory access pattern of an optimized version not visible to the user. This could be very unintuitive and confuse the developer because the results cannot be mapped to the actual accesses in the code. The only reliable analysis could be run and shown for a whole SCoP which may suffice for a high-performance developer. Therefore, we have a trade-off between the accuracy of the cache model results and the comprehensibility of these.

In order to run Haystack, we need to transform the output of Polly. We added an LLVM pass that takes a SCoP and returns the needed information as a JSON encoded string. We chose this representation because the integer set library (ISL) [13] used by Polly and Haystack provides functions for converting to/from a string. In addition, the infrastructure to return information from an LLVM pass is limited and strings are one of the few methods supported. JSON is used because of its simplicity and of the integrated support in LLVM. The following data is extracted from the SCoP:

- Reads and writes including the bounds of the used array indices
- Schedule tree of the SCoP
- The size of the arrays and the bounds of the used array indices
- Mapping from a statement to a line and column in the source file
- Loop parameters and their location in the source file
- Binary and unary operations such as negation, addition, multiplication and so forth including their location in the source file
3. Implementation

The line and column information about statements, loop parameters and operations in the source file are not needed for Haystack but they are crucial for the VS Code extension to map the analysis to the right code segments.

3.3.3 Haystack

In this section, we describe how we run Haystack and how we transform the results into a format that the VS Code extension accepts. We implemented a caching mechanism for the Haystack object similar to the one for the SCoPs. This guarantees to speed up future re-runs of the cache model with different cache sizes or varying parameter values. The first step is therefore to check if we have already an existing run for the current configuration and file hash. If not, we have to initialize the cache model. As we already have discussed, Haystack operates on one SCoP for which the information is passed as a string. Thus, we need to parse this string to the ISL objects which are required. Thereafter, we create the cache model object passing in the SCoP information and configurations such as the cache line size, the cache size of each level and if we want to compute the stack distance bounds. This configured object is then cached for future runs.

After either a successful creation or fetching from the cache, we verify whether there are no unknown parameters. Haystack provides a function to get the parameters defined in the SCoP and should not return any in this case. We do not allow variable loop parameters because Polly generalizes the name of such parameters which makes the mapping to the actual field or local variable very hard and is therefore omitted in this tool. Moreover, we would have to parse the abstract syntax tree of the compiler to find such a mapping. We accept therefore only loop parameters defined by a C macro which are just a constant number. Once the parameters are validated, we run the actual cache miss analysis.

In the end, we transform the results to a JSON object which is mostly just filling in the information to the right place. However, Haystack returns the cache miss info for each read and write statement of the IR. Thus, we have to add the line and column of the statement obtained in the Polly pass to the results. Without this information, the extension could not show the analysis at the right location in the code.

Haystack provides us with the following details for each statement:

- Number of compulsory cache misses
- Number of capacity cache misses
- Total number of accesses
- List of statements from which we might reuse data in the cache
Unfortunately, Haystack does not model registers which results in a high number of L1-cache hits. We try to circumvent this problem by simulating the registers using an additional level in the memory hierarchy. In our case, we found that 12 registers are sufficient to have a more reasonable number of L1-cache hits. Since we do not know the actual data transfer between the registers and the L1-cache, we do not provide any analysis on this part of the memory hierarchy. Therefore, we add this register cache level hidden from the user in our VS Code extension and remove it before showing the results.

3.3.4 Repository Pattern

To simplify the data flow in our VS Code extension, we use the repository pattern which adds an abstraction layer to the data access. Consumer of such a repository do not know anything about how the data is saved or fetched from, which is in our case the Clangd. This allows us to easily cache the results from the Clangd server using an LRU strategy. A hash of the file is computed and compared to verify if the results are already in the cache. We employ this mechanism to reduce the load on the Clangd server and to increase the response time of the extension. An overview of the repository pattern is given in Figure 3.13.

There are three different repositories, one for each data object. They all have various functions to retrieve the required results from the cache model such as per file or for a range in the code. Moreover, one can add listeners to

Figure 3.13: Repository pattern and its components.
3. Implementation

every repository in order to get updates for a specific file opened in VS Code. This makes it simple to refresh the visible visualizations for a newer code version as the repositories can simply re-run the cache model and notify their listeners.
Chapter 4

Evaluation

As a first step, we show an example application (§ 4.1) of our Visual Studio Code extension to optimize a Sobel operator implementation. Secondly, we evaluate the responsiveness and usability (§ 4.2) of our plugin as a day-to-day tool for a high-performance code engineer.

4.1 Optimization Story

In this section, we show how one can use our Visual Studio Code extension to optimize their code. For this purpose, we implement a naive version of the Sobel operator which is primarily used in image processing for detecting edges. Then, we do several iterations where we first analyze the code with our extension and then depending on the results, apply different optimization strategies. In the end, we should have an efficient solution to our problem for a specific set of hardware.

