FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

An Analysis of Performance Recipes with C Applications

Nuno Miguel Rodrigues Gomes

WORKING VERSION

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João Manuel Paiva Cardoso

Co-supervisor: João Carlos Viegas Martins Bispo

April 8, 2021

Abstract

Performance aware programming is an essential approach in many software engineering fields, either with the goal of improving application execution times, reducing the amount of energy consumed by a device while executing an application or any other motivation. In this dissertation we conduct a study of code transformations oriented towards improving software performance.

Firstly we propose a portfolio of appropriate recipes, estimate their impact and describe their implementation and evaluate the possibility of automating each recipe using a source-to-source compiler. Developing such portfolio may provide interested parties with a useful platform that can be used as a starting point to the field of code refactoring for performance improvement. The portfolio includes recipes to apply loop transformations, e.g., Loop Unrolling and Loop Splitting, data layout transformations, e.g., Data Reuse. During its development we found that recipes that are not restricted by data dependencies, such as Loop Normalization, are directly applicable, have low impact on performance and are used to enable other, often data dependent, transformations. In terms of automation, one characteristic that makes it more useful for a given recipe is the likelihood of human errors being introduced during manual refactoring, which are avoided when applying code refactoring automatically. The biggest impediment found when automating a recipe is, for example, in Data Reuse, to determine how many array elements to reuse in each iteration so that compiler transformations can take full advantage of it. Such is the case for all transformations that can be applied according to a varying factor.

We perform an analysis of the San Diego Benchmark Suite for computer vision, a set of benchmarks in which performance is a critical factor, in order to determine how its performance can be improved in different aspects, mainly execution time. We point out recipes from the portfolio of selected code transformations that can be used in the process, estimating their impact and describing key characteristics of each benchmark that makes it suitable for refactoring. This analysis enables us to understand the scenarios in which performance recipes from our portfolio can be applied, possible constraints and even drawbacks. We find that the benchmarks contain a large number of loops that iterate through two-dimensional arrays, which makes the application prone to loop related transformations. In some cases, performance hotspot function predictably reuse array values between loop iterations, making Data Reuse a possibility as well.

We effectively apply the transformations to the studied benchmarks and profile their execution time, cache behaviour, vectorization data and energy consumption when executing them in two distinct machines, a High Performance Computer and an Embedded System. Overall, results indicate that transformations do not always cause a performance improvement when applied on their own, although, because they might enable more transformations, their combined impact can become significant. More specifically, Loop Splitting can be used to remove data dependent iterations from a loop and combined with Loop Unrolling to produce higher speedups. Both transformations also increase the likelihood of the compiler being able to apply further optimizations as is suggested by the data obtained when using compiler optimization flags. We apply Data Reuse after unrolling loops by specific factors, which further improves execution times in the absence of

compiler performance flags. On the other hand, compiler optimization flags generally work best on unrolled loops with no Data Reuse, especially for larger amounts of reusable array elements. The energy profiling made on the embedded system suggests that the recipes used have a low impact on the power consumed by an application, meaning that the total energy consumed is closely correlated to the execution time.

Keywords: performance engineering, code transformations, profiling

Acknowledgements

These last few years have been a terrific journey, and nearing the end of such an important landmark in my life brings me an overwhelming feeling of pride and relief, but also a nostalgic, bittersweet taste of a closing chapter.

I want to thank my family most of all for the support they gave me to endure this challenge of moving far away from home to pursue my dreams. Above all, I thank you, Mom, for being the best role model I could possibly look up to.

To my girlfriend and best friend, Ana, I am grateful for being here at all times throughout the years, giving me the strength to overcome each challenge and grow as a person. I could not have done it without you.

I want to thank my oldest friends, António and Lourenço, for encouraging me over and over again to not settle with "good enough", for the great laughs and for being amazing people who have and will continue to achieve great things.

I'd also like to cheer to all the other great people who I have met along the way. Vocês sabem quem são.

I am thankful to my supervisor, João Cardoso, for the experience and knowledge given during this study, and co-supervisor, João Bispo, for always being available to clarify my queries and guarantee I had all the conditions to proceed with work.

To all of the above, I promise to keep pursuing great achievements. This is just the beginning.

Obrigado,

Nuno

"That's for all the kids out there who dream the impossible."

Lewis Hamilton

Contents

List of Figures

List of Tables

Abbreviations

ANTAREX	AutoTuning and Adaptivity appRoach for Energy efficient eXascale HPC systems
API	Application Programming Interface
AST	Abstract Syntax Tree
CIF	Common Intermediate Format
CPU	Central Processing Unit
ECC	Energy Consuming Construct
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HPC	High-Performance Computer
J	Joule
MEM	Memory Unit
MJPEG	Motion Joint Photographic Experts Group
QCIF	Quarter Common Intermediate Format
SAD	Sum of Absolute Differences
SDVBS	San Diego Vision Benchmark Suite
SIFT	Scale Invariant Feature Transform
SIMD	Single Instruction, Multiple Data
SPeCS	Special-Purpose Computing Systems
SQCIF	Sub Quarter Common Intermediate Format
VLE	Variable Length Encoder
W	Watt

Chapter 1

Introduction

In this chapter, we introduce performance computing, provide context behind the problem which we tackle in this dissertation, and describe the main objectives of our work and how they have been achieved.

1.1 Problem Statement

Execution time and energy consumption optimizations are essential in embedded computing, as minimization of execution time and energy/power utilization are key to product efficiency. One reason why these optimizations are important is because code deployed into embedded systems will have a considerable impact on aspects such as battery consumption, response time and device temperature. When dealing with portable and smaller devices, non-optimized portions of code can quickly become critically detrimental to the quality of the product as a whole.

In the following sections we describe how developers can take advantage of the compiler to improve performance of an application, and explore how developers can refactor code on a source-to-source basis to optimize performance.

1.1.1 Compiler Level Optimizations

During code compilation, compilers have an opportunity to refactor the code structure and improve its performance without changing its mode of operation [3]. This process can be triggered by selecting compiler flags [23]. These transformations improve execution time, as well as code size, and, depending on the flags used, compilation time can increase.

To improve performance, the compiler performs and automated search for known characteristics in code that can be refactored in a more efficient format. This means that detection is a critical aspect of the optimization process, as recipes that search for more specific characteristics become more difficult to implement and more targeted to those specific scenarios.

1.1.2 Source-to-Source Optimizations

Source-to-source transformations [17, 19, 47] consist of applying a strategy to the input code and generating a modified version before potentially compiling and executing it. This method is the most commonly used and enables developers to apply a strategy according to the characteristics of a portion of code, e.g., modify a function depending on its parameters. Because the code is not modified during its compilation, this approach is independent of the compiler.

This is where source-to-source compilers [5, 15] become relevant to this dissertation, as we focus on studying the impact of applying performance recipes at source code level. Such compilers can help us define our strategies and apply them to the source code.

1.2 Motivation

Performance aware programming is an essential approach to many fields of software engineering, as applications which consume less resources generally provide better user experience, because we intuitively link shorter execution times of an application and a device's longer battery life to better products. Moreover, the performance of an application is entirely dependant on the design constraints of a product, e.g., if it has virtually unlimited power supply and component cooling capabilities, the energy and power it consumes may become less relevant to its overall performance.

The performance metrics of an application can be improved by the means of software refactoring. Such transformations applied to a code base are what we call "Performance Recipes", as these can be applied under different scenarios, follow a set of implementation steps and have an expected result. Transformations can result in a faster execution time at the expense of increasing the usage on a given hardware component, e.g., increasing the number of accesses to the memory. This means that, if the objective is to decrease the usage on a component, the shorter execution time is not necessarily a performance improvement.

Our motivation to perform this study is to understand the link between given transformations and the overall performance of an application. After a code base is refactored, its behaviour and properties can change drastically, which has an impact in performance. We mainly seek to understand how a performance recipe affects the execution time of an application, but also the impact in other metrics, such as the energy consumption and the size of the code base in memory. During this research we intend to learn in which aspects a transformation improves execution times and the potential drawbacks it has in other metrics.

1.3 Goals

The first main objective for this work is to select and propose a portfolio of performance recipes and develop strategies to apply them on a source-to-source basis, either manually or by the means of a compiler. A portfolio of recipes is a valuable asset for developers interested in high-performance

computing, as it lists a set of recipes, describes how and when to apply and explains what results to expect. The selected transformations must be relevant in a source-to-source code refactoring context or enable other transformations, as the objective is to include recipes that would not be applied by compiler or at least enable such transformations.

The second goal of this dissertation is to analyse a set of benchmarks and determine how the selected recipes included in our portfolio can help improve their performance. Understanding how performance recipes can be applied to a relevant collection of benchmarks further validates the pertinence of the proposed portfolio. The application used for this part of our work must have critical performance requirements in the studied metric, so that it reflects real scenarios where code refactoring is essential.

Our final goal is to apply the code transformations to the benchmarks where needed and empirically measure the behaviour of both the original and modified versions to determine the impact of the optimization. Having this information lets us point to possible causes of improvement or lack thereof, while learning about what, where and when transformations can be applied. Given the pertinence of the application used, the data obtained in this stage is proven to be relevant for future studies.

1.4 Contributions

This dissertation provides the following contributions:

Portfolio of Performance Recipes: develop a portfolio of existing performance recipes that states how to implement each one, as well as the expected advantages and drawbacks. For each recipe, we explore the possibility of developing an automated implementation script.

Analysis: profile an application and determine its main characteristics, detecting the transformations that can be applied and estimating the impact in performance.

Empirical Data: apply the selected recipes to the application and profile its performance, obtaining empirical data on execution times, vectorization data, cache behaviour and energy consumption.

1.5 Document Structure

In Chapter 2 of this document, we present previously conducted studies related to our research. This gives us a perspective of existing techniques and data so we learn about the transformations we can apply and what to expect them based on previous studies.

In Chapter 3, we present the methodology we proposed to use in our research, as well as the existing tools that can assist us in each step of the process.

Chapter 4 is where we propose our portfolio of recipes, succinctly describing each one, as well as describing the expected results and evaluating the possibility of automation.

In Chapter 5 we analyse the selected applications. In our analysis, we state the main characteristics of the code base and point out recipes from our portfolio that can be applied to them, while estimating the inherent impact in performance.

In Chapter 6, we explore the results obtained from profiling the application, interpret them and determine possible correlations and causes.

Lastly, in Chapter 7 we conclude the dissertation, overview the achieved results, review our approach and point out our main contributions, possible improvements and future work.

Chapter 2

Related Work

To optimize both energy consumption and execution time, many studies have been conducted on the impact of transforming code. Studies performed on this subject are usually very specific to one transformation or class of transformations and can have the goal of ascertaining the possibility of implementing a given transformation, estimating performance impacts or actually measuring improvements from the original code to the modified one. This generally implies a more challenging approach for some recipes over others, especially when some precautions and potential drawbacks are involved.

In Section 2.1, we introduce studies conducted on improving applications' execution time and respective results.

Secondly, in Section 2.2 we analyse and impact of performance recipes when it comes to minimizing energy consumption. Generally speaking, performance recipes aimed at minimizing energy consumption also have beneficial results in execution time.

In Section 2.3 we take a look at some processor manufacturers' guidelines, including the recommended practices for their respective processors' architectures and the code transformations implemented by their compilers.

2.1 Execution Time

In this section we take a look at performance recipes for execution time optimization. Some of these techniques involve simple changes to the source code and can be implemented with practically no risk of compromising the integrity of the algorithm, i.e., changing its logic so that it performs a different operation. On the other hand, more complex recipes require changes to multiple sections of the code or to the way data is accessed in memory and, therefore, need to be carefully implemented as to not defect code sections or lose data. Execution time metrics are extracted by including timers around code sections i.e., completely within the software, which can be done without affecting its performance or the validity of the results [4, 15].

In Section 2.1.1 we explore loop optimizations, one of the largest classes of optimizations, and how they exploit different architectures in order to achieve better performance and also the

drawbacks of implementing said transformations. Section 2.1.2 describes studies that used transformations that manipulate the way code is structured to improve performance. Lastly, Section **??** describes work done on Function Inlining.

2.1.1 Loop Optimizations

Loops are one of the most universal and common programming concepts. They are used to perform the repetitive tasks of an algorithm and are very likely to be the cause of a time consumption hotspot and, therefore, loop optimizations are one of the most common recipes for performance optimization [7, 8, 18, 19, 47]. In this section, we analyse state of the art work related to performance optimization using loop transformations.

Loop Unrolling

Loop Unrolling consists of replicating the contents of a loop while applying the appropriate adjustments to the related array indices and loop increments [18]. It always respects the program control data dependencies [7] and, therefore, does not compromise code integrity.

Dongarra and Hinds [18] report that a larger unrolling factor, i.e., the number of times the loop contents are replicated, results in a more significant impact in performance relative to the rolled version of the code.

Cardoso and Diniz [7] propose a model that estimates the impact of full loop unrolling in the execution time and resource ratio without explicitly performing said loop in a C program compiled using the eXtreme Processing Platform - Vectorizing C Compiler (XPP-VC) [9]. They find that, despite better performance, loop unrolling can have a negative impact on the resources needed over resources available ratio. As an open issue of the paper, it was determined that although loop unrolling usually leads to other optimizations, its impact is hard to estimate without performing the transformation.

So et al. [47] propose that the number of cycles in relation to the inner loop unroll factor behaves similarly to a multiplicative inverse hyperbole for a given outer loop unroll factor and the bigger the latter, the wider the aperture of said hyperbole becomes. More specifically, this effect is the most noticeable in Finite Impulse Responses, but is also present in other studied multimedia kernels (e.g., Matrix Multiply and String Pattern Matching).

In addition to removing the loop overhead, Loop Unrolling exposes the loop body to a number of other, more specialized, transformations such as Data Reuse [8, 14], Loop Level Parallelism operations [7, 19, 43], and some compiler optimizations involving memory copying [2].

Dragomir uses Loop Unrolling in [19], combined with Loop Fission, Loop Shifting and Loop Skewing, to propose improvement methods in Kernel Loops (K-Loops) used in signal processing algorithms. We take a more in-depth look at Loop Fission, Loop Fission and Loop Skewing in further sections. Amdahl's Law is used as a point of comparison between the maximum theoretical speedup and the obtained speedup.

The highest obtained speedups, i.e., the ratio between the baseline version and an improved version of code, were of 10.27 using an unroll factor of 7 for the Discrete Cosine Transformation (DCT) and a 9.56 speedup with a factor of 4 for the Sobel Convolution kernels, constituting, respectively, 51.40% and 70.90% of the theoretical values.

The time over area, and vice versa, optimized Sum of Absolute Differences (SAD) kernels had a speedup of 5.12 with a factor of 6 and 7.08 with factor of 5, respectively. SAD kernels were the farthest from the Amdahl's maximum, with 14.51% for area and 20.06% for time.

Lastly, the Quantizer kernel Q-8 became 2.32 times faster with a factor of 1, proving that "Loop Unrolling is used for exposing hardware parallelism, and enabling parallel execution of identical kernel instances on the reconfigurable hardware." [19].

Loop Fusion and Loop Fission

As the names suggest, Loop Fusion and Loop Fission are opposite transformations. Loop Fusion [43, 33, 35] consists of the merging of two loops which iterate with the same properties into a single loop and Loop Fission being the inverse action [8]. It is easy to see how Loop Fusion can improve performance in terms of execution time by promoting variable reuse in code, minimizing loop overhead, enhancing data locality and reducing the total number of loop iterations taken to compute an algorithm [16, 22, 41, 42]. However, as Dragomir [19] mentions, merging loops can be detrimental to performance if, for instance, the memory architecture favours the initialization of different arrays in separate loops. In such cases, the inverse strategy can be applied.

Loop Fission or Loop Distribution [43] consists of breaking a single loop into multiple loops and can be considered an enabling transformation, i.e., a transformation that may enable the application of a subsequent technique. With this respect, Dragomir [19] mentions that Loop Fission, along with Loop Shifting [8, 42], and Loop Peeling [8], can be used to break possible loop-carried dependencies, enabling other strategies such as Loop Unrolling. It can also enable transformations such as Loop Permutation, and be used to improve data locality in a variety of cases [33].

Dragomir [19] uses Loop Distribution to split large loop bodies in order to potentially enable parallelization with Loop Unrolling and Loop Shifting. Experimental results were obtained from a Motion JPEG (MJPEG) algorithm consisting of three Kernels, DCT, Quantizer and Variable Length Encoder (VLE). The performances of an unrolled and shifted/K-pipelining version of the loop with and without distributing its body beforehand were compared. They found a higher or equal speedup when using Loop Fission relative to when not using it. This corroborates that applying Loop Fission to a loop with a large body has a positive impact in performance when subsequently unrolling and shifting/K-pipelining it, given that different kernels can benefit from different unroll factors.