4.1.1 Sobel Operator

The goal of the Sobel operator is to approximate the gradient of the image intensity function by using convolution with a filter in horizontal and vertical directions. This filter is small, separable and integer-valued and therefore rather cheap to apply on an image. More formally, if $A$ is the source image, we compute two images $G_x$ and $G_y$ where each point corresponds to the horizontal or vertical derivative approximation. We apply two 5x5 kernels on $A$ by using 2-D convolution. The calculations are

$$G_x = \begin{bmatrix} 1 & 2 & 0 & -2 & -1 \\ 4 & 8 & 0 & -8 & -4 \\ 6 & 12 & 0 & -12 & -6 \\ 4 & 8 & 0 & -8 & -4 \\ 1 & 2 & 0 & -2 & -1 \end{bmatrix} * A$$
4. Evaluation

and

\[
G_y = \begin{bmatrix}
1 & 4 & 6 & 4 & 1 \\
2 & 8 & 12 & 8 & 2 \\
0 & 0 & 0 & 0 & 0 \\
-2 & -8 & -12 & -8 & -2 \\
-1 & -4 & 6 & -4 & -1 \\
\end{bmatrix} \ast A
\]

where \( \ast \) stands for the 2-D signal processing convolution operation.

As we already stated, the Sobel operator is separable. Therefore, we can rewrite \( G_x \) as

\[
G_x = \begin{bmatrix}
1 \\
4 \\
6 \\
4 \\
1 \\
\end{bmatrix} \ast ([1 \ 2 \ 0 \ -2 \ -1] \ast A)
\]

and \( G_y \) as

\[
G_y = \begin{bmatrix}
1 \\
2 \\
0 \\
-2 \\
-1 \\
\end{bmatrix} \ast ([1 \ 4 \ 6 \ 4 \ 1] \ast A)
\]

Lastly, we combine \( G_x \) and \( G_y \) to represent the resulting gradient approximations by calculating

\[
G = \frac{1}{8}(\text{ABS}(G_x) + \text{ABS}(G_y))
\]

where \( \text{ABS} \) is taking the absolute value of each element of the matrix. We also need to apply a scale factor to ensure that the Sobel kernels are proper estimators.

4.1.2 Experimental Setup

In the following implementations, we assume that we receive a source image of size 2044x2044. The image is then padded with two layers of zeroes as the Sobel operator utilizes the neighbors of each pixel. Padding ensures that the calculations can be applied to the border pixel as well. This leaves us with a resulting matrix \( A \) with \( \text{WIDTH} = 2048 \) and \( \text{HEIGHT} = 2048 \).
Moreover, all code variants are compiled with the -O3 flag which reduces the accuracy of the cache model as it evaluates the non-optimized code. Still, the analysis is accurate enough to provide valid cache miss information that can be used to improve an algorithm. Lastly, our test system consists of an Intel Core i7-6700K CPU with four cores and 16 GB DDR4-2133 RAM.

4.1.3 Naive Implementation

A very naive implementation of the Sobel operator consists of:

- Loop through the image and compute each entry of $G_x$ by applying the kernel
- Loop through the image and compute each entry of $G_y$ by applying the kernel
- Loop through $G_x$ and $G_y$ simultaneously and combine them to the result

```c
for (int i = 2; i < HEIGHT - 2; i++) {
    for (int j = 2; j < WIDTH - 2; j++) {
        G_X[i][j] = abs(g);
    }
}
for (int i = 2; i < HEIGHT - 2; i++) {
    for (int j = 2; j < WIDTH - 2; j++) {
        G_Y[i][j] = abs(g);
    }
}
```
4. Evaluation

\begin{verbatim}
28     G_Y[i][j] = abs(g);
29 }
30 }
31 for (int i = 2; i < HEIGHT - 2; i++) {
32     for (int j = 2; j < WIDTH - 2; j++) {
33         int g = (G_X[i][j] + G_Y[i][j]) / 8;
34         g = max(0, g);
35         g = min(g, 255);
36         G[i][j] = g;
37     }
38 }
\end{verbatim}

Listing 4.1: Naive implementation of the Sobel operator.

Obviously, this is very inefficient since we iterate twice over the image and once over \( G_x \) and \( G_y \) respectively, which have the same size as \( A \). An implementation in C++ of this algorithm is given in Listing 4.1.

<table>
<thead>
<tr>
<th>Data Volume</th>
<th>Hit Ratio</th>
<th>Compulsory Misses</th>
<th>Capacity Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>223.6 MB</td>
<td>0%</td>
<td>28.6%</td>
</tr>
<tr>
<td>L2</td>
<td>223.6 MB</td>
<td>50.0%</td>
<td>57.2%</td>
</tr>
<tr>
<td>DDR</td>
<td>111.9 MB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Cache model data for the naive Sobel operator implementation.

Figure 4.1: Cache miss curve for varying cache sizes using the naive Sobel operator implementation.