Figure 2.1 displays two simple loops which are the fused/distributed versions of each other, Figure 2.1a being the result of applying Loop Fusion to the code of Figure 2.1b, and Figure 2.1b being the result of applying Loop Distribution to the code of Figure 2.1a.

```
(...)
for(i = 0; i < N; i++) {
    for(j = 0; j < M; j++) {
        sum += V[i][j];
        sub -= V[i][j];
    }
}
(...)
```

(a) Fused Loop

```
(...)
for(i = 0; i < N; i++) {
    for(j = 0; j < M; j++) {
        sum += V[i][j];
    }
}
for(i = 0; i < N; i++) {
    for(j = 0; j < M; j++) {
        sub -= V[i][j];
    }
}
(...)
```

(b) Distributed Loop

Figure 2.1: Simple example of Loop Fusion and Loop Fission

Other Loop Transformations

There are numerous more techniques around loops, usually easier to implement and often used as enablers for other recipes [8]. In this section, we briefly cover such loop transformations, and how they helped state of the art studies.

Loop Splitting [43] differs from Loop Fission by splitting one loop into two or more with the same body but distinct iteration domains, instead of loops with the same iteration domain and distinct loop bodies. This allows the user to treat problematic iterations separately, e.g., a frame of size S in the boundaries of a matrix where the innermost values are subject to logic dependant indices, removing the need of checking whether a given iteration corresponds to the inner region or to the frame of the matrix. Figure 2.2 shows an example of a simple loop being split into two iteration domains

```
(...)
for(i = 0; i < N; i++) {
    for(j = 0; j < M; j++) {
        sum += V[i][j];
    }
}
(...)
```

(a) Original Loop

```
(...)
for(i = 0; i < N/2; i++) {
    for(j = 0; j < M + i; j++) {
        sum += V[i][j];
    }
}
for(i = N/2; i < N; i++) {
    for(j = 0; j < M + i; j++) {
        sum += V[i][j];
    }
}
(...)
```

(b) Split Loop

Figure 2.2: Simple example of Loop Splitting

Loop Shifting is a useful technique we can use to remove data dependencies between software

and hardware functions [8, 19]. It consists of moving the operations of one iteration of a loop to the previous iteration, adding a copy of the moved operations to the loop prologue in order to preserve code integrity. The study described for Loop Unrolling by Dragomir [19], also suggests a speedup of up to 18.70 when applying Loop Shifting to K-Loops. Dragomir also states that, when combined with Loop Unrolling, Loop Shifting consistently improved or, at worst, maintained performance.

Loop Skewing [43] can be used to remove dependencies in a nested loop iterating over a multidimensional array. A demonstration of the technique being implemented on a simple loop is shown in Figure 2.3, where we assume each iteration of j depends on the previous and current value of i. Just like in other transformations, the removal of said dependencies enables loop unrolling.

Dragomir performed an analysis [19] on one of the hotspots parts of the H.264 video codec, Deblocking Filter (DF), with a combination of Skewing and Unrolling generating increasing speedups for pictures of higher resolution and for higher unroll factors. Results also suggest that applying loop skewing without further unrolling the loop becomes detrimental for performance, moreover implying that this technique is mainly an enabling transformation.

```
1 (...)
2 for(i = 0; i < N; i++) {
3     for(j = 0; j < M; j++) {
4         sum += V[i][j];
5     }
6 }
7 (...)
```

(a) Original Loop

```
1 (...)
2 for(i = 0; i < N; i++) {
3     for(j = i; j < M + i; j++) {
4         sum += V[i][j - i];
5     }
6 }
7 (...)
```

(b) Skewed Loop

Figure 2.3: Simple example of Loop Skewing

Loop Peeling is a special type of transformation that moves the first or last few iterations outside of the body of a loop, as shown in Figure 2.4. It can be used to enable parallelization or match the number of iterations of different loops to perform Loop Fusion [19].

Loop Permutation [43] consists of interchanging the nested loops, as shown in Figure 2.5. The permutation of the iterating order of a nested loop can favour cache locality and the compiler's data access patterns to help it automatically vectorize or unroll the loop, either fully of by a factor.

Loop Normalization [43] consists of changing a loop's start and end values so that its control variable initial value is 0 (zero). This helps both compilers and developers detect the number of iterations the loop performs, which facilitates other transformations. Figure 2.6 illustrates a simple example of Loop Normalization being applied to a loop with an original starting value of 1.

```
(...)
for(i = 0; i < N; i++) {
    for(j = 0; j < M; j++) {
        sum += V[i][j];
    }
}
(...)
```

(a) Original Loop

```
(...)
for(i = 0; i < N; i++) {
    sum += V[i][0];
    for(j = 1; j < M; j++) {
        sum += V[i][j];
    }
}
(...)
```

(b) Peeled Loop

Figure 2.4: Simple example of Loop Peeling

```
(...)
for(i = 0; i < N; i++) {
    for(j = 0; j < M; j++) {
        sum += V[i][j];
    }
}
(...)
```

(a) Original Loop

```
(...)
for(j = 0; j < M; j++) {
    for(i = 0; i < N; i++) {
        sum += V[i][j];
    }
}
(...)
```

(b) Interchanged Loop

Figure 2.5: Simple example of Loop Permutation

```
(...)
for(i = 0; i < N; i++) {
    for(j = 1; j < M; j++) {
        sum += V[i][j];
    }
}
(...)
```

(a) Original Loop

```
(...)
for(i = 0; i < N; i++) {
    for(j = 0; j < M - 1; j++) {
        sum += V[i][j + 1];
    }
}
(...)
```

(b) Loop Normalized Loop

Figure 2.6: Simple example of Loop Normalization

2.1.2 Data Layout Optimizations

The way we structure code, namely data structures, arrays and classes, can have a significant impact on the performance of applications, as inefficient data layouts can result in unnecessary data being loaded to the cache, leading to a less time-efficient algorithm. Structuring data in a predictable format for the compiler is also an important step to maximize the performance of our applications, as it enables the detection of sections in which recipes can be applied. These techniques can require a deeper knowledge of the compiler's framework and its data access patterns.

Array of Structures vs Structure of Arrays

A common data layout dilemma is the use of structures of arrays versus arrays of structures as described by Panda et al [38] and Intel [31], which states that in the case of applications using Single Instruction, Multiple Data (SIMD) technology, "performing SIMD operations on the original AoS format can require more calculations and some operations do not take advantage of all SIMD elements available." [11].

The algorithm proposed by Panda et al [38] evaluates the possibility of transforming arrays of structures into structures of arrays without compromising the integrity of the application, in which case, the algorithm (1) transforms each field in the structure into an array, (2) tries to regroup the arrays into a structure more suitable for the memory accesses being made using a clustering algorithm and (3) finally computes the cost of the assignment of a given array to a given cluster. The experiments conducted achieved improvements of up to 44% relative to the original code. Furthermore, data cache miss ratio obtained for the used Fast Fourier Transform (FFT) algorithm suggests this transformation has a bigger impact on systems with lower cache sizes and similar performance on systems with more cache memory.

Object Inlining

Inlining of objects takes advantage of compiler patterns and, if implemented, refactors the code drastically to improve performance.

Proposed by Julian Dolby [17], this technique consists of inlining an object's fields and methods to the class it inherits from. When implementing this recipe, developers have to be mindful of use specialization and assignment specialization. With use specialization, the values resulting from field accesses in the program have to be tagged so that we keep track of the object field that originated them. Assignment specialization prevents aliasing of relations between an object and its container. Dolby ran a collection of benchmarks with the Concert compiler, achieving the best performance improvements on the polyOver benchmark, which reduced the normalized execution time to a third with inlining over without, and a reduction to 50% execution time in the OOPACK. Finally, the other benchmarks showed improvements of less than 15%, proving that, for the used benchmarks, Object Inlining either improves or matches a method's run time. Due to the complexity of this technique, both when it comes to detecting the possibility of implementing it and

because of how difficult it is to create an automatic implementation tool, there is still a lot to be done with Object Inlining.

2.1.3 Function Inlining

Similarly to Object Inlining, this transformation consists of performing a called function's instructions directly in the caller. This transformation aims at improving performance by removing overheads at the expense of the number of lines of code.

Results obtained by Kim et al [34] show that one instance of the Lightness Sensing algorithm increased in size from 16908 to 17558 bytes, while slightly improving its execution time performance. The Simple LED On/Off algorithm, on the other hand, had a 65% performance increase in execution time with an increase in storage from 8300 to 8552 bytes. From this study, we understand that the size of an application could be linked to the maximum execution time improvement possible using this transformation.

2.1.4 Data Reuse

As described by Cardoso and Diniz [8], in the context of signal processing, data is often reused, particularly when an algorithm is applying a signal transformation or repeatedly accessing overlapped sections of an array. Instead of repeatedly accessing the same data, compilers can cache the information in internal registers in the first access and reuse the cached data in subsequent accesses. This is done in an effort to improve performance by decreasing the data access latency and reducing the number of external memory accesses. On the other hand, this technique implies an increase in storage requirements and requires a precise knowledge of the compiler's data access patterns.

One adversity faced when implementing this recipe is detecting code sections suitable for the transformations, as the slightest variation in the loop parameters can hinder the ability for the compiler to safely transform the code without affecting its integrity, even if it is technically possible.

Figure 2.7 shows the result of the application of this transformation to a simple loop. In this example, the number of calls to memory is reduced from 8 stores and 24 loads, all performed inside the loop, to 8 stores and 10 loads, with 2 loads performed outside the cycle and the other accesses being performed within the loop.

2.2 Energy Consumption

In specific cases, developers need to take into consideration the amount of energy that will be consumed by the application being developed. One common category of applications that needs to minimize energy consumption is mobile software, which, if done inefficiently, can drain the device's battery life and become detrimental to the application's quality and, consequently, to the user's experience. Whereas time efficient programming takes a software based approach when

```
1  (...)
2  for(j = 2; j < 10; j++) {
3      a[j] = b[j] + b[j-1] + b[j-2];
4  }
5  (...)
```

(a) Original Loop

```
1  (...)
2  int b_2 = b[0]
3  int b_1 = b[1];
4  int b_0;
5  for(j = 2; j < 10; j++) {
6      b_0 = b[j];
7      a[j] = b_0 + b_1 + b_2;
8      b_2 = b_1;
9      b_1 = b_0;
10 }
11 (...)
```

(b) Loop after Data Reuse

Figure 2.7: Simple example of Data Reuse applied to a loop

it comes to instrumentation by introducing timers before and after code sections being studied, energy profiling requires either the use of electrical hardware or the mathematical computation of the consumed energy based on hardware related factors [37].

A frequent cause of inefficient mobile development comes from software decay [20] as functionalities are changed and bugs are fixed, which results in design flaws and anti-patterns. Morales et al [36] take a look into these problems and propose a refactoring approach for mobile devices which aims to minimize the presence and impact of some major anti-patterns, e.g., Blob Classes and Lazy Classes. The study was conducted on 20 open-source Android applications and achieved an increase in battery duration of up to 29 minutes.

On the other hand, a study on energy consumption for mobile applications [44], conducted by Sahin et al, concludes that the applied performance tips had a statistically significant impact in the energy consumption in only 2 out of 17 of the tested benchmarks and that those which did produce a consistent improvement did not extend the devices' battery life for more than 9.2 minutes. The number of covered changes in each of the relevant usage scenarios seems to suggest no relation between the number of times each tip is applied to an application and the resulting energy consumption reduction, as the tips applied most frequently in these scenarios have similar rates in scenarios with no registered improvements. Hoing et al. [30] identify Energy Consuming Constructs (ECCs) by analyzing approximately 70 source files, written in C, having defined 7 ECCs: (1) Loop Initialization, (2) Math Function, (3) Passing By Structure, (4) Tail Recursion, (5) Global Variables, (6) Escape With Flag and (7) Nested Loop.

Hoing et al state that (1), the initialization of the values of an array within the loop body, consumes a lot of energy due to the comparison operation executed in each iteration. The proposed refactoring technique to eliminate (1) is shown in Figure 2.8. For (2), the authors propose inlining math functions by defining the mathematical operation in a macro, instead of the user created function. To improve (3) it is proposed that a pointer to the address of a data structure should be passed to functions instead of the structure itself. Refactoring recursive function into a simpler iterative statement is suggested to minimize energy consumption by (4). The energy consumed

by (5) can be reduced by performing energy intensive operations using local variables and only assigning results to global variables in the end instead of repeatedly changing the value of the global variable in, for instance, a loop body. To minimize the consumption of (6), the usage of an explicit escape statement, e.g., break, is suggested over a flag variable. Finally for (7), which are commonly used to iterate through a two dimensional array, the simplification of nested loops into a single loop iterating through both variables is recommended. When applied to basic codes from XEEMU [28], transformations produced improvements of 20.08% and 20.34% for (1) and (7) respectively, inlining of math functions, (2), was the least impacting transformation, with a 2.15% reduction and the improvement of the other 4 transformations ranged from 3.98% and 5.41%.

Conversely, when applied as a case study to Lame, an application which encodes/decodes mp3 files from/to other media file formats, the impact of the proposed transformations reduced drastically with savings of 3.5% or less. Although the paper does not mention it explicitly, we're led to believe that this is caused by the low number of changed lines of code (at most 9) in each transformation, compared to the size of the application, with a total of 12367 lines of code. These results are reportedly translated to an increase of 1.5 minutes in battery life per 1 hour of utilisation of a smartphone.

```
1 (...)
2 int array[100];
3 int i
4 (...)
5 for(i = 0; i< 100; i++) {
6     array[i] = 0;
7 }
8 (...)
```

(a) Original Loop

```
1 (...)
2 int array[100] = {0};
3 (...)
```

(b) Refactored array initialization

Figure 2.8: Simple example of the proposed Loop Initialization

2.3 Processor/Compiler Applied Performance Guidelines

Renowned processor manufacturers commonly establish an assemblage of performance guidelines for their compilers and processors. In this section, we take a look at relevant guidelines from Intel and ARM. These guidelines involve many lower architecture level considerations, but we mostly explore higher level source code recommendations.

Intel [31] propose a collection of techniques to optimize applications running on their processors, taking advantage of micro-architectural features of Intel 64 and IA32. The company uses Intel C++ and FORTRAN Compilers, which eases developer's optimization by automatically taking advantage of the target processor's architecture and eliminating the need to write code depending on the processor. The results were obtained using the VTune [32] performance analyzer

to measure performance improvements or setbacks upon using specific recipes. Branch Prediction, proposed first, consists of a collection of techniques and recipes that improve performance, including code alignment and loop unrolling as the relevant ones for this subject. Code alignment consists of predicting the statistic probability of certain blocks of code being ran and moving them to the most suitable section of the program. Loop unrolling is presented as a way of removing branch overhead along with the other advantages mentioned in Section 2.1.1.

ARM [2] also keeps up to date documentation of their suggested guidelines when it comes to the development of applications to be run in their processors. The guidelines include recommended methods of applying performance strategies such as memory copying, which involves loop unrolling, among other transformations. The optimization manual guarantees unaligned load-/store accesses are handled without performance penalties, with few exceptions.

2.4 Summary

The subject of performance and energy aware programming has been intensively studied in the past years with lots of documentation suggesting that it is worth, if not essential, to seek optimal time and energy consumption when developing applications, even though, in some conditions, the improvements found were insignificant.

In some cases, performance recipes do not immediately generate a better performance, but rather make the code more predictable for the compiler to apply its own automatic transformations and that could be the reason why some studies imply lesser results. We can, therefore, interpret studies with lower improvements as indication that applying code transformations will not be effective in all compilers, CPUs or even the algorithms being measured.

One defective characteristic of the study of performance recipes is the lack of use of a unified benchmark suite and measurement approach, making data obtained in distinct studies much more difficult to compare. Moreover, not all publications are clear on the used methodology, benchmarks, machine specifications, architectures, compilers, etc., rendering precise analysis and replication of their research impossible. Consequently, the irrelevancy of some of these variables when correlating data needs to be assumed in this dissertation.

In summary, the state of this art shows that there is still a long path to traverse, as we come to understand more and more the effects of performance-aware programming and this work aims, therefore, at adding insight into the subject.

Chapter 3

Performance Analysis Framework

In this chapter, we describe the performance analysis framework developed to research the impact of performance recipes in a program.

Section 3.1 describes our methodology and the types of tools that can aid in each stage. Sections 3.2 to 3.5 describe each tool selected for each stage. Section 3.7 describes the specifications of the machines we used throughout this work.

3.1 Methodology

For this study, we need to detect performance hotspots in code to determine what transformations can be applied as well as an interface to measure the resulting impact. The flowchart shown in Figure 3.1 illustrates this process. The flowchart represents the work flow we follow to iterate over the benchmarks, instrument their baseline versions and apply relevant recipes from our portfolio of recipes, which we explore in more depth in Chapter 4. Once enough recipes are implemented in a given version of the benchmark, we profile it and record the results. This process is replicated systematically until we have implemented all targeted transformations to each benchmark.