Parts of the analysis of this code with our tool are presented in Table 4.1. The results show that we do not have any L1-cache hit and the L2-cache has a
cache miss ratio of 50%. Furthermore, we observe that about three-quarters of the L1-cache misses and 42.8% of L2-cache misses are due to the cache capacity. The cache miss curve for increasing cache sizes (Figure 4.1) shows that our three arrays do not even fit into the L2-cache by a long shot. Lastly, the performance model displays that the time for DDR accesses is the limiting factor of the algorithm. It takes about 25% longer than the arithmetic computations and 2.9 times longer than L2-cache accesses. When we compare the predicted bottleneck performance, which is 11.2 ms, with the actual runtime of 16.8 ms, we see that the model is too optimistic. This makes sense as the model assumes a high throughput and no latency. Overall, we can safely say that, as expected, this naive variant of the code performs very inefficient in terms of memory costs.

Figure 4.2: Memory access heatmap of the naive Sobel operator implementation.
4. Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Time Estimate</th>
<th>Data Volume</th>
<th>Hits</th>
<th>Hit Ratio</th>
<th>% of Total Misses</th>
<th>Compulsory Misses</th>
<th>Capacity Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>N/A</td>
<td>15.84MB</td>
<td>0</td>
<td>0%</td>
<td>7.09%</td>
<td>0</td>
<td>259.59K</td>
</tr>
<tr>
<td>L2</td>
<td>276.8μs</td>
<td>15.84MB</td>
<td>0</td>
<td>0%</td>
<td>14.17%</td>
<td>0</td>
<td>259.59K</td>
</tr>
<tr>
<td>DDR</td>
<td>1.58ms</td>
<td>15.84MB</td>
<td>259.59K</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.3: Analysis of a single access that has a high number of capacity misses.

If we find the source of these capacity misses, we may adapt our code to better utilize the cache. By having a closer look on the individual accesses with the heatmap feature shown in Figure 4.2, we notice that some accesses are especially expensive. Moreover, some of these costly reads and writes only cause capacity misses: the second access to \(A\) on line 22 and the accesses to \(G_x\) and \(G_y\) on line 33 in Listing 4.1. The analysis of the former memory access is given in Figure 4.3. Thus, these two code pieces are perfect candidates for our first optimization technique.

Figure 4.4: Highlighting of memory accesses that make reuse possible for the naive Sobel operator.

4.1.4 Loop Fusion

One optimization method is loop fusion which moves statements closer together that use the same memory region. This reduces capacity cache misses.
4.1. Optimization Story

if there are other memory accesses between the statements that interact with the same part of the cache. In order to understand the results of loop fusion, we only apply this technique on the accesses to \( A \) on lines 18-27 of Listing 4.1 in this section. We discuss the optimization of the other candidates in Section 4.1.5.

```c
1 for (int i = 2; i < HEIGHT - 2; i++) {
2   for (int j = 2; j < WIDTH - 2; j++) {
3       int g_x = A[i - 2][j - 2] + 2 * A[i - 2][j - 1]
4           - 2 * A[i - 2][j + 1] - A[i - 2][j + 2]
5           + 4 * A[i - 1][j - 2] + 8 * A[i - 1][j - 1]
6           - 8 * A[i - 1][j + 1] - 4 * A[i - 1][j + 2]
8           - 12 * A[i][j + 1] - 6 * A[i][j + 2]
9           + 4 * A[i + 1][j - 2] + 8 * A[i + 1][j - 1]
10          - 8 * A[i + 1][j + 1] - 4 * A[i + 1][j + 2]
12          - 2 * A[i + 2][j + 1] - A[i + 2][j + 2];
13       int g_y = A[i - 2][j - 2] + 4 * A[i - 2][j - 1]
14           + 6 * A[i - 2][j] + 4 * A[i - 2][j + 1]
15           + A[i - 2][j + 2] + 2 * A[i - 1][j - 2]
16           + 8 * A[i - 1][j - 1] + 12 * A[i - 1][j]
17           + 8 * A[i - 1][j + 1] + 2 * A[i - 1][j + 2]
18           - 2 * A[i + 1][j - 2] - 8 * A[i + 1][j - 1]
19           - 12 * A[i + 1][j] - 8 * A[i + 1][j + 1]
21           - 4 * A[i + 2][j - 1] - 6 * A[i + 2][j]
22           - 4 * A[i + 2][j + 1] - A[i + 2][j + 2];
23       G_X[i][j] = abs(g_x);
24       G_Y[i][j] = abs(g_y);
25   }
26 }
27 for (int i = 2; i < HEIGHT - 2; i++) {
28   for (int j = 2; j < WIDTH - 2; j++) {
29       int g = (G_X[i][j] + G_Y[i][j]) / 8;
30       g = max(0, g);
31       g = min(g, 255);
32       G[i][j] = g;
33   }
34 }
```

Listing 4.2: Fused loop implementation of the Sobel operator where \( G_x \) and \( G_y \) are computed in the same iteration over the image.