During the benchmark analysis stage, we estimate the worth of creating an automated script to implement the recipe. The idea is to develop such scripts if the transformations in question prove to be possible to automate and valuable for future use. The selected source-to-source compiler and its API are used in the development of the automated scripts and can also be used to execute the various benchmark versions in succession to extract data in bulk.

The criteria to determine if enough transformations have been applied to constitute a new version is dependant on the objective for which we aim in that specific benchmark version, the most obvious example being finishing the implementation of a complex recipe. On the other hand, creating versions with multiple transformations of lower impact is also a possibility, so we can assess their collective improvement or lack thereof. In both cases, the analysis provides useful information.

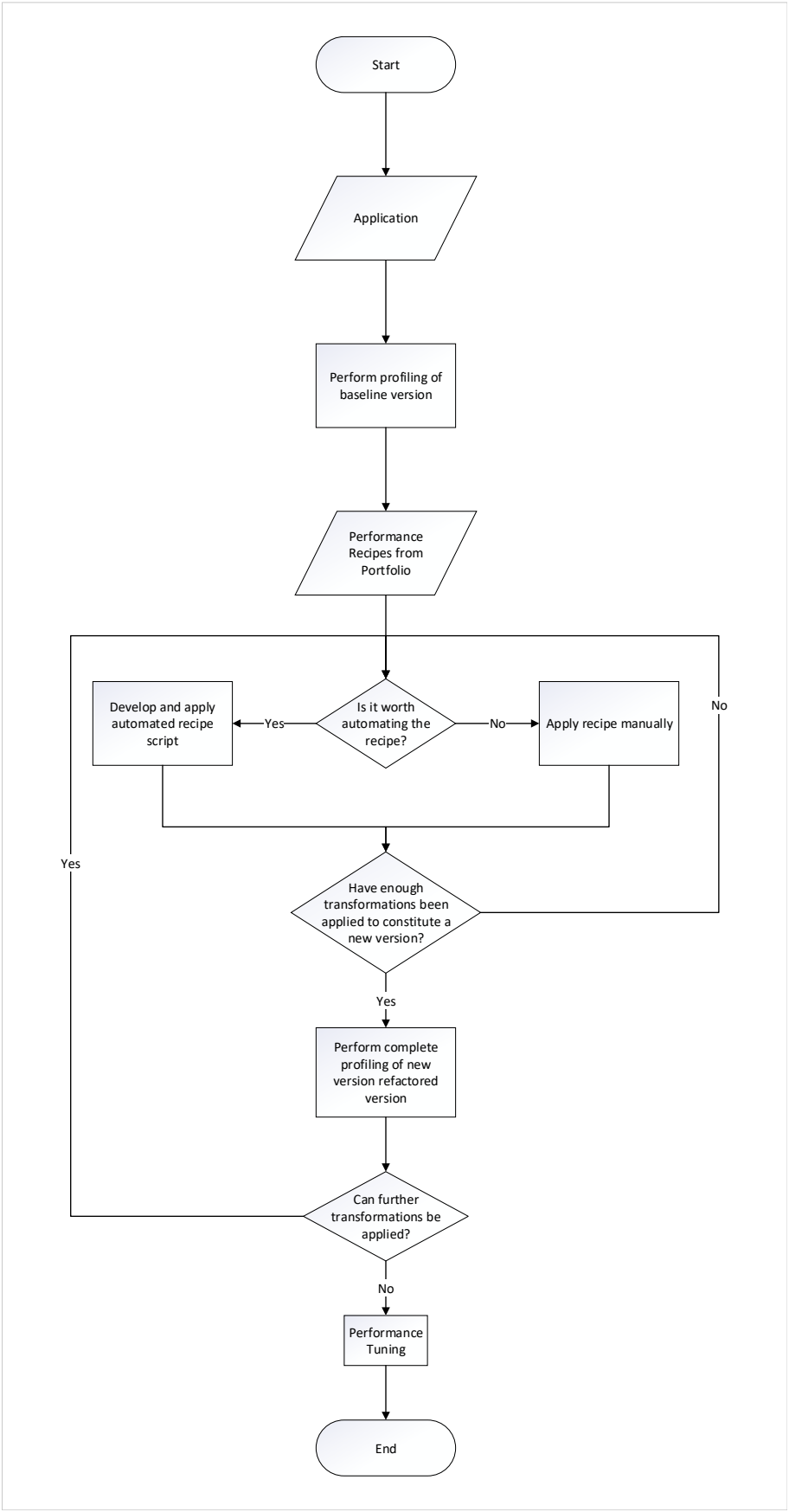


Figure 3.1: Analysis work flow used in our research

We have added Tuning as a final step before finishing our analysis, as a it would be an interesting exercise to understand the best combination of flags and/or transformations. This is, however, subject for future work.

3.2 GPROF and gprof2dot

GPROF [25] is a profiling tool made by GNU which runs on activation of a compiler flag, *-pg*.

The **gprof2dot** [21] script is a python program which converts call graphs from GPROF profiles into a DOT graph [49].

We initially use GPROF to profile our code because it generates a useful call graph of every function called during the execution of a program, along with some useful statistics on the number of times each function is called, the total execution time of all calls of a function and the percentage of the whole execution time which is spent executing each function. By converting the output of GPROF using gprof2dot, we are able to obtain a clear visual representation of the behaviour of our benchmarks and determine which hotspots to tackle for performance improvement.

Despite its usefulness in the initial stages of our work, GPROF adds a significant overhead to other metrics, meaning that a lighter execution time profiling tool needs to be used in later stages.

3.3 Clava Source-to-Source Compiler

Clava [5, 11] is a source-to-source compiler framework developed in the SPeCS [50] group. The tool executes scripts written in **LARA** [6, 10, 12], a JavaScript-based language, that analyses and transforms C/C++ code. Inspired by Aspect Oriented programming languages, LARA scripts are developed as separate entities from existing source code, and applied over an Abstract Syntax Tree (AST) representation of the source code, to which users can add or remove nodes.

A Clava project operates in a typical project directory, and we can provide two distinct paths to it, (1) the source and (2) the output. The tool has a built-in text editor in which it is possible to build a LARA script containing the performance improving strategy we aim to apply. As shown in Figure 3.2, Clava applies the LARA strategy to the C/C++ code and builds a new, hopefully more efficient version of the program, represented by "Modified Code", in the output folder.

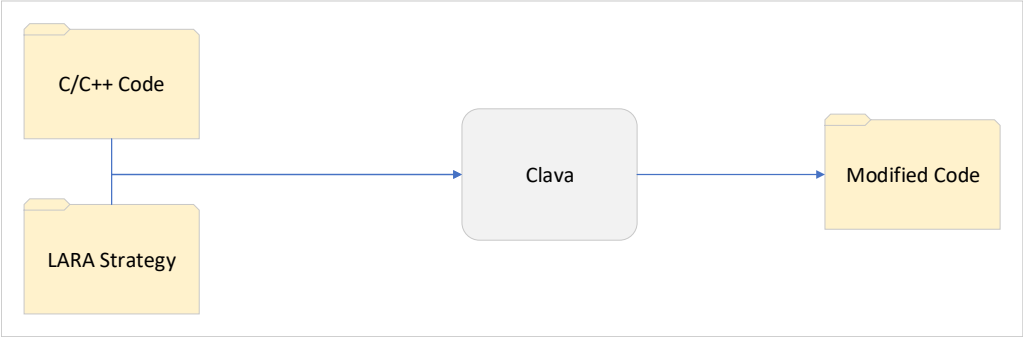


Figure 3.2: Diagram representation of the Clava Framework

```
1  aspectdef QueryCalls
2      for(var $call of Query.search("call")) {
3          println($call.name);
4      }
5  end
```

Figure 3.3: LARA script that queries all function calls and prints their name

The Clava Framework is relevant for the study of performance recipes because it enables us to input potentially non-optimized C/C++ code, provide a collection of optimization recipes, as LARA strategies, and measure the performance improvement of the modified code in terms of both execution time and energy consumption using the available Application Programming Interfaces (APIs). To achieve this, we used the following Clava libraries to achieve each of the respective goals:

- **weaver.Query** — query the generated AST for specific types of nodes, as shown in Figure 3.3. Each node contains specific attributes which can be manipulated using the LARA APIs.
- **lara.code.Timer** — insert timing statements before and after given nodes, as shown in Figure 3.4. The inserted timers add a much lower overhead to the global execution time compared to GPROF.
- **clava.ClavaJoinPoints** — create new nodes or join points in the code, as shown in Figure 3.5. This class can be used to create and manipulate nodes in the AST which result in transformations to the source code.
- **lara.code.Logger** — insert print statements before or after given nodes, as shown in Figure 3.6. This interface enables us to obtain information from code, e.g., variable values.
- **lara.Io** — whereas the previous classes are used in the application of automated transformations, Io has generic utility methods related to read/writing files, and is used to alternate between multiple versions of a program by navigating through the project directory and changing the files used in a given execution, as illustrated in Figure 3.7.

```
1  aspectdef InsertTimers
2      for(var $call of Query.search("call")) {
3          var timer = new Timer();
4          timer.time($call);
5      }
6  end
```

Figure 3.4: LARA script that inserts timers around every function call

```
aspectdef ClavaJoinPoints
    for(var $call of Query.search("call")) {
        var expression = "x / 1000";
        var joinPoint = ClavaJoinPoints.expressionLiteral(expression);
    }
end
```

Figure 3.5: LARA script that creates a join point representing a simple expression

```
aspectdef InsertLoggers
    for(var $call of Query.search("call")) {
        var logger = new Logger();
        logger.text("Logger message before " + $call.name);
        logger.logBefore($call);
    }
end
```

Figure 3.6: LARA script that inserts a print statement before all function calls

```
aspectdef FileIO
    fileAContent = Io.readFile("path/file_A.c");
  Io.writeFile("path/file_B.c", fileAContent);
end
```

Figure 3.7: LARA script that replaces the content of a file with the content of another file

3.4 Valgrind and Callgrind

Valgrind is an open source "instrumentation framework for building dynamic analysis tools" [26], made available by GNU. Among a collection of tools, Valgrind provides Callgrind [27], a profiling tool that records the call history among functions in a program's run as a call-graph.

Callgrind is used to profile the cache behaviour of a program executions. The report generated by this tool after execution displays information on the cache hit/miss rate and counts the instructions executed. In terms of the instruction count, the report also specifies how many of them were stores and loads to memory.

Obtaining these metrics enables us to quantify changes in the total number of instructions executed and assess how performance recipes affect the efficiency of memory accesses in an application. Changes in these metrics can potentially be correlated to improvements or deterioration of execution time.

3.5 Energy Monitor

To profile the energy consumed by the main components of the System on a Chip in the ODROID board used [13], we use the Energy Monitor library provided by SPeCS [50], which is based on the power measures of the CPUs, GPU and DRAM provided by the board.

The library provides a C interface, which we use in our process. The profiler receives one attribute, the delta value for the power measurement in microseconds, and returns a collection of samples that represent the average power consumed by the Central Processing Unit (CPU), Graphics Processing Unit (GPU) and the Memory Unit (MEM) in a given instant. The power values are returned in Watts (W). We can also obtain the execution time value by multiplying the number of samples with the chosen delta. We compare the result with the execution time profiling value for a given benchmark version, simultaneously validating both profilings' data. We can then use this execution time value to calculate the energy consumed in each unit, in Joules (J), by multiplying the execution time with the average power.

This library gives us useful insights into the impact of performance recipes in terms of both power and energy consumption. The power consumption results allow us to estimate if the applied transformations increase the likelihood of one of the hardware components overheating, whereas the energy consumption results can tell us how the battery life of a portable ODROID embedded system would be affected by the applied transformations.

3.6 Open Tuner

Open Tuner [1] is an extensible framework for auto-tuning, which allows the development of tuning strategies in Python to detect aspects such as the best combination of flags to use when compiling an application or the best unroll factor to apply to a loop.

Although we have not developed a tuning strategy, we identify it as an interesting next step in our research and consider Open Tuner a useful tool to use in this process.

3.7 Machine Specifications

Over the course of this work we used two distinct machines to extract metrics from each benchmark execution:

ANTAREX [45, 46] - A High-Performance Computer (HPC) with 2x Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz CPU and 4x 16GiB DIMM Synchronous 2133 MHz RAM. We decided to use an HPC node to extract metrics in the initial stages of our research due to its more advanced specifications, as it would be the easiest and quickest method of assessing our results before executing any benchmark in an embedded system. The compiler used in this machine is GCC [24], version 7.5.0.

ODROID [13] - An embedded system with a Exynos 5422 Cortex™-A15 2.0 GHz CPU and 2 GByte LPDDR3 RAM at 933MHz. Ultimately the scope of our work requires us to study the impact of performance recipes in an embedded system, therefore we also measured the ODROID's performance executing our final version of the improved code. The compiler used in this machine is GCC [24], version 6.5.0.

These systems have distinct processing architectures and computing power, therefore, their behaviour to the same input is likely to vary. The definition of a performance improvement is also distinct in each machine, as our goals is to not deteriorate ODROID's power consumption because it is an embedded system, whereas ANTAREX has no such constraint.

3.8 Summary

Having a consistent methodology and using the right tools is essential for any study. Therefore, before refactoring code and extracting metrics, we determined the process to follow during our research.

We have described the essential aspects of our methodology, determined the tools to use for execution time profiling, transforming and instrumenting code on a source-to-source basis and cache profiling. We have also looked at the available machines and respective specifications so we can understand how their computing power affects their behaviour when measuring the impact of performance recipes.

Chapter 4

Source-to-Source Code Transformations

A transformation is applicable if certain conditions are met in the original version of the code, and has a linked expected result on a variety of aspects, mainly in execution time, but also in cache behaviour. After studying existing code transformations, we selected a number of interesting ones that could be applied on a source-to-source basis, either individually or in combination with each other.

Section 4.1 provides context about the selection process and chosen techniques. In each of Sections 4.2 to 4.12, we translate the respective transformations into recipes which describe the code characteristics that make an algorithm suitable for a performance improvement strategy, the expected behaviour of the refactored code and the possibility of automating them. Section 4.13 describes the automated scripts we developed to implement some of the selected recipes.

4.1 Context

After researching about existing code transformations, we need to choose the ones to use in our study of performance optimization. Our selection has the goal of including a strong variety of techniques ranging in aspects such as complexity and expected impact. Table 4.1 displays each selected transformation and respective references.

4.2 Replace Doubles with Floats

When real numbers are used in a C expression, e.g., `x = 0.5`, they are treated as *double* literals, which adds potentially unnecessary precision to calculations in detriment of performance. To avoid this, real numbers can be used as *float* literals by using `x = 0.5f`, in the previous example.

The applicability of this transformation depends on the presence of decimal literals in the algorithm and its simplicity makes the development of a recipe very straight forward. As long as the program does not depend on the precision provided by a *double*, this transformation will not compromise its integrity.

Transformation	References
Replace double with float	N/a
Replace pow() calls with multiplication	N/a
Loop Normalization	[8, 43]
Loop Peeling	[7, 8, 18, 19, 47]
Loop Splitting	[8, 43]
Loop Permutation	[8, 43]
Variable Specialization	[43]
Multiple Function Versioning	[43]
Function Inlining	[43]
Data Reuse	[8]

Table 4.1: References to selected recipes

A major downside of manually applying this transformation is that instances of *double* literals can be easily missed, especially if a program involves different developers working on separate files. On the other hand, due to its simplicity, automating this technique is not difficult, provided the correct tools. For instance, a source-to-source compiler that is able to query an AST for such expressions can help detecting and transforming promising regions of code. Also due to its simplicity, minimal performance improvement resulting from this transformation is expected unless it is applied a significant number of times.

4.3 Specialization of pow Calls

The Math.h function $pow(a,b)$ [40] is essential to perform the product of a with itself b times in cases where the value of b is an integer and not known at compile time. Although, for instance, despite making the code more readable than $y*y*y*y*y$ for a developer, the expression $x = pow(y,5)$ can be detrimental for performance, due to the added function call overhead. Moreover, performing the operation in its inline format, instead of a function call, helps the compiler apply further transformations, such as Loop Unrolling.

Logically, this transformation is applicable whenever the function in question is used, and its attributes are of type *double*, which means there is no technical restriction for replacing this function call with an arithmetic operation, as long as the exponent's power has a known value.

Detecting instances of the $pow()$ function being used is relatively easy, but not all of them are valid for this transformation and filtering invalid occurrences can become inconvenient for developers if the algorithm has a lot of them. Automating this transformation is, therefore, of interest. This can be achieved by searching for calls to the function and applying it in the ones where the second attribute is a literal.