Before applying this optimization step, one has to find statements that make sense to bring closer together. The tool provides this information when
4. Evaluation

moving the cursor to any memory access. It then highlights all reads and writes operating on the same data in memory. When analyzing the access on line 22 of Listing 4.1 with this feature (Figure 4.4), we immediately observe that a lot of accesses in the loop where we compute $G_X$ are highlighted.

This makes sense as we iterate over the same image twice and could reuse data from the cache. But the input image is rather large and does not fit into the cache which prevents reuse. Besides, we save the entry of $G_X$ in every loop iteration taking up part of the available cache size. However, if we fuse the loops over the images to one, the data for calculating $g_y$ can be read from the cache as we have already fetched it for computing $g_x$. The code that does exactly this can be seen in Listing 4.2.

![Figure 4.5: Comparison of the raw cache miss information between the naive Sobel operator and the fused loop version.](image)

![Figure 4.6: Comparison of the naive implementation to the fused loop version.](image)

When we compare the results to the naive implementation, we observe
about 40% less cache capacity misses which is presented in Figure 4.5. This matches our expectations of applying loop fusion. However, the cache miss rate is higher for larger cache sizes as one can see in Figure 4.6. The culprit for this inconsistency is the compiler who stores the accesses to $A$ in lines 3-12 of Listing 4.2 locally and reuses it when calculating $g_y$. This results in about 28% less total data transfer which in turn increases the cache miss rate. The outcome is therefore even better than we have anticipated. Without our tool, this improvement by the compiler would have gone unnoticed. Hence, not only can one optimize their code with the extension but also better understand the work of the compiler. The latter is a very important aspect when creating high-performance programs because the compiler greatly helps in writing simple code that is under the hood optimized.

![Figure 4.7: Comparison of the performance model between the naive Sobel operator and the fused loop version.](image)

Now we want to have a look at the performance model and compare it to the actual runtime. We measure 16.4 ms when we execute the algorithm and in Figure 4.7, we can observe that the model predicts 9.6 ms. The approximation is getting worse in contrast to the naive Sobel operator. This shows that there are still some effects affecting the execution time which our estimation does not capture and thus influences the accuracy. As we have already stated, the performance model has a trade-off between abstraction and precision but it is still a valuable information source. Nevertheless, it is evident that we need to further optimize the code.

For the next improvement, we already found a candidate in the previous section. On line 29 of Listing 4.2, $G_x$ and $G_y$ are accessed and both reads cause only capacity cache misses. Therefore, the next optimization step is again loop fusion.
4. Evaluation

```c
for (int i = 2; i < HEIGHT - 2; i++) {
    for (int j = 2; j < WIDTH - 2; j++) {
            - 12 * A[i][j + 1] - 6 * A[i][j + 2]
            - 2 * A[i + 2][j + 1] - A[i + 2][j + 2];
        int g_y = A[i - 2][j - 2] + 4 * A[i - 2][j - 1]
            + 6 * A[i - 2][j] + 4 * A[i - 2][j + 1]
            + A[i - 2][j + 2] + 2 * A[i - 1][j - 2]
            + 8 * A[i - 1][j - 1] + 12 * A[i - 1][j]
            + 8 * A[i - 1][j + 1] + 2 * A[i - 1][j + 2]
            - 2 * A[i + 1][j - 2] - 8 * A[i + 1][j - 1]
            - 12 * A[i + 1][j] - 8 * A[i + 1][j + 1]
            - 4 * A[i + 2][j - 1] - 6 * A[i + 2][j]
            - 4 * A[i + 2][j + 1] - A[i + 2][j + 2];
        int g = (abs(g_x) + abs(g_y)) / 8;
        g = max(0, g);
        g = min(g, 255);
        G[i][j] = g;
    }
}
```

Listing 4.3: Default implementation of the Sobel operator.

4.1.5 Loop Fusion II

Sometimes, we can apply an optimization technique several times on an algorithm. We have already fused loops in the last section but we have found another part of the code where loop fusion makes sense. For our Sobel operator, we can actually compute the final image in the same loop instead of storing the intermediate results and reusing them later on. Not only does this prevent cache misses but it also saves memory space as we can drop $G_x$ and $G_y$ altogether. This is the implementation that is usually provided as an example for the Sobel operator. An example in C++ is provided in Listing 4.3.
4.1. Optimization Story

Figure 4.8: Comparison of the raw cache miss information between the two loop fusion versions.