4.4 Loop Normalization

This transformation consists of changing a loop's start value to 0 (zero) and subtracting the original start value to the end value in order to maintain the number of iterations consistent. This also implies adding the original loop's start value to every use of the control variable in the loop body so that the array positions accessed also remain the same as originally. This transformation can be used to help the compiler detect the exact number of iterations of the loop or improve code readability and interpretability for future analysis and application of other transformations.

Any loop which starts its control variable at a value different than zero can be normalized, even for non literal start values. An automated Loop Normalization script would have to detect loops with such conditions and accordingly change their start and end value parameters and control variable instances in its body.

4.5 Loop Peeling

Peeling a loop by a factor of n consists of explicitly performing n iterations of said loop outside of its body while adjusting its start or end values to guarantee the integrity of the code, i.e., not changing the operation performed by the algorithm. This transformation has very little to no impact on performance when used on its own, but it can be used to enable other transformations. E.g., moving problematic iterations outside of a loop in cases where the compiler is not able to apply vectorization.

Developing a script that automatically peels a loop by an ideal factor is complicated, as it is often used to enable other transformations. Unless the subsequent transformation is known, the peeling factor needs to be determined by the user in some way. The automated script could run the application once, analysing limitations in terms of parallelization, for instance, but that would be just one of many possible scopes of analysis.

Once determined the peeling factor n, on the other hand, the transformation can be applied by inserting a copy of the loop body before/after the loop n times and adjusting the start/end value by adding/subtracting n to it. The inserted copy of the loop body then has to have the control variable replaced with the correct n iteration indexes, if needed.

4.6 Loop Splitting

Loop Splitting consists of dividing a loop body over two or more distinct iteration domains. This transformation is not expected to improve performance by itself, but rather change the code structure so that other transformations become applicable.

Any loop can be split as many times as its total number iterations, as long as all the original and only the original iterations are performed, in the same order and the resulting loop bodies remain unchanged.

Automating this transformation is tricky, because the ideal number of split loops and respective domains is directly related to the subsequent transformations a developer wants to perform. At this level, it becomes practically impossible to predict the best possible parameters to use in this transformation, without human input, e.g., the iteration domains. Nevertheless, the automation of this technique can be specialized for algorithms with an arbitrary set of characteristics, creating a trade-off between global and algorithm specific usefulness.

4.7 Loop Fission

Loop Fission or Loop Distribution consists of dividing a given loop body between at least two loops with equivalent iterative domains. Applying this transformation by itself could slightly hinder performance, as an added loop overhead is created for each distributed loop, but it can be used to enable other transformations.

A loop can be distributed if its body contains at least two independent instructions. The resulting loops should execute the same number of iterations and each instruction in the original loop body should be executed the same number of times.

4.8 Loop Permutation

Loop Permutation consists of swapping two nested loops with each other to favour the order of memory accesses. This transformation's efficiency highly depends in the architecture of the processor used to execute the program.

This transformation can be applied to any pair of nested loops that does not have any statement that is exclusive to the outer loop, i.e., is inside the outer loop and outside the inner loop. Logically, only nested loops that operate over arrays of two or more dimensions can take advantage of this transformation in terms of performance, but as long as the aforementioned conditions are met, there are no technical impediments.

This recipe can be automated on any tool that can detect nested loops with no mutually exclusive statements and swap the loop parameters with one another.

4.9 Specialization

When creating a function, developers might create a dynamic variable declaration for a value that ends up being constant. In such cases, the variable can be specialized so that, instead of being calculated, it is assigned a literal value as soon as it is appropriate. This removes the overhead added by the calculations and helps the compiler to detect the variable value at compile time. The latter advantage, can facilitate the implementation of compiler transformations such as Loop Unrolling or vectorization. This transformation, therefore, always has a positive impact in performance even if very small.

As long as the pre-requisites for this transformation to be applied are met, there are no limitations in its implementation. Automating this recipe is tricky, as multiple executions of the full program have to be performed with different inputs to ensure that a variable can be specialized.

4.10 Multiple Function Versioning

This transformation consists of creating multiple versions of a function that differ on a given aspect and are executed each on certain conditions. A function that, for instance, has an input array with three possible length values can be divided into three functions for each possible array size. This enables the explicit attribution of the array length for all three cases - Specialization, as described in Section 4.9 - and can enable other transformations.

A function can be specialized into multiple versions if one of its variables has a predictable set of values. There is no requirement to specialize a function for all possible values of the variable in question, as long as each specialized version is called only when appropriate. This may involve testing or calculating a value before calling the function. The function calls are, therefore, a concern when applying this transformation. E.g., if a function loop through N iterations and we can isolate the function calls where N has a given value, it could be worth creating a version for this specific value of N.

Automating this transformation would be difficult without user input, as it requires a post-execution analysis of each variable to determine which ones are predictable. This analysis has to be performed even when applying the transformation manually, therefore, the effort and time that an automated script would save a developer would be in the code refactoring stage of the recipe. That being said, automating this transformation would likely be very valuable, especially for larger code bases, in which the transformed function is called many times.

4.11 Function Inlining

Inlining a function consists of inserting the instructions of the calling function directly in the place where the function is called, instead of calling the function. This technique is particularly useful if the called function has a particular behaviour in a known scenario and we want to apply other code transformations to it without compromising the code integrity or versioning the called function.

Technically speaking, any function can be inlined in its calling place in detriment of code readability. Inlining all functions in a program would result in a single, potentially long, file containing exclusively the main function. When applying this transformation, all uses of the function's parameters have to be accordingly changed to the values passed in the original call. Variables with similar names also have to be taken into consideration in order to avoid loss of information. Libraries used in the inlined function also need to be included in the caller to maintain code integrity.

An automated version of this recipe would be useful as it could be run as the last step before deploying a program to, for example, an embedded system. Receiving user input enables developers to specify which functions to inline. The script would have to consider all of the previously mentioned aspects and be able to receive or interpret input from developers, e.g., using *pragmas*. Although Clava APIs contain interfaces that implement this recipe, do not make use of them, as inlining a whole application can become detrimental to performance due to the increase in the number of instructions. We therefore only apply this transformation manually or using compiler optimizations.

4.12 Data Reuse

Data Reuse can be applied to a loop which loads similar elements of an array, i.e., sliding window, over consecutive iterations. Because in any given iteration some of the used array elements are the same as in the next iteration, they can be stored in scalar variables and be reused as long as the sliding window encapsulates them. The increased number of registers needed to execute the transformed code could hinder transformations applied by the compiler involving parallelization. The main advantage of this transformations is, therefore, the reduction of the number of memory accesses performed by the loop, which is expensive in terms of execution time.

Loops that use more than one element of the same array in any given iteration and have a predictable and continuous array access pattern are, therefore, a potential target for Data Reuse. This transformation almost always requires peeling the loop by a factor equal to the number of reused array values. On the other hand, improvements are not guaranteed, especially for larger amounts of reused array elements, in which cases register spilling might occur and hinder cache performance and, consequently, execution time.

Full automation of recipe is not necessarily valuable, because there is no guarantee that the performance will not deteriorate, even if data from the loop in question can technically be reused. In any case, an automation script could be created for simple scenarios where n array values are accessed in consecutive iterations. The script would need to create n new variables and use them to store the array element values involved in each iteration.

4.13 Automated Transformations

As we explored in previous sections, there is a significant value in automating some performance recipes. The main goal of automation is to remove the risk of human introduced errors in the refactored code. Moreover, if a script can detect the possibility of implementation of a transformations it can reduce the time spent by a developer looking for those portions of code.

We propose three scripts that implement the three performance recipes described in Sections 4.2, 4.3 and 4.6 and one script that combined the recipes described in Sections 4.6 and 4.7. These scripts can be executed in Clava and are written in LARA, which we introduced in Chapter 3.

```
1   Transformations.replaceDoublesWithFloats =  function() {
2     for(var $call of Query.search("call")) {
3       for(var argument of $call.args) {
4         if(argument.type.code !== "double")
5           continue;
6         if(argument.instanceOf("literal")) {
7           argument.replaceWith(argument.code + "f");
8         }
9       }
10    }
11  }
```

Figure 4.1: LARA script that replaces doubles with floats used as argument of functions

We choose replacing doubles with floats and replacing $pow()$ calls with multiplication, because instances of code that can be targeted for such recipes can occur numerous times in a given application. That being said, performing these recipes manually is a repetitive task which is prone to human error and missed instances where they could be applied, therefore the automated scripts exist to prevent that from happening.

Although Loop Splitting or Loop Fission are not applicable as often as the transformations in the other two scripts, they are much more complicated to implement manually, as they involve iterative domain manipulation and loop body refactoring. This makes these recipes prone to human error, therefore the LARA scripts can be used to prevent it.

The first script we propose transforms literals of type *double* into literals of type $float$, as explained in Section 4.2. This is performed by querying the AST generated by Clava for *double* literals and adding the "f" that sets the type of the literal to $float$. Figure 4.1 displays the final version of the algorithm.

The second script we propose replaces calls to the Math function $pow(a,b)$ with the product of a with itself b times, as described in Section 4.3. This is performed as illustrated by the pseudo-code shown in Figure 4.2, where the script queries the generated AST for calls to the $pow()$ function and replaces calls with power 0 with a literal 1, calls with power 1 with the exponent's base and calls with power equal to or greater than 2 with the appropriate multiplication.

Thirdly, we propose a Loop Splitting script, which searches for *pragmas* in the source code specifying the iterative domains for each resulting loop. For each useful *pragma* found, the script creates a copy of the original loop node, sets the copy's initial and end values to the ones passed in the object and inserts the new nodes, replacing the original. Figure 4.3 presents this algorithm in pseudo-code.

Figure 4.4 shows an example of a nested loop being split using our generic approach where the resulting code contains two loops performing half of the initial iterations each, as shown in Figure 4.4b.

The final script we propose implements a combination of Loop Splitting and Loop Fission. The example given in Figure 4.4 creates a potentially unfavourable memory access order, requiring a

```
1
2   Transformations.replacePowCalls =  function() {
3       var changes = true;
4     while(changes) {
5       changes = false; // remains false if there are no initially nested pow()
            calls
6       for(var $call of Query.search("call", {name: "pow"})) {
7         // If power is not a literal integer, no change is supported
8         if(power >= 2) {
9           // Store first 2 multiplication operands in node
10          for (var i = 3; i <= power; i++) {
11              // Add multiplication operand to node
12          }
13          // Wrap node in parenthesis to guarantee arithmetic consistency
14          // Replace call with node
15        }
16        else if (power == 1)
17            // Replace with exponent base
18        else if (power == 0)
19          // Replace with literal 1
20      }
21    }
22  }
```

Figure 4.2: LARA script that replaces *pow*() calls with multiplications

```
1   Transformations.loopSplitting = function() {
2       for each iteration_domain in pragma {
3           //Create a copy of the original loop
4           //Create split loop in iteration_domain
5           //Insert loop over iteration_domain after original loop
6       }
7       //Delete original loop
8   }
```

Figure 4.3: LARA script that splits a loop using pragmas in the source code

```
(...)
for(i = 0; i < N; i++) {
    #pragma clava data intervals: [\
    {\
      startValue: "0",\
      endValue: "M/2",\
    },\
    {\
      startValue: "M/2",\
      endValue: "M",\
    },\
  ]
    for(j = 0; j < M; j++) {
        int start = j;
        for(k = start; k < L; k++) {
            //loop body
        }
    }
}
(...)
```

(a) Original Loop

```
(...)
for(i = 0; i < N; i++) {
    for(j = 0; j < M/2; j++) {
        int start = j;
        for(k = start; k < L; k++) {
            //loop body
        }
    }
    for(j = M/2; j < M; j++) {
        int start = j;
        for(k = start; k < L; k++) {
            //loop body
        }
    }
}
(...)
```

(b) Split Loops

Figure 4.4: Automated Loop Splitting example

distribution of the outer loop wrapping the split loops and potentially giving the compiler a more difficult task of applying its own optimizations. Figure 4.5 illustrates in pseudo-code the automated script's algorithm.

Figure 4.6 provides an example original loop and the resulting loop from the execution of the script. Note that, if the control variable being incremented was that of the outer loop, the refactored code would not result from a Loop Fission, as only Loop Splitting would need to be applied.

4.14 Summary

The first step towards understanding the effect of code transformations on performance is to select the transformations to apply. Each technique can be applied under certain conditions and have an expected impact on the performance of an application. In this chapter, we have proposed a portfolio that describes the implications associated to each performance recipe included. This portfolio can serve as both an introductory piece to developers learning the initial stages of performance aware computing and a guide for the optimizations we explore in Chapter 5.

The selected transformations that manipulate single line instructions, e.g., replace pow() calls with multiplication, have a lower expected impact on performance and a greater possibility of being fully automated.

Recipes developed around loop transformations can typically be applied directly, provided the user inputs useful information. The reason behind this is that loop transformations can be applied to various degrees, e.g., a loop can be peeled by multiple factors, rendering it difficult for

```
    Transformations.loopSplittingFission = function() {
        // Check if only one control variable is being used
        if the control variable being incremented belongs to the outer loop {
            // Set the outer loop as target loop
        }
        else if the control variable being incremented belongs to the inner loop {
            // Set the inner loop as target loop
        }
        else {
            // This transformation cannot be applied
        }
        for each iterative_domain in pragma {
            // Create a copy of the outer loop
            // Get the target loop equivalent from outer loop copy
            // Set target loop parameters to iterative_domain
            // Insert outer loop copy after original loop nest
        }
        // Delete original loop nest
    }
```

Figure 4.5: LARA script that splits and distributes a loop using *pragmas* in the source code

```
(...)
#pragma clava data intervals: [\
  {\
    startValue: "0",\
    endValue: "M/2",\
  },\
  {\
    startValue: "M/2",\
    endValue: "M",\
  },\
]
for(i = 0; i < N; i++) {
    for(j = 0; j < M; j++) {
        int start = j;
        for(k = start; k < L; k++) {
            //loop body
        }
    }
}
(...)
```

(a) Original Loop

```
(...)
for(i = 0; i < N; i++) {
    for(j = 0; j < M/2; j++) {
        int start = j;
        for(k = start; k < L; k++) {
            //loop body
        }
    }
}
for(i = 0; i < N; i++) {
    for(j = M/2; j < M; j++) {
        int start = j;
        for(k = start; k < L; k++) {
            //loop body
        }
    }
}
(...)
```

(b) Split and distributed Loops

Figure 4.6: Automated combined Loop Splitting + Loop Fission script example

an automated tool to detect the ideal degree to use. The outcome of applying these recipes has a low expected impact on performance, although they can enable other transformations - either source-to-source or compiler applied.

The recipes that involve manipulating code structure in terms of function calls or data layouts can have a significant impact on performance, due to their potential in removing function overheads or reducing the amount of accesses to memory. Function Inlining, in particular, could be automated without the need for user input. Conversely, transformations such as Data Reuse or Specialization would be tricky, as they require an analysis of how a program behaves given different inputs.

We have developed in LARA, using Clava, three automation scripts that implement single transformations and one script that combines two transformations. The combined recipe script is aimed at nested loop scenarios in which performance could be hindered by the resulting memory access order.

Chapter 5

Benchmark Analysis

Measuring the impact of a performance recipe requires a clear understanding of the benchmarks used. In our case, we chose the San Diego Vision Benchmark Suite (SDVBS) [52], due to its algorithms in the context of computer vision. Image processing involves looping over two-dimensional arrays and performing operations between given pixels, often using images with different, although predictable, properties. This makes a benchmark suite such as SDVBS a perfect candidate to study the selected transformations.

In Section 5.1, we introduce and make some considerations about SDVBS. Sections 5.2, 5.3 and 5.4, explore the Disparity, Feature Tracking and Scale Invariant Feature Transform benchmarks, respectively. We also describe the initial profiling done on each respective benchmark using the GPROF/gprof2dot [21, 25] analysis framework, described in Chapter 3, point out important details present in their hotspot functions, determine how their code can be refactored using the recipes we selected and mention the expected outcome of those code transformations. The effects caused by the code transformations are explored in Section 5.5

Lastly, in Section 5.6 we describe the changes made to the studied benchmarks when applying the developed LARA automated scripts, explored in Chapter 4.

5.1 Introduction

SDVBS contains ten computer vision benchmarks, from which we have selected three in which to study the impact of the selected performance recipes. These benchmarks have been selected due to their structural and performance characteristics suiting the recipes from our portfolio, which we explain in more detail in the following sections. Each benchmark contains data sets of various sizes, including specifically SQCIF, QCIF, CIF and Full HD, which we use to evaluate how different input image sizes affect the time spent on each benchmark function. The data sets also provide a file with the expected output values that should be obtained from the program execution. These are verified at the end of each benchmark and can be used to confirm that the code transformations implemented did not compromise the integrity of the application.

Before looking into each benchmark, it is important to note the structure of the suite, as it can be divided in two code sections: (1) the common library and (2) the benchmark specific libraries.