Now this optimization results in quite a memory-efficient implementation which is presented in Figure 4.8. As there are no capacity misses in the L2-cache, we completely utilize its potential. The L1-cache has still about twice as many capacity misses than compulsory misses and no hits at all. We conclude from this that five lines of our image do not fit into the 32KiB L1-cache. This finding can be verified by multiplying the width of the image with the size of an integer which is 4B resulting in an image row size of 2048 * 4B = 8KiB. We now know that four rows of our input image fit in the L1-cache. Since we use a 5x5 kernel, it is very hard to better utilize the L1-cache. Nonetheless, we have reduced the total accessed data volume by 40% and the estimated time the main memory needs to provide the data by two thirds. All the results show that we currently have an efficient Sobel operator in terms of memory.

![Image](image.png)

Figure 4.9: Comparison of the performance model between the two loop fusion versions.
So far our runtime has been around 16 ms for the last two code versions. In the newly optimized variant the code takes 11.8 ms to finish. We have therefore a speedup of 1.4 compared to our initial algorithm which is already a great improvement. The performance model for our current implementation of the Sobel operator is presented in Figure 4.9. Comparing the change of the runtime to the difference predicted by the performance model, which is about 2.5 ms, we see that the model is better this time.

One interesting fact can be seen when we look at the computation time. It is reduced by about 18% even when there are exactly the same operations in the code. The reason is again the compiler as it uses a right shift when applying the scaling factor in the current implementation. A right shift is a valid choice in C++ when operating on positive numbers whereas it is implementation-dependent for negative numbers. In the current code version, the compiler knows that it is a positive value because of the absolute function which is in the same for-loop. In contrast, the compiler cannot assume the same in the last variant where the absolute function is in a different loop and therefore uses the more expensive division.

However, there are two main contributors to the runtime in the execution of a program: the computation time and the memory access time. Thus, improving the memory hierarchy efficiency can speed up the execution but if the CPU is the limiting factor, we should invest our effort in this aspect as well. In the performance model, we can easily see that we significantly reduced the data-movement time compared to the previous implementation. Yet, we still have an imbalance between computation and memory time. Usually, the goal is to get these two as close together as possible. Therefore, our next step in the optimization process is to reduce the computation time.

```cpp
for (int i = 2; i < HEIGHT - 2; i++) {
    for (int j = 2; j < WIDTH - 2; j++) {
    }
}
for (int i = 2; i < HEIGHT - 2; i++) {
    for (int j = 2; j < WIDTH - 2; j++) {
        int g_x = K_X[i - 2][j] + 2 * K_X[i - 1][j] - 2 * K_X[i + 1][j] - K_X[i + 2][j];
        int g_y = K_Y[i - 2][j] + 4 * K_Y[i - 1][j] + 6 * K_Y[i][j] + 4 * K_Y[i + 1][j] + K_Y[i + 2][j];
    }
}
```
4.1. Optimization Story

```c
int g = (abs(g_x) + abs(g_y)) / 8;
g = max(0, g);
g = min(g, 255);
G[i][j] = g;
```

Listing 4.4: Implementation of the separated Sobel operator.

4.1.6 Filter Separation

One way to reduce computation time for our algorithm is to separate the Sobel operator into a vertical and a horizontal 1-dimensional filter which we have already discussed in Section 4.1.1. In this case, we need to save the results of applying the first 1-dimensional filter for when we use the second one. This trade-off between memory and performance is necessary for us as it currently makes more sense to improve the computation time.

In a first attempt, we use two 2-dimensional arrays $K_x$ and $K_y$ of the size of the image to store the intermediate results. The algorithm consists of:

- Loop through the image and compute each entry of $K_x$ and $K_y$ by applying the corresponding 1-dimension kernel
- Loop through $K_x$ and $K_y$ simultaneously, apply the second 1-dimension kernel and compute entry of $G$

The C++ implementation is given in Listing 4.4.

<table>
<thead>
<tr>
<th></th>
<th>Time Estimate</th>
<th>Data Volume</th>
<th>Hits</th>
<th>Hit Ratio</th>
<th>% of Total Misses</th>
<th>Compulsory Misses</th>
<th>Capacity Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>N/A</td>
<td>207.59MB</td>
<td>0</td>
<td>0%</td>
<td>100%</td>
<td>1.05M</td>
<td>2.35M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(+116.7%)</td>
<td>(0%)</td>
<td>(0%)</td>
<td>(0%)</td>
<td>(+100%)</td>
<td>(+125%)</td>
</tr>
<tr>
<td>L2</td>
<td>3.63ms</td>
<td>207.59MB</td>
<td>1.83M</td>
<td>53.82%</td>
<td>100%</td>
<td>1.05M</td>
<td>523.26K</td>
</tr>
<tr>
<td></td>
<td>(+116.7%)</td>
<td>(+116.7%)</td>
<td>(+75%)</td>
<td>(+12.8%)</td>
<td>(0%)</td>
<td>(+100%)</td>
<td>(--            )</td>
</tr>
<tr>
<td>DDR</td>
<td>9.57ms</td>
<td>95.88MB</td>
<td>1.57M</td>
<td>199.9%</td>
<td>199.9%</td>
<td>199.9%</td>
<td>199.9%</td>
</tr>
<tr>
<td>Computation</td>
<td>2.99ms</td>
<td>(58.1%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.10: Comparison of the raw cache miss information between the second loop fusion and filter separation code variant.