(1) contains generic methods that execute tasks such as allocating and freeing memory, reading files and performing semi-complex arithmetic operations. It is also where the data structures used throughout the suite and the macros to access them are declared. The suite has three types of data structures, $I2D$, $UI2D$ and $F2D$, and all of which store the image width, the image height and a one dimensional array of image pixels. The array of pixels is of type *int* for $I2D$, *unsignedint* for $UI2D$ and $float$ for $F2D$. To access the arrays in these structures, the library uses 3 macros, as shown in Listing 5.1. Changing a function in this library affects the performance of any benchmark that calls it, which means that a substantial performance improvement in the execution time of one of these functions would likely impact multiple benchmarks. (2) contains the benchmark specific methods and these vary from algorithm to algorithm, thereby, depending on the common library to perform their roles.

When discussing loops, we can characterise them by their rank, which gives us the level of nesting that they have. For instance loops in a 3 level nest with one loop in each level, can be counted as one loop of rank 1, one loop of rank 2 and one loop of rank 3.

```
1 (...)
2 #define subsref(a,i,j) a->data[(i) * a->width + (j)]
3 #define asubsref(a,i) a->data[i]
4 #define arrayref(a,i) a[i]
5 (...)
```

Listing 5.1: SDVBS macros to access arrays in the structures

5.2 Disparity

This benchmark contains an algorithm which computes the disparity in depth between objects present in two images of the same scene obtained from slightly different positions. This algorithm is often in applications such as pedestrian detection and cruise control [39].

The results of the profiling of this benchmark, shown in Figure 5.1, suggest that the larger the data set used is, the heavier the workload of the function $finalSAD()$ becomes, as its sibling functions consume a lower percentage of the execution time. The fact that the caller function $correlateSAD\_2D()$ consistently consumes approximately 83% of the execution time further validates this observation. That being said, it is fair to consider $finalSAD()$ the main hotspot in Disparity, consuming 40.12% of the total execution time for Full HD input data, as shown in Figure 5.1d.

The hotspot function $finalSAD()$ consists mainly of a nested loop that computes the Sum of Absolute Differences (SAD) of the corner values in a window inside the array, shown in Listing

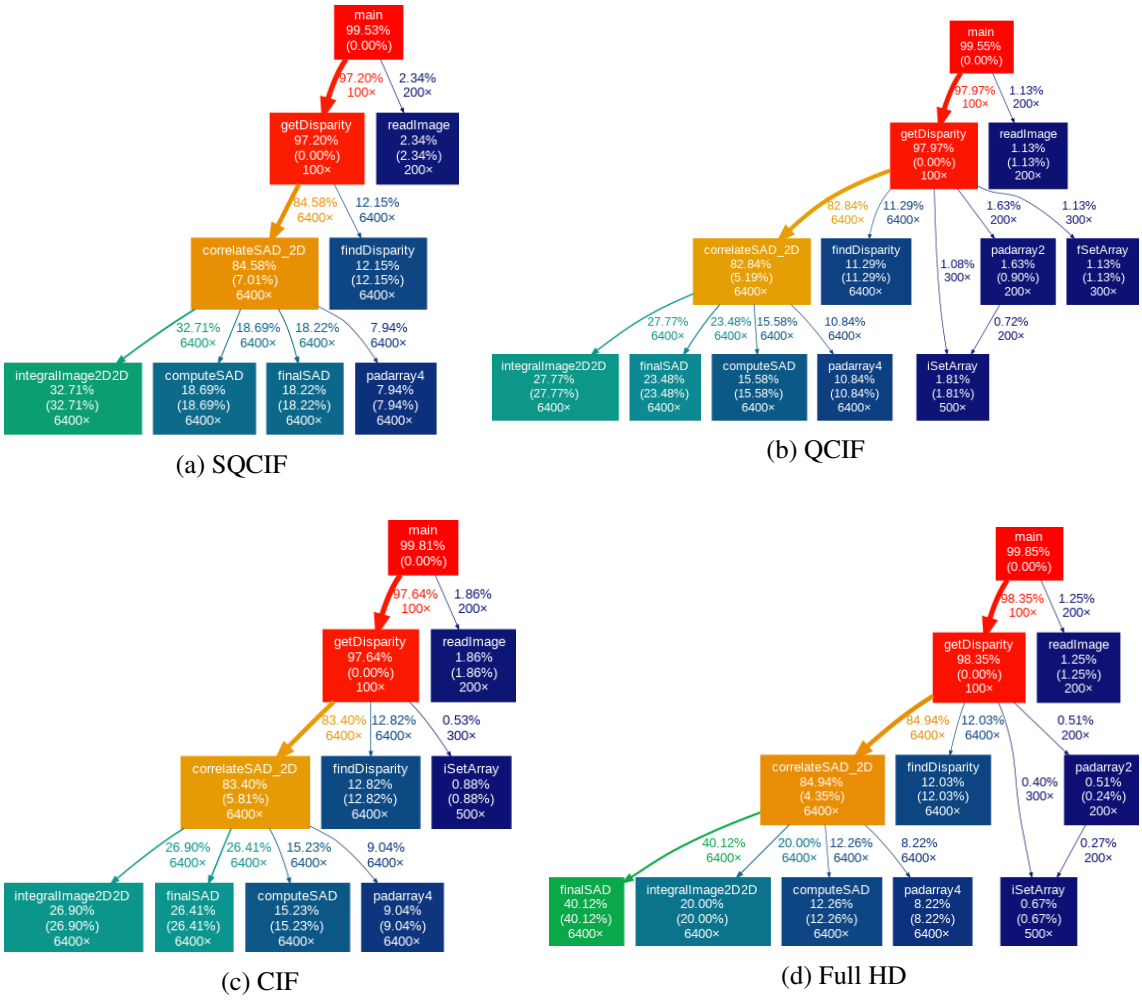


Figure 5.1: Disparity Benchmark Profiling Results from 100 executions

5.2 which makes the hotspot potentially viable for a performance strategy involving loop transformations. Also, the window that determines the values used in the SAD operation, *win_sz*, scrolls through the input array incrementally in the vertical direction before moving horizontally by one position. This means that, under specific conditions, the bottom values included in a given iteration eventually become the top values included in a subsequent iteration.

```
1 for(j=0; j<(endC-win_sz); j++)
2 {
3     for(i=0; i<(endR-win_sz); i++)
4     {
5         subsref(retSAD,i,j) = subsref(integralImg,(win_sz+i),(j+win_sz)) + subsref(
              integralImg,(i+1) ,(j+1)) - subsref(integralImg,(i+1),(j+win_sz)) -
              subsref(integralImg,(win_sz+i),(j+1));
6     }
7 }
```

Listing 5.2: Loop used in the Disparity benchmark hotspot function, *finalSAD*()

Further analysis on this benchmark reveals that the value of *win_sz* is constant and equal to 8 when using Full HD inputs. The number of iterations executed until array values are reused is determined by this variable, which gives us a clear possibility of implementing Data Reuse.

The transformations we applied to this function were, therefore, a Specialization of *win_sz* and Loop Peeling of the inner loop shown in Listing 5.2 by a factor of 7, which enabled Data Reuse in that same loop.

A particularity of this hotspot when it comes to Data Reuse is that, despite only 2 values being reused in each iteration, because they are only accessed for a second time after 8 iterations, there are 14 values that need to be stored in registers for future reused. Of these 14 values, 7 correspond to the bottom right of the sliding window and 7 correspond to the bottom left. Note that the sliding window moves vertically.

Typically a reduction in the amount of memory accesses resulting in a performance improvement in terms of execution time would be expected from the transformations applied, but the amount of new registers that need to be used hinders the likelihood of improvement. This trade-off could have an especially significant impact on machines with lower-end hardware, due to effects such as register spilling.

5.3 Feature Tracking

Feature Tracking is an algorithm used to obtain movement information of an element present in 4 sequential images. It is commonly used in robotic vision and automotive object tracking.

The initial profiling on this benchmark reveals a spread out workload between a variety of its functions, as can be seen in Figure 5.2, with a lot of fluctuation in the percentage execution time consumed by $fillFeatures()$, $imageBlur()$, $calcSobel\_dY()$, $fSetArray()$ and $calcSobel\_dX()$ between all data sets. An analysis of the Full HD results obtained, shows us that the most time consuming function is $imageBlur()$, at 20.95% of the total execution time, as seen in Figure 5.2d. The first aspect to note about the structure of the three hotspots is that the matrix size passed to $imageBlur()$ is always 1080x1920 and the matrix size passed to *calcSobel_dX* and *calcSobel_dY* is always either 1080x1920 or 540x960. The instances where each matrix size is provided to each hotspot is also predictable. This means that Specialization can be directly applied to $imageBlur()$. On the other hand, specializing this variable for *calcSobel_dX* and *calcSobel_dY* requires a prior implementation of Multiple Function Versioning so that the appropriate matrix size can be used each time.

The hotspot function, $imageBlur()$, contains two nested loops with 3 levels. In each nest, the two outermost loops iterate over the input image matrix, and the innermost multiplies the pixel values with Sobel kernels [48] of length 5. This is a very similar structure to one seen in $calcSobel\_dY()$ and $calcSobel\_dX()$, except these functions use kernels of size 3.

Listing 5.3 shows the first loop used in the $imageBlur()$ hotspot. The first aspect to note in terms of applicable transformations is that the kernel values are constant and the innermost loop is iterating over them. Completely peeling this loop exposes the whole kernel in every iteration of the

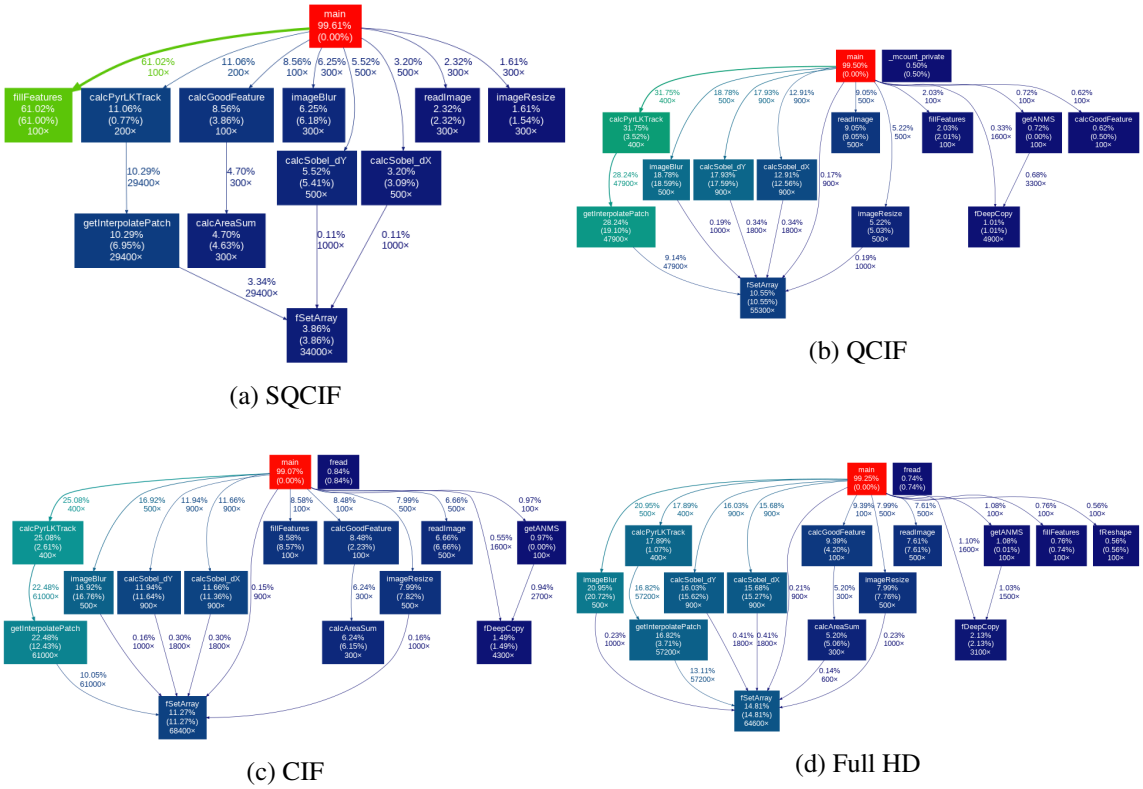


Figure 5.2: Feature Tracking Benchmark Profiling Results from 100 executions

loop that starts in line 104. This takes away the need to initialize a data structure to store the kernel values, *kernel*, i.e., accesses to the kernels in *imageBlur*(), *calcSobel_dY*() and *calcSobel_dX*() can be replaced by the value in each position after completely peeling their innermost loops. Another variable that has a constant value is *halfKernel* which is 2 in *imageBlur*() and 1 in *calcSobel_dY*() and *calcSobel_dX*(), meaning that it can also be replaced by a literal in those instances.

```
1  for(i=startRow; i<endRow; i++) {
2      for(j=startCol; j<endCol; j++)
3      {
4          temp = 0;
5          for(k=-halfKernel; k<=halfKernel; k++)
6          {
7              temp += subsref(imageIn,i,j+k) * asubsref(kernel,k+halfKernel);
8          }
9          subsref(tempOut,i,j) = temp/kernelSum;
10     }
11 }
```

Listing 5.3: First loop used in the Feature Tracking benchmark hotspot function, *imageBlur*()

After applying the aforementioned transformations, it becomes clear that in each of these functions, one of the two nested loops present can be targeted for Data Reuse, as is the case for the loop shown in Figure 5.3. Data Reuse cannot be applied to the other nested loop without applying Loop Permutation first, as the reused array position has a dependency in the outer loop's control variable. A permutation in the second nested loop could be justified, if the subsequent Data Reuse improves the execution time more than a less favourable memory access order hinders it.

Lastly, each hotspot initializes a buffer array with all values equal to zero to help with the calculations, although the non-border ones are overwritten without ever being used - border values being the ones in the first and last *halfKernel* rows and columns. The function that performs this initialization is called *fSetArray*() and it belongs to the common library of functions. In order to prevent breaking other benchmarks that use this function, instead of changing the function itself, it is valid to inline its algorithm within each hotspot.

Overall, this benchmark has a reasonable potential of being improved in terms of execution time, mainly due to the possible removal of the *kernel* array and the low number of registers needed to implement Data Reuse.

5.4 Scale Invariant Feature Transform

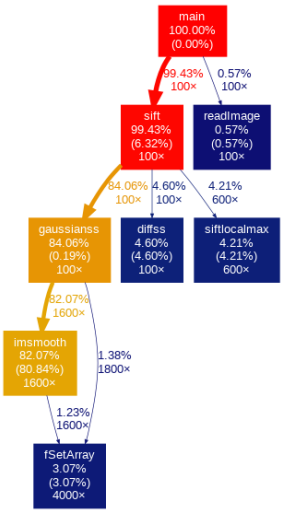
SIFT is used to detect and describe robust characteristics of highly descriptive images. This algorithm is mostly used for navigation and match moving applications.

Analysing the initial profiling results, shown in Figure 5.3, the *imsmooth*() function stands out from the rest. This hotspot consumes over 80% of the total execution time for all image sizes, and specifically 83.41% for Full HD inputs, as seen in Figure 5.3d.

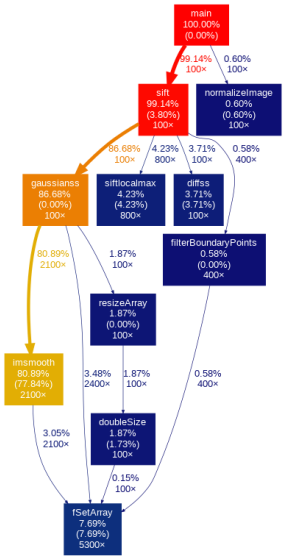
The first aspect we noted in this benchmark was the use of *assert*() functions throughout the hotspot. These functions potentially add a significant overhead, therefore, removing them would be the most logical transformation to implement before anything else.

In case a threshold value is surpassed, the main section of *imsmooth* uses loop nesting to perform two convolutions of the input array pixel values. The first convolution is done over the column elements and the second over the row elements, storing the final results in an output array. In case the threshold value is not surpassed the function simply copies the input array values to the output array. The convolutions are made using a sliding window technique to determine the innermost start and end values, as shown in Listing 5.4.

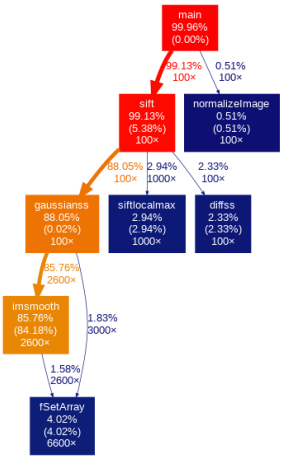
```
1 for(j = 0 ; j < M ; ++j)
2 {
3   for(i = 0 ; i < N ; ++i)
4   {
5     int startCol = MAX(i-W,0);
6     int endCol = MIN(i+W, N-1);
7     int filterStart = MAX(0, W-i);
8     for(k=startCol; k<=endCol; k++)
```



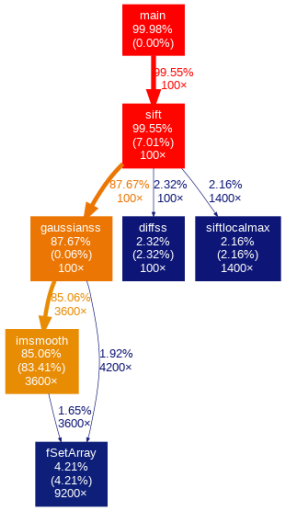
(a) SQCIF



(b) QCIF



(c) CIF



(d) Full HD

Figure 5.3: SIFT Benchmark Profiling Results from 100 executions

```
9           subsref(buffer,j,i) += subsref(array, j, k) * temp[filterStart++];
10        }
11    }
```

Listing 5.4: First convolution loop used in the SIFT benchmark hotspot function, *imsmooth*()

A remarkable characteristic of the convolution loops is that they use macros for conditional expressions to determine the values of *startCol*, *endCol* and *filterStart*. Figure 5.4 illustrates the behaviour of the three conditional expressions, each described by a piecewise function, which reveals that, as expected, all three variables either have a constant value or follow a linear equation. Further analysis of the *startCol*, *endCol* and *filterStart* value progression reveals that

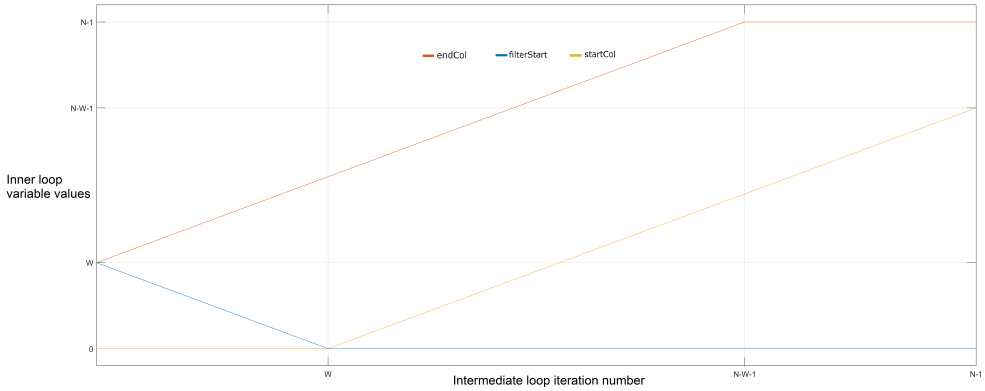


Figure 5.4: Mathematical representation of the values of the *startCol*, *endCol* and *filterStart* variables in relation to the medium loop's control variable value

there are two distinct breakpoints in the three piecewise equations, i.e., $i = W$ for *startCol* and *filterStart* and $i = N - W - 1$ for *endCol*. These breakpoints, in practice, represent the moments at least one of the macro outputs swaps from one member to the other. This provides us the ideal number of times the loop should be split so that the macros become redundant. The transformation results in three loops, where the innermost iteration domains are (1) $startCol = 0$, $endCol = i + W$ and $filterStart = W - i$, (2) $startCol = i - W$, $endCol = i + W$ and $filterStart = 0$ and (3) $startCol = i - W$, $endCol = N - 1$ and $filterStart = 0$. Moreover, to maintain a favourable memory access order after splitting the loop represented by Listing 5.4, a Loop Fission of the outer loop can be applied. All resulting loops can then be normalized so that the number of iterations performed becomes more clear for both the compiler and developers.

A study on the values of the matrix size, MxN, of the images passed to this function and the possible lengths obtained for the sliding window, W, shows that the hotspot is called a total of 36 times, with 35 different combinations of MxN and W. The first important aspect to note is that M is always equal to N. Secondly, the first call of *imsmooth*() has a $N = 2160$ and $W = 5$. Lastly, the 35 subsequent calls are made using unique combinations of 7 values of N and M, which can be 2160, 1080, 540, 270, 135, 64, or 35, and 5 values of W, which can be 5, 7, 8, 10 or 13. Either one, or even both, of these variables can be specialized by creating multiple versions of

imsmooth() and calling the appropriate one each time. A specialization of the matrix sizes helps the compiler detect the number of iterations each outer and intermediate convolution loop will perform, which might lead to some level of parallelization. On the other hand, a specialization of the sliding window size helps the compiler detect the number of iteration each inner loop will perform.

Since it affects the innermost loop, the specialization of W is the most likely to cause an impact in the compilation and, consequently, in the execution time of the benchmark. Having W declared as a literal, also exposes that two of innermost the loops, i.e. the ones iterating over domain (2) mentioned above, have $2 * W$ iterations.

The relatively small values of W facilitates the complete peeling of both loops that iterate over domain (2) without sharply increasing the amount of lines of code. The 2-level nested loop that computes the convolution along the columns of the input image can then be targeted for Data Reuse, as multiple elements of the *temp* and *input* arrays fit the criteria. This transformation could be particularly effective for lower values of W. Conversely, for higher values of W, the number of registers required to implement the recipe might hinder performance, especially for machines with less processing power.

The loop that performs the convolution along the rows through iteration domain (2), on the other hand, has a data dependency on the outer loop's control variable. Applying Loop Permutation would enable Data Reuse, but performance might deteriorate if a less favourable array access order hinders the execution time more than Data Reuse improves it.

5.5 Benchmarks Overview

Our application of the selected performance recipes to the benchmarks explored thus far, resulted in a set of benchmark versions with different respective code transformations.

Table 5.1 displays each benchmark's hotspots and respective characteristics in terms of the number of Lines of Code (LOC), Loops and Arrays in their original versions. Tables 5.2, 5.3 and 5.4 present the version naming structure used to identify the transformations applied in each stage to SIFT, Feature Tracking and Disparity, respectively, as well as the associated changed files, number of LOC, Loops and Arrays. These values expose an increase in LOC upon the application of Multiple Function Versioning, Function Inlining or Data Reuse. Reductions in the number of Loops are a result of Loop Peeling.

The Baseline version of SIFT contains a total of 9 Loops in its hotspot, of which 5 are of rank 1, 2 are of rank 2 and 2 are of rank 3. The increase in number of LOC and Loops in SIFTv1 is a result of creating multiple versions of the hotspot for each sliding window size. After splitting the convolution loops, the benchmark also suffers an increase in the number of loops, despite the complete peeling of two loops mitigating this value. SIFT has no change in the number of arrays after the implementation of Multiple Function Versioning and the increase relative to Baseline is a consequence of the different hotspot function versions. The *gaussianss.c* file contains the

Benchmark	Hotspots	# LOC	# Loops	# Arrays
SIFT	imsmooth	76	9	3
	gaussianss.c	106	6	6
Feature Tracking	calcSobel_dX	77	6	5
	calcSobel_dY	72	6	5
	imageBlur	60	6	4
	script_tracking.c	222	5	20
Disparity	finalSAD	22	2	2

Table 5.1: Benchmark hotspots and respective characteristics

Version	Transformations	File	# LOC	# Loops	# Arrays
SIFTv1	Baseline Specialization Multiple Function Versioning	imsmooth.c	331	45	15
		gaussianss.c	183	6	6
SIFTv2	SIFTv1 Loop Splitting Loop Peeling Loop Normalization	imsmooth.c	705	90	15
		gaussianss.c	183	6	6
SIFTv3	SIFTv2 Data Reuse	imsmooth.c	1076	90	15
		gaussianss.c	183	6	6

Table 5.2: SIFT Benchmark version information.

Version	Transformations	File	# LOC	# Loops	# Arrays
TRACKv1	Baseline Specialization Loop Peeling Data Reuse	calcSobel_dX.c	56	4	3
		calcSobel_dY.c	59	4	3
		imageBlur.c	61	4	3
		script_tracking.c	222	5	20
TRACKv2	TRACKv1 Function Inlining Specialization Multiple Function Versioning	calcSobel_dX.c	133	12	6
		calcSobel_dY.c	130	12	6
		imageBlur.c	82	6	3
		script_tracking.c	227	5	20

Table 5.3: Feature Tracking Benchmark version information.

Version	Transformations	File	# LOC	# Loops	# Arrays
DISPv1	Baseline Specialization Loop Peeling Data Reuse	finalSAD.c	79	2	2

Table 5.4: Disparity Benchmark version information.

imsmooth() function's caller *gaussianss*(), which is why its number of LOC increases when after applying Multiple Function Versioning.

Disparity contains a single hotspot with an initial 22 LOC, 2 Loops and 2 Arrays. Data Reuse and Loop Peeling increase the number of LOC to 79 in DISPv1, while the other metrics remain unchanged.

5.6 Automated Transformations

In our efforts to use the automated transformations described in Chapter 4, we first created an interface within Clava to load, compile and execute the SDVBS. The interface consists of a SDVBSBenchmarkSet class, inherited from lara.benchmark.BenchmarkSet and a SDVBSBenchmarkInstance, inherited from lara.benchmark.ClavaBenchmarkInstance [51].

Our first goal was to implement the fully automated recipes to SDVBS, i.e., *replaceDoublesWithFloats* and *replacePowCalls*. As shown in Tables 5.5 and 5.6, SIFT was the benchmark that took the most advantage out of the scripts, whereas Feature Tracking took the least. The values presented were obtained by inserting counters around each transformed code section and logging the results in the end using the Clava ClavaJoinPoints and LARA Logger interfaces described in Chapter 3

The discrepancy between the number of times applied and number of times executed of *replaceDoublesWithFloats*() observed in Disparity and Feature Tracking is due to the existence of common code throughout all benchmarks, as explained in Section 5.1, which is always compiled and subsequently transformed by Clava, regardless if it is executed or not. Results, therefore, suggest that this transformation will almost uniquely favour SIFT, which has 8 *double* literals in its benchmark specific library and the 16 common library *double* literals replaced by the LARA script. These 24 transformed literals are subsequently executed 1343 times, which is a reasonable number, but it might not be enough to significantly improve performance. When it comes to *replacePowCalls*(), SIFT was the only benefited benchmark, having 5 calls replaced with multiplication, which are executed 15 times throughout the algorithm. The other benchmarks had no calls to this function.

Secondly, we executed both *loopSplitting* and *loopSplittingFission*() automated scripts to the convolution loops in the SIFT hotspot. Both scripts require user input through Clava *pragmas* in the source code to determine the iteration domains of each resulting loop. Figure 5.5 shows the object used to maximize the efficiency of splitting the row convolution nested loop in *imsmooth*(). Both strategies can be applied to any loops that follow an appropriate *pragma* and there are two fitting loops for the transformations, which can be split into three loops each.

5.7 Summary

Before effectively measuring the performance of each version of the benchmarks, we analysed their code and determined potential areas of improvement. To each benchmark we assigned a set of performance recipes from our portfolio that could be applied and estimated the impact each one

Benchmark	# of times applied	# of executions
SIFT	24	1343
Disparity	18	2
Feature Tracking	16	0

Table 5.5: Number of times the replaceDoublesWithFloats() function was applied and executed in each benchmark

Benchmark	# of times applied	# of executions
SIFT	5	15
Disparity	0	0
Feature Tracking	0	0

Table 5.6: Number of times the replacePowCalls() function was applied and executed in each benchmark

```
1    #pragma clava data intervals: [\
2        {\
3          startValue: "0",\
4          endValue: "W",\
5        },\
6        {\
7          startValue: "W",\
8          endValue: "M-W-1",\
9        },\
10       {\
11         startValue: "M-W-1",\
12         endValue: "M",\
13       },\
14     ]
```

Figure 5.5: Pragma used to split the row convolution loop in *imsmooth*

would have on the execution time. All hotspots found involved looping over one or more arrays of pixels from images passed to each function, storing the results in an output array.

Disparity reveals a single hotspot with a considerable percentage of the total execution time. The images passed to the function have constant size throughout the benchmark and the loop that iterates over the input reuses two array elements in each iteration, 8 iterations after those values are used for the first time.

Feature Tracking presents, rather than a single hotspot function, a group of similarly structured functions which consumes the majority of the execution time. These functions involve cyclical operations between the input array and specific kernels, making them suitable for the application multiple performance recipes such as Loop Peeling and Specialization. This benchmark's hotspot functions are a part of the common library of functions used across multiple benchmarks in the suite, expanding the value of any improvements achieved to other benchmarks in SDVBS.

SIFT contains a single hotspot consuming a significant portion of the execution time. The hotspot performs two convolutions with the pixel values in the input images, which have predictable dimensions, and each convolution is computed using sliding windows of predictable lengths. The benchmark has a large potential for improvement, due to the high percentage of time consumed by the hotspot function and the various transformations that can be applied to it.

Overall, the benchmarks chosen for our analysis show a considerable potential for improvement and provide a useful variability in their structures and modes of operation, within the field of computer vision.

Chapter 6

Experimental Results

In this chapter we describe our experimental procedure and results of the work described previously.

Section 6.1 describes the experimental setup used in this benchmark analysis. Sections 6.2 and 6.3 analyse the results obtained in ANTAREX and ODROID, respectively. Metrics obtained include execution times, cache behaviour, vectorization and energy consumption. Section 6.4 described the efforts made in creating automated transformation scripts and the subsequent results obtained.

6.1 Experimental Setup

Each benchmark version was run on two machines, a HPC named ANTAREX and an embedded system called ODROID. Section 6.2 presents the results obtained in the ANTAREX and Section 6.3 explores the results obtained using ODROID. A more in-depth description of each of these systems, including hardware specification, is provided in Chapter 3.

After applying the mentioned transformations to each benchmark, we extracted different metrics divided into three distinct categories.

The first is the execution time of each benchmarks version. This metric is the most relevant one, as it directly reflects the effectiveness of the applied performance recipes, and through it we can calculate the speedup obtained after implementing each transformation relative to the Baseline versions. The execution times we present are averages obtained from 30 executions of a given version of a benchmark [29]. This was done to remove the effects of standard deviations. The different results shown also correspond to different optimization flags provided at compilation.

We use GCC [24], version 7.5.0 in the ANTAREX machine and 6.5.0 in the ODROID embedded system, using its optimization flags $-o0$, $-o2$ and $-o3$ applied to all benchmarks, whereas, $-ofast$ is only applied to SIFT and Feature Tracking and is not used in our study of Disparity because it does not maintain the code integrity, i.e., introduces errors in the algorithm.

The second metric category is the vectorization log provided by compiling each version using the $-ftree-vectorizer-verbose=5-fopt-info$. This lets us search for instances of vectorization and other loop transformations being applied to the profiled loops. Changes in these logs from one version to another can justify performance improvements and clarify if our transformations aided in the parallelization of the studied loops.

We perform a cache profiling in ANTAREX to understand how the code transformations affect the cache behaviour of each benchmark. This profiling provides us with useful metrics such as the total number of instructions executed, the number of memory access instructions performed and the cache miss rate. ODROID was not subject to a cache profiling due to a recurring software lock when extracting the metrics from the benchmarks.

Lastly, we perform an energy and power profiling using the ODROID energy monitoring library. The goal is to understand the impact of the selected transformations in these metrics, taken from the CPU and from the memory units.

6.2 ANTAREX Machine Results

This section shows the results obtained from executing the three studied benchmarks using the ANTAREX HPC machine. Using this machine, we performed a global execution time profiling of each benchmark and analysed the vectorization log and cache behaviour of their hotspot functions. For the SIFT benchmark, we also profile the execution times of its hotspot functions and correlate the results to the sliding window size used in the convolution algorithm.

Scale Invariant Feature Transform (SIFT)

Figure 6.1 shows the execution time of each SIFT version using each optimization flag. Comparing the performance of each version for a given GCC optimization level (e.g., O0, O2, O3 and Ofast), we can see that SIFTv1 approximately maintains performance, as execution times for this version are never more than 2% lower than the Baseline version. This suggests that simply specializing the sliding window size does not favor the execution time. The vectorization log further justifies this by reporting that no vectorization transformation was applied by the compiler.

SIFTv2, on the other hand, shows an increase in performance for all optimization flags. The execution times obtained for $-o0$ correspond to a speedup of 1.11, as shown in Table 6.1, indicating that the loop transformations applied in this version can slightly improve performance. Using more powerful optimization flags, the speedup increases to 1.39 using $-o2$, 1.79 using $-o3$ and 1.80 using $-ofast$, which suggests that, in addition to directly reducing execution times, the recipes used in SIFTv2 help the compiler improve performance even more. Although, the vectorization log reveals that no new loops are vectorized in this version compared to the Baseline, therefore, these results are caused by a different type of compiler transformation.