As expected, the data volume coming from the DDR is tripled which can be seen in Figure 4.10. This reflects the two additional matrices $K_X$ and $K_Y$. Moreover, we have more L2-cache hits because in the second loop nest $K_X$, $K_Y$ and $G$ compete for the lines in the L1-cache. This is due to the fact that only four rows of the image fit in the L1-cache. But we utilize three
4. Evaluation

different data structure operating on the same size of rows which overwrite each other’s results.

Figure 4.11: Comparison of the performance model of the memory optimized and separated Sobel operator.

On the other hand, the computation time decreased by 58.1% compared to the implementation in Section 4.1.5. That is what we wanted to achieve. However, if we look at the runtime, we see a decrease of about 0.6 ms whereas our extension predicted an increase of 2.5 ms which is presented in Figure 4.11. It seems that it does not fully capture all factors contributing to the runtime. One possible explanation is the omitted integration of array index operations in the computation time calculations. In the loop fusion variant, we have 41 array accesses where almost each one of them uses an arithmetic operation for at least one of the indices. In contrast, our newest implementation only features 21 accesses with 16 arithmetic operations for the indices. The index calculations are computed at each iteration thereby influencing the runtime.

A notable change is that the number of capacity misses has increased. We can, therefore, try to reduce them again. When differentiating between the memory accesses, we indeed find that on lines 12 to 16 in Listing 4.4 the reads have very high capacity miss rates. So the next step in improving memory efficiency is minimizing these cache misses.

```c
for (int i = 0; i < 4; i++) {
    for (int j = 2; j < WIDTH - 2; j++) {
    }
}
```
4.1. Optimization Story

    - 2 * A[i][j + 1] - A[i][j + 2];

    }
  }
for (int i = 2; i < HEIGHT - 2; i++) {
  int L_i1 = (i - 2) % 4;
  int L_i2 = (i - 1) % 4;
  int L_i3 = i % 4;
  int L_i4 = (i + 1) % 4;
  for (int j = 2; j < WIDTH - 2; j++) {
    int L_x = A[i + 2][j - 2] + 4 * A[i + 2][j - 1]
               + 6 * A[i + 2][j] + 4 * A[i + 2][j + 1]
               + A[i + 2][j + 2];
    int L_y = A[i + 2][j - 2] + 2 * A[i + 2][j - 1]
              - 2 * A[i + 2][j + 1] - A[i + 2][j + 2];
    int g_x = L_X[L_i1][j] + 2 * L_X[L_i2][j]
              - 2 * L_X[L_i4][j] - L_x;
    int g_y = L_Y[L_i1][j] + 4 * L_Y[L_i2][j]
              + 6 * L_Y[L_i3][j] + 4 * L_Y[L_i4][j]
              + L_y;
    L_X[L_i1][j] = l_x;
    L_Y[L_i1][j] = l_y;
    int g = (abs(g_x) + abs(g_y)) / 8;
    g = max(0, g);
    g = min(g, 255);
    G[i][j] = g;
  }
}

Listing 4.5: Implementation of the vertical tiled Sobel operator.

### 4.1.7 Vertical Tiling

In our previous implementation, we have saved all intermediate results in $K_x$ and $K_y$ respectively. This is not needed because we only utilize five rows when applying the second vertical filter. Therefore, $L_x$ and $L_y$ are 4x2048 matrices in this section. We call this vertical tiling as we go tile by tile through the image and compute the missing row on-the-fly. The flow of an optimized version is:

- Pre-compute the entries of $L_x$ and $L_y$ for the first four rows of the image by applying the corresponding 1-dimension kernel
4. Evaluation

- Loop through the image row-by-row and for every entry:
  - Calculate entry of $L_X$ and $L_Y$ on the missing fifth row for the second filter
  - Apply the second kernel and compute $G$
  - Overwrite the obsolete result with the newly calculated entry of the fifth row

An implementation in C++ is given in Listing 4.5.

Figure 4.12: Comparison of the performance model of the separated and vertically tiled Sobel operator.

When we look at the analysis of our tool, we observe a big improvement in terms of data-movement time as one can see in Figure 4.12. We reduce the time in the DDR about two thirds plus we observe a small decrease in the L2-cache. This is already a sign that $L_X$ and $L_Y$ fit in the L2-cache whereas $K_X$ and $K_Y$ did not.

![Comparison of the raw cache miss information between the separated Sobel filter and vertically tiled algorithm.](image)

Figure 4.13: Comparison of the raw cache miss information between the separated Sobel filter and vertically tiled algorithm.

<table>
<thead>
<tr>
<th></th>
<th>Time Estimate</th>
<th>Data Volume</th>
<th>Hits</th>
<th>Hit Ratio</th>
<th>% of Total Misses</th>
<th>Compulsory Misses</th>
<th>Capacity Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>N/A</td>
<td>143.81MB</td>
<td>42</td>
<td>0%</td>
<td>100%</td>
<td>524.8K</td>
<td>1.83M</td>
</tr>
<tr>
<td></td>
<td>(-30.7%)</td>
<td>(-)</td>
<td>(+0%)</td>
<td>(0%)</td>
<td>(-49.9%)</td>
<td>(-22.2%)</td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>2.51ms</td>
<td>143.81MB</td>
<td>1.83M</td>
<td>77.73%</td>
<td>100%</td>
<td>524.8K</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(-30.7%)</td>
<td>(-)</td>
<td>(+23.9%)</td>
<td>(0%)</td>
<td>(-49.9%)</td>
<td>(-100%)</td>
<td></td>
</tr>
<tr>
<td>DDR</td>
<td>3.2ms</td>
<td>32.03MB</td>
<td>524.8K</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(-66.6%)</td>
<td>(-66.6%)</td>
<td>(-66.6%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computation</td>
<td>2.99ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(+0.1%)</td>
</tr>
</tbody>
</table>
4.1. Optimization Story

We can corroborate our statements by looking at the data in Figure 4.13. The data volume provided by the DDR has decreased by about 64MB compared to the implementation in the previous section. Moreover, we have now a hit ratio of 77.7% for the L2-cache and no capacity misses. All these facts support that $L_X$ and $L_Y$ fit in the L2-cache thereby reducing the data traffic from the main memory.

An important point is that we eliminated capacity misses in the L2-cache which means that the DDR provides each data block once. This is an important goal of memory optimization as the main memory has a low bandwidth compared to the caches. Reducing data-movement from the DDR is thereby a valuable objective. As a result, the runtime is reduced by 4.8 ms compared to the filter separation version. The performance model predicts a difference of 6.4 ms which is already quite close. A probable explanation for the more accurate estimation is that we have a better balance between the computation and data movement costs. Both the processor and the memory hierarchy are more working to their capacity. These workloads are better reflected by the measured bottleneck bandwidths used in the performance model. This argument can also be used on the different levels in the memory hierarchy. If the L2-cache access costs are similar to the DDR costs, they share the workload which makes our estimations more accurate.

Figure 4.14: Varying width for the vertically tiled Sobel operator.
4. Evaluation

4.1.8 Optimal Solution

Have we reached an optimal solution for our Sobel operator? This is a non-trivial question and cannot be answered by our tool. Nevertheless, it provides valuable evidence whether we have accomplished a memory efficient implementation. If, for example, we have reached a capacity miss rate of 0% on every level of the memory hierarchy, we have a good solution in terms of memory for the used data structures. In our case, we still have L1-cache misses in the vertically tiled Sobel operator. Thus, the next optimization technique could be horizontal tiling. Varying the width of the image, which can be observed in Figure 4.14, provides us with the optimal tile size for such an algorithm. This demonstrates that we still have the potential for optimizations.

However, the goal of this optimization story is not to find the optimal solution. It shows how a developer can use our extension throughout the development process to compensate for the lack of intuition for memory access patterns. In the previous sections, we have demonstrated that our tool can provide the necessary cache miss information required for this compensation. Moreover, we have reduced the runtime from an initial 16.8 ms to 6.4 ms which is a speedup of 2.6. So not only have we improved the understanding of caching effects but we also found a better solution to our Sobel operator problem.

4.2 Responsiveness and Usability

Another important aspect of our tool is the responsiveness. If we have to wait several seconds or even minutes on a result, it may still have its usage for a thorough analysis. However, our goal is to fill the gap between caching effects and the intuition of a programmer. It is thus essential that a user can witness the consequence of a change in the code on the memory access pattern in real-time.

The response time of the Visual Studio Code extension is in fact less than five seconds for most algorithms. This is sufficiently fast for our target to be reached. One weak point is the graph where we vary a parameter. The issue is that we need to change the parameter in the code which causes full runs of Polly and Haystack for each value. So even if the response time is two seconds, the calculation of eight different data points for the graph takes 16 seconds. Although this is rather slow, this feature is more thought of as an automated search for an optimal tile size. For this purpose it is sufficient considering that this is usually done once per algorithm.

We focused to implement features that are more of a graphical nature than just raw numbers. This enables faster perception of memory effects in general and helps especially inexperienced programmers to better understand
4.2. Responsiveness and Usability

them. Moreover, one cannot only analyze one piece of code but also compare it to different versions. Such a visual connection between changes in the algorithm and their effects on the number of cache misses is essential to improve performance. Nonetheless, we provide raw numbers as well to empower senior developers to analyze their code in a much more thorough manner.