The third version of the benchmark, SIFTv3, shows a 1.70 speedup compared to the Baseline when using $-o0$, which indicates that applying Data Reuse can reduce execution times. Using the $-o2$ flag, SIFTv3 has a speedup of 1.58 compared to the Baseline, which is higher than SIFTv2's.

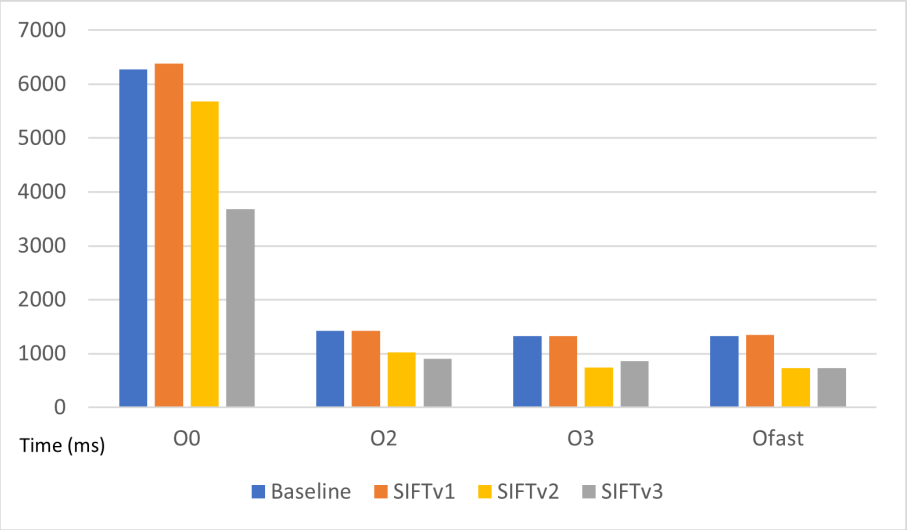


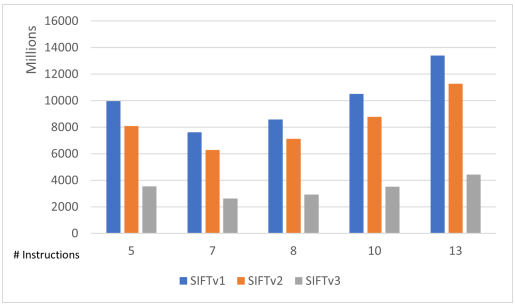
Figure 6.1: Execution times of SIFT sliding window size hotspots in Baseline, SIFTv1, SIFTv2 and SIFTv3 using ANTAREX

The vectorization log for this version shows that one less loop is vectorized using $-o3$, namely the innermost loop in the third row convolution loop nest. This could be the reason for the performance deterioration when compared to SIFTv2 using this flag. Using $-ofast$, on the other hand, one new loop is vectorized in SIFTv3, more specifically, the innermost loop of the third column convolution loop nest, equaling the speedup of 1.80 in SIFTv2. These results indicate that there is a slight correlation between the number of parallelized loops and the performance improvement of the benchmark, but there are logically, more variables at hand, otherwise, using $-ofast$, SIFTv3 would presumably have a lower execution time than SIFTv2.

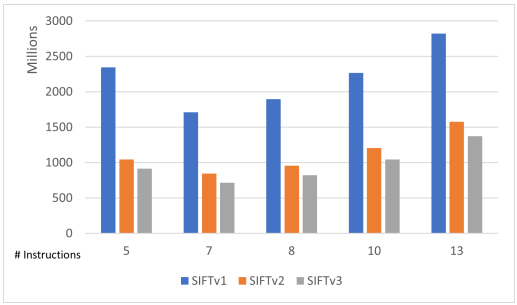
The cache profiling we performed on this benchmark was made with the intent of understanding how the number of instructions and memory accesses evolves as the sliding window size increases. The first aspect we verified was that both the Baseline and SIFTv1 have an identical behaviour in all aspects and for all optimization flags. This means it is valid to use SIFTv1 as a reference point. This association of the Baseline with SIFTv1 was essential for this analysis, as the used profiler, Cachegrind, appears to add a significant overhead when instrumenting specific functions in the Baseline, which was to only other way of obtaining data relative to each sliding window size. Figure 6.2 presents the total number of instructions executed by each version for a

Speedups	**SIFTv1**	**SIFTv2**	**SIFTv3**
-o0	0.98	1.11	1.70
-o2	1.00	1.39	1.58
-o3	1.00	1.79	1.54
-ofast	0.98	1.80	1.80

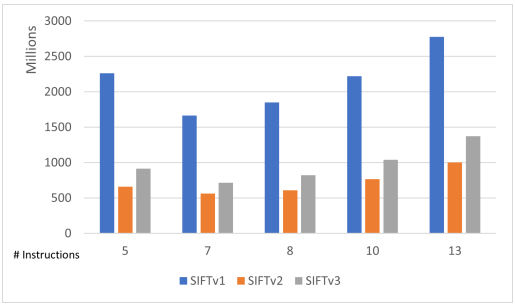
Table 6.1: Speedups of each SIFT benchmark version with performance flags compared to Baseline using ANTAREX



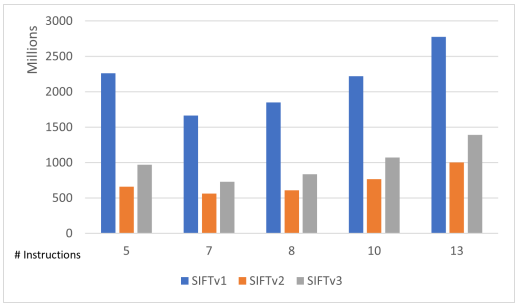
(a) Using - O0 optimization flag



(b) Using - O2 optimization flag

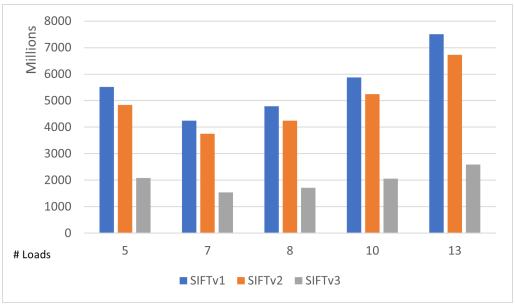


(c) Using - O3 optimization flag

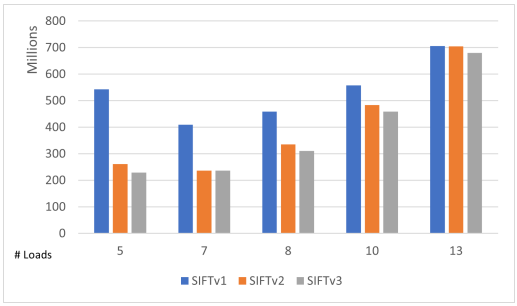


(d) Using - Ofast optimization flag

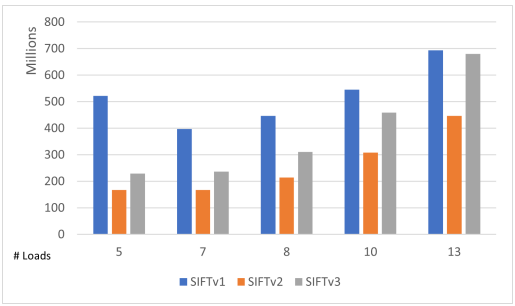
Figure 6.2: Number of instructions executed by SIFTv1, SIFTv2 and SIFTv3 relative to sliding window size using ANTAREX



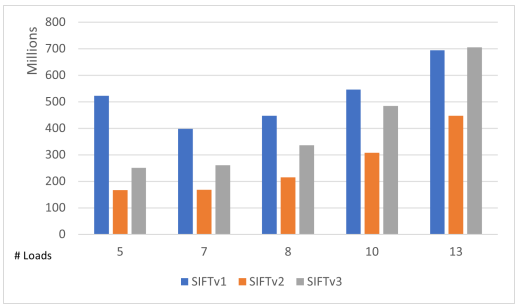
(a) Using - O0 optimization flag



(b) Using - O2 optimization flag

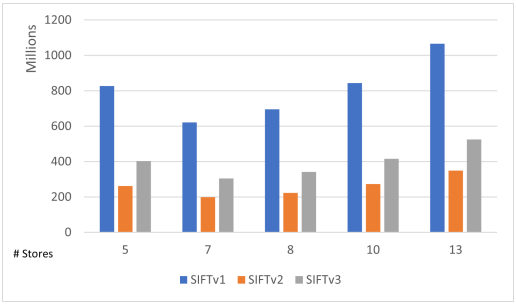


(c) Using - O3 optimization flag

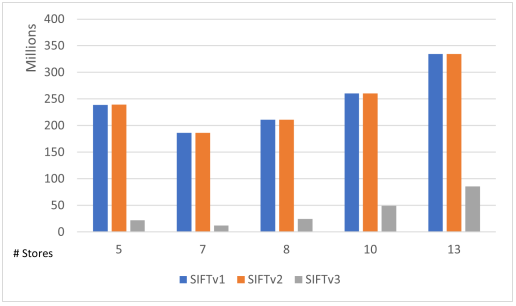


(d) Using - Ofast optimization flag

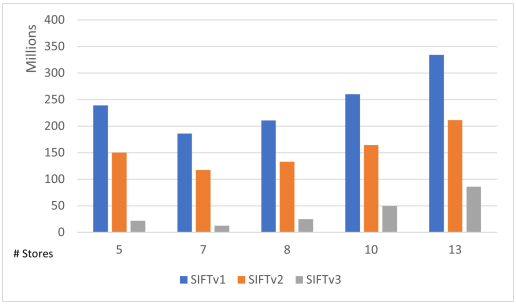
Figure 6.3: Number of load instructions executed by SIFTv1, SIFTv2 and SIFTv3 relative to sliding window size



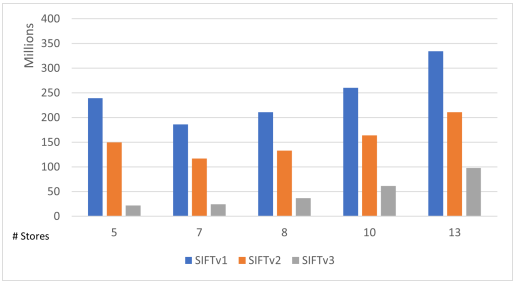
(a) Using - O0 optimization flag



(b) Using - O2 optimization flag



(c) Using - O3 optimization flag



(d) Using - Ofast optimization flag

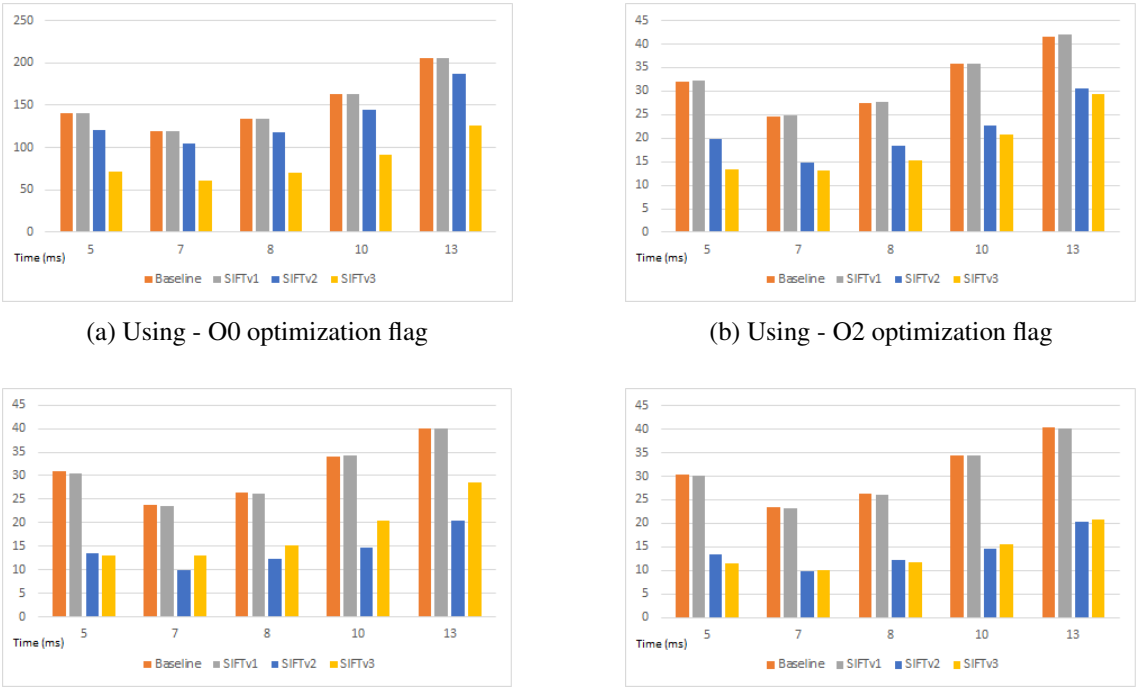
Figure 6.4: Number of store instructions executed by SIFTv1, SIFTv2 and SIFTv3 relative to sliding window size using ANTAREX

given flag. The first aspect that becomes clear is that the recipes used in SIFTv2 reduce this number regardless of the used optimization flag. Also, the reduction in this value is more significant the more powerful the optimization flag is, similarly to the speedup behaviour mentioned above. Conversely, in SIFTv3, this metric is the highest for $-o0$, but fairly similar for $-o2$, $-o3$ and $-ofast$. When using the $-o0$ and $-o2$ flags, reduction in the number of instructions performed in each sliding window function of SIFTv2 and SIFTv3 compared to SIFTv1 indicates that, in these conditions, the execution time of this benchmark is correlated to the reduction of the number of instructions performed. Using $-o3$ and $-ofast$, the number of instructions in SIFTv3 becomes higher than SIFTv2, but still significantly lower than in SIFTv1.

The number of load and store instructions performed, respectively shown in Figures 6.3 and 6.4, firstly demonstrate that the recipes applied in both SIFTv2 and SIFTv3 result in a reduction of both these metrics compared to SIFTv1. Moreover, these values indicate that Data Reuse is much more efficient in reducing the number of loads from memory than the transformations applied in SIFTv2 on a source-source only scope, as seen in Figure 6.3a. Only if the compiler applies its own optimizations, i.e., using $-o2$, $-o3$ or $-ofast$, does Data Reuse become less impacting than the techniques implemented in SIFTv2, as seen in Figures 6.3b, 6.3c and 6.3d. Also, the impact of Data Reuse in this metric is, as expected, significantly reduced for larger sliding window sizes. Contrarily, the number of stores to memory is reduced the most for flags that apply compiler transformations, as evidenced in Figures 6.4b, 6.4c and 6.4d.

Since we converted the SIFT hotspot function into multiple versions specialized by each W,

we are able to see how our performance recipes affect the benchmark relative to the value of W. To perform this measurement, we instrumented each version of *imsmooth*() in terms of execution time, which results are represented by the chart in Figure 6.5.



(a) Using - O0 optimization flag

(b) Using - O2 optimization flag

(c) Using - O3 optimization flag

(d) Using - Ofast optimization flag

Figure 6.5: Execution times of the hotspot function relative to W in each version of the SIFT benchmark using ANTAREX

The first aspect we can note is that the general impact in performance of each recipe is fairly similar, regardless of the optimization flag used, i.e., SIFTv1 has a very close performance compared to Baseline, whereas SIFTv2 and SIFTv3 improve the hotspot execution time significantly. That being said, the only scenarios where SIFTv3 shows improvements relative to SIFTv2 for all values of W is when the flags $-o0$ and $-o2$ are used.

The lower effectiveness of SIFTv3 is accentuated in the presence of more powerful compiler flags, suggesting that compiler optimizations, can achieve better performance results without the application of Data Reuse. Furthermore, when the $-o3$ flag is used, the performance of SIFTv3 deteriorates compared to SIFTv2, namely for values of W larger than 5. This suggests that Data Reuse becomes detrimental for larger amounts of reusable array elements. It is important to note that the hotspot function is executed with $W = 5$ more often than with other W values, as we explain in Chapter 5, and this is the only value of W in which the performance of SIFTv3 is consistently better than or equal to that of SIFTv2. We, therefore, believe that the most favourable combination of transformations would be to apply the transformations that correspond to SIFTv2, while only applying Data Reuse to the specialized hotspot function for $W = 5$, in cases where further compiler optimizations are applied. Otherwise, SIFTv3 appears to already include the most efficient combination of recipes.