Furthermore, we integrated some of our analysis results into existing features of Visual Studio Code. We wanted to use these well-designed features in order to be as user-friendly as possible. However, the bigger part is in a webview where we use HTML and JavaScript to present the data. These technologies are often used in modern web applications that are dynamic and responsive. For our graphs, we utilize ChartJS [1], a JavaScript library that provides nicely designed charts. Overall, we can confidently say that we achieved our goal to keep the results simple, easily perceptible and informative for developers of all skill levels.
Chapter 5

Conclusion

The scale of today’s data processing is growing steadily. Efficient algorithms to analyze this ever-increasing stream become more important. Memory performance has not increased like the processor performance resulting in deep memory hierarchies. Understanding the memory access pattern is therefore an essential ability today’s high-performance engineers need to possess. Yet, the time complexity of an algorithm is for most developers more obvious than the cost of memory accesses which is dependent on the cache state. The ever-growing and deepening cache hierarchy of modern processors and the increasingly more complex compilers even worsen this gap.

With our Visual Studio Code extension that provides programmers with accurate information about the memory behavior, we reduce the discrepancy of the intuition between the two main metrics of a program’s execution time. We evaluated various visualization techniques about their importance for memory optimizations. They enable developers to get a feeling of the memory access costs and tune their code to efficiently use the memory hierarchy. The results of our tool are reflected in the runtimes of the different Sobel operator implementations that we used as an example application. Moreover, they can be used to evaluate the benefit from future optimizations for appropriate prioritization. As a result, memory-aware programming is easier to grasp for a broad spectrum of developers.

To be adopted in the day-to-day workflow of a programmer striving for utmost performance, the key of a cache information providing tool is responsiveness. The analysis needs to be available when the development process requires it. Otherwise, it is pushed to a later stage when fundamental design flaws are hard to straighten out. The response time of the used cache model is often problem size-independent enabling our extension to be an interactive learning and analysis tool.
A.1 Naive Implementation

In Section 4.1.3, we showed the analysis for the naive Sobel operator. Figures A.1 and A.2 emphasize our findings.

Figure A.1: Performance model of the naive Sobel operator implementation.

<table>
<thead>
<tr>
<th></th>
<th>Time Estimate</th>
<th>Data Volume</th>
<th>Hits</th>
<th>Hit Ratio</th>
<th>% of Total Misses</th>
<th>Compulsory Misses</th>
<th>Capacity Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>N/A</td>
<td>223.56MB</td>
<td>0</td>
<td>0%</td>
<td>100%</td>
<td>1.05M</td>
<td>2.62M</td>
</tr>
<tr>
<td>L2</td>
<td>3.91ms</td>
<td>223.56MB</td>
<td>1.83M</td>
<td>49.97%</td>
<td>100%</td>
<td>1.05M</td>
<td>785.41K</td>
</tr>
<tr>
<td>DDR</td>
<td>11.17ms</td>
<td>111.84MB</td>
<td>1.83M</td>
<td></td>
<td>100%</td>
<td>1.05M</td>
<td></td>
</tr>
<tr>
<td>Computation</td>
<td>8.91ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure A.2: Full analysis of the naive Sobel operator implementation.
A.2 Loop Fusion

The following Figure (A.3) shows additional results for the first loop fusion optimized implementation of the Sobel filter. It enforces our findings drawn from the figures in Section 4.1.4.

Figure A.3: Comparison of the number of cache misses between the naive Sobel operator and the fused loop version.
A.3 Loop Fusion II

The Figure below (A.4) supports our findings in Section 4.1.5.

Figure A.4: Comparison of the number of cache misses between the two loop fusion versions.
A.4 Filter Separation

The next Figure (A.5) provides further analysis of the separated Sobel filter implementation. It is congruent with our findings in Section 4.1.6.

Figure A.5: Comparison of the number of cache misses between the second loop fusion and filter separation code variant.
A.5 Vertical Tiling

The Figure (A.6) suggests similar observations as in Section 4.1.7.

Figure A.6: Comparison of the number of cache misses between the separated Sobel filter and vertically tiled algorithm.
A.6 Optimal Solution

Last but not least, this Table (A.1) provides all runtimes of the different algorithms implemented in Sections 4.1.3-4.1.7.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>16.8 ms</td>
</tr>
<tr>
<td>Loop Fusion</td>
<td>16.4 ms</td>
</tr>
<tr>
<td>Loop Fusion II</td>
<td>11.8 ms</td>
</tr>
<tr>
<td>Filter Separation</td>
<td>11.2 ms</td>
</tr>
<tr>
<td>Vertical Tiling</td>
<td>6.4 ms</td>
</tr>
</tbody>
</table>

Table A.1: Runtime of the different Sobel operator implementations.


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Cache Model Plugin for Memory Hierarchy Aware Programming

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):
Fompeyrine

First name(s):
Patrick

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Wädenswil, 22.04.2020

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.