Speedups	TRACKv1	TRACKv2
-o0	1.37	1.52
-o2	1.27	1.32
-o3	0.97	0.99
-ofast	0.98	0.99

Table 6.2: Speedups of each Feature Tracking benchmark version with performance flags compared to Baseline using ANTAREX

Feature Tracking

The execution time profiling run on the Feature Tracking benchmark suggests a decrease in efficiency of our transformations as more powerful optimization flags are used, as shown in Figure 6.6 and Table 6.2. The vectorization analysis explains this behaviour, as it shows that using the flags $-o3$ and $-ofast$, the Baseline achieves vectorization on both kernel loops in each hotspot of the benchmark. On the other hand, TRACKv1 misses the parallelization of one loop in each hotspot using either flag. TRACKv2 achieves even less vectorization, as using $-o3$, only one of the kernel loops in *calcSobel_dY* is vectorized. Using $-ofast$ one kernel loop in each hotspot is vectorized in TRACKv2 as well as one of the inlined $fSetArray()$ loops. The $-o0$ and $-o2$ flags, on the other hand, do not activate such optimizations, furthering validating this claim, with speedups of 1.37 and 1.27 in TRACKv1 and 1.52 and 1.32 in TRACKv2.

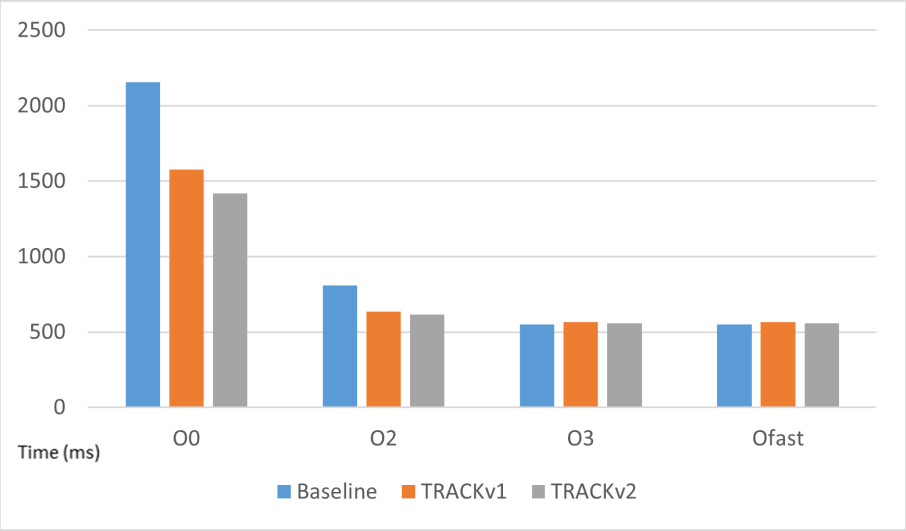
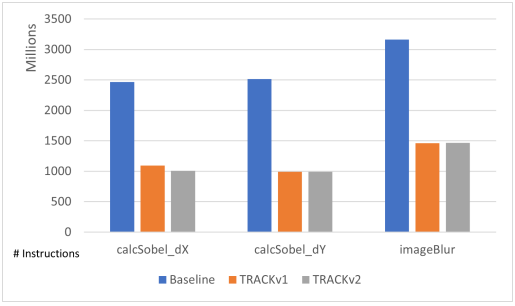
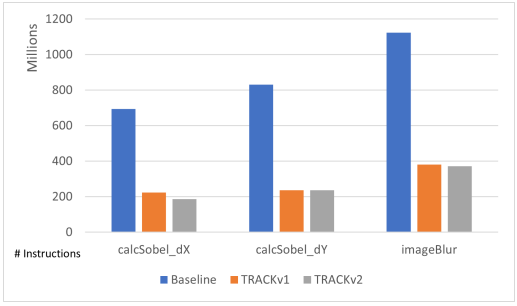


Figure 6.6: Execution times of Feature Tracking hotspots in Baseline, TRACKv1 and TRACKv2 using ANTAREX

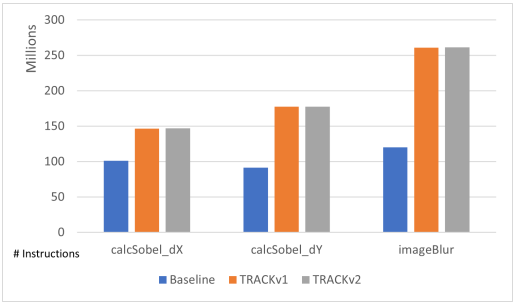
For this benchmark, the cache profiling focused on the three hotspot functions *calcSobel_dX*(), *calcSobel_dY*() and *imageBlur*(). The variable specialization implemented in TRACKv2 resulted in multiple versions of each of these functions, therefore values presented results, shown in Figure 6.7, reveals that, using the $-o0$ and $-o2$ flags, the hotspot functions of both TRACKv1 and TRACKv2 have an over 50% reduction of the number of instructions. This suggests once more



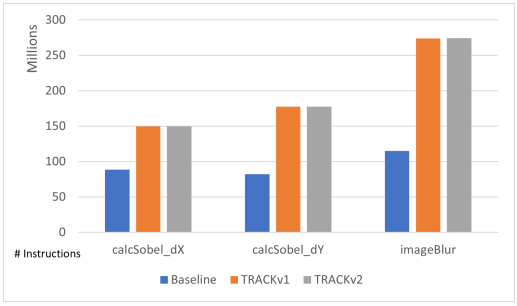
(a) Using - O0 optimization flag



(b) Using - O2 optimization flag



(c) Using - O3 optimization flag



(d) Using - Ofast optimization flag

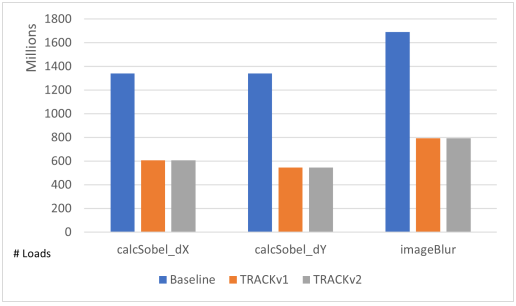
Figure 6.7: Number of instructions executed by Feature Tracking hotspots in Baseline, TRACKv1 and TRACKv2 using ANTAREX

that Data Reuse improves performance by reducing the amount of instructions executed when the compiler implements no vectorization focused transformations. This situation is not verified when passing $-o3$ or $-ofast$ to the compiler, as the number of instructions of both transformed versions show a significant increase with the cache miss rate lower than 1% for all three versions. The $-o3$ and $-ofast$ flags, increase this value to approximately 1.7%, 2.0% and 2.1%, respectively, which is still negligible.

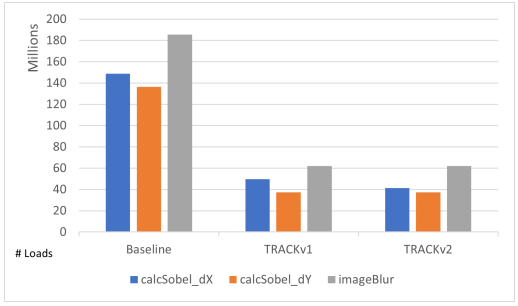
The analysis on the number of memory access instructions in this benchmark reveals that similarly to the number of instructions, less compiler action, i.e., using the $-o0$ and $-o2$ flags, results in a reduction of the number of load and store instructions when comparing both refactored versions of Feature Tracking with the Baseline, as shown in Figures 6.8 and 6.9. This behaviour further indicates that an increase in the number of instructions performed by an algorithm, does not necessarily cause a worse performance.

Disparity

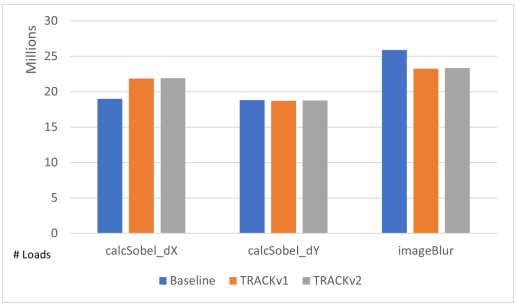
Our performance analysis of Disparity in terms of execution time, shown in Figure 6.10 and Table 6.3, presents a significant performance deterioration in the absence of compiler optimizations and a speedup close to 1 using the $-o2$ and $-o3$ flags. This behaviour was expected in this benchmark, due to the large number of registers needed to implement Data Reuse. The vectorization log corroborates this, as no vectorization is applied to the hotspot function in either version.



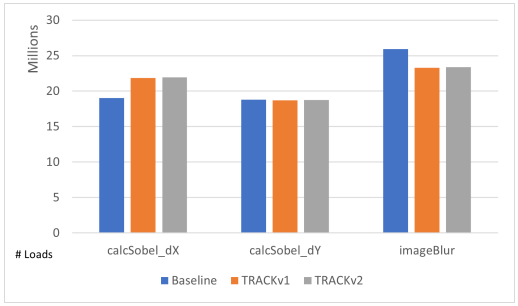
(a) Using - O0 optimization flag



(b) Using - O2 optimization flag



(c) Using - O3 optimization flag



(d) Using - Ofast optimization flag

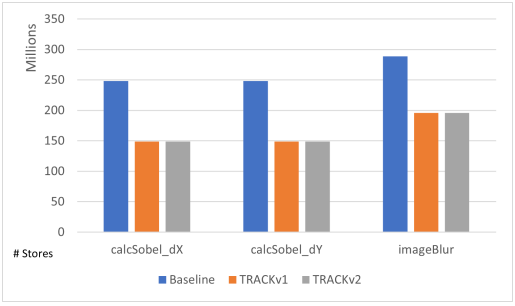
Figure 6.8: Number of load instructions executed by Feature Tracking hotspots in Baseline, TRACKv1 and TRACKv2 using ANTAREX

The cache profiling results, shown in Figure 6.11, further justify the lack of performance improvements, with an increase in the number of instructions executed in DISPv1 relative to Baseline, represented in 6.11a, for the three performance flags used. The benchmark had a similar cache behaviour using $-o2$ and $-o3$, which is not surprising given the lack of loop vectorization achieved in the hotspot function. The cache miss rates for Baseline and DISPv1 are 2.0% and 1.7%, respectively, using $-o0$. Using $-o2$, on the other hand, raises these values to 11.1% and 11.9% and, using $-o3$, the rates achieve 12.9% and 14.1%.

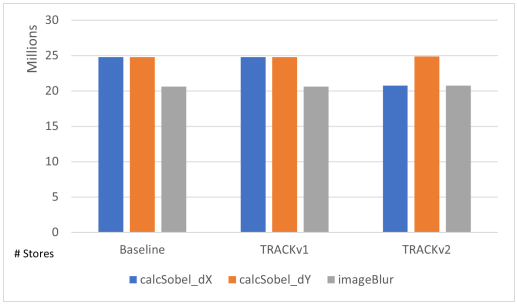
The number of memory access instructions executed using $-o0$, presented in Figures 6.11b and 6.11c, show an enormous increase in the amount of store instructions in DISPv1 compared to Baseline. This is very likely to be the cause of the performance deterioration observed in terms of execution time. Conversely, using $-o2$ and $-o3$, both versions of the benchmark had similar cache behaviour, with DISPv1 having a more instructions executed in $finalSAD()$, approximately half the memory reads, and similar memory writes relative to Baseline. The decrease in the number of

Speedups	DISPv1
-o0	0.86
-o2	1.04
-o3	0.99

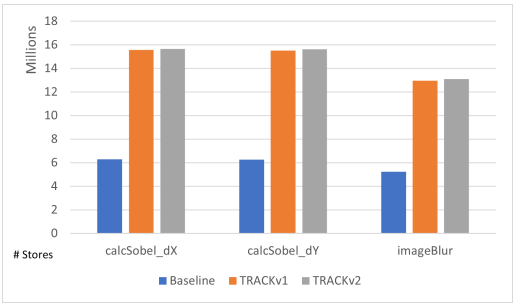
Table 6.3: Speedups of the refactored Disparity benchmark version with performance flags compared to Baseline using ANTAREX



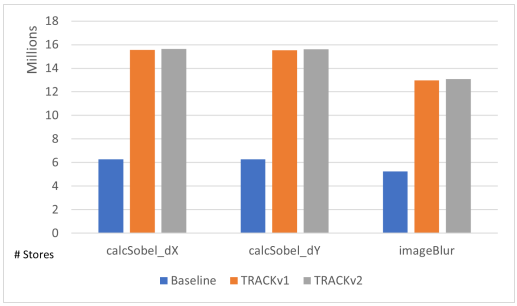
(a) Using - O0 optimization flag



(b) Using - O2 optimization flag



(c) Using - O3 optimization flag



(d) Using - Ofast optimization flag

Figure 6.9: Number of store instructions executed by Feature Tracking hotspots in Baseline, TRACKv1 and TRACKv2 using ANTAREX

load instructions and increase in total instructions appear to have balanced each other in terms of execution times.

Overall, the data suggests a substantial misuse of the cache and an untapped potential for vectorization in the hotspot function, especially for the more powerful optimization flags. Unlike in previous benchmarks, the application of Data Reuse did not, by itself, improve performance in terms of execution time. This is not exactly unexpected, because the array elements used in each iteration are eight positions apart in memory in both versions. Moreover, the Data Reuse applied in this function required the utilization of a large amount of registers, as each 2 array values are

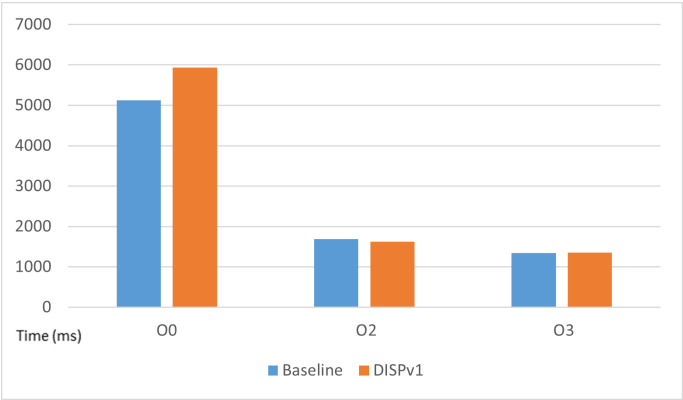
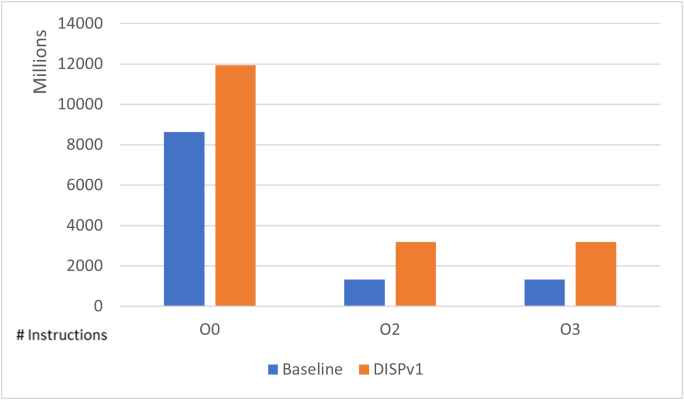
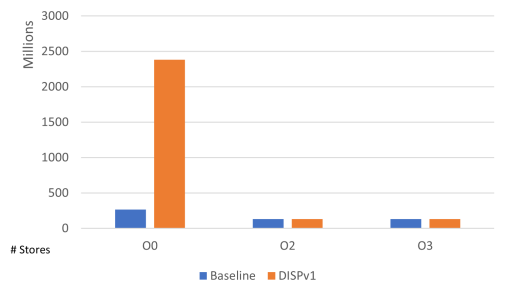


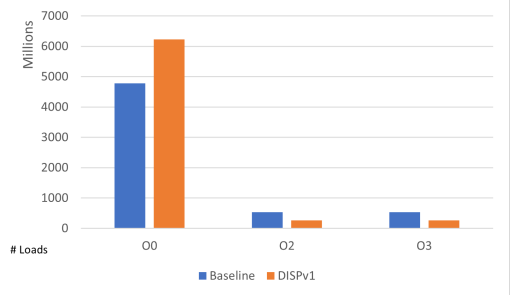
Figure 6.10: Execution times of Disparity hotspots in Baseline and DISPv2 using ANTAREX



(a) Number of instructions executed by *finalSAD* in Baseline and DISPv1



(b) Number of load instructions executed by *finalSAD* in Baseline and DISPv1



(c) Number of store instructions executed by *finalSAD* in Baseline and DISPv1

Figure 6.11: Cache profiling results of *finalSAD*() in Baseline and DISPv1 using the $-o0$, $-o2$, $-o3$ flags

only reused after 7 iterations. Logically, these array elements need to be passed between registers every iteration until they are reused.

6.3 ODROID Results

Being an embedded system, ODROID poses as an interesting subject performance measurement in our benchmarks. The cache profiling done on this system was limited by the computing power available, as some of the benchmark versions lead to software locks when extracting the cache metrics. Therefore, results obtained in relation to this metric have been excluded from our data, as we consider them unreliable. In this section, we profile the global execution times of each benchmark as well as the energy consumption of SIFT.

Scale Invariant Feature Transform (SIFT)

The results obtained for the SIFT benchmark are presented in Figure 6.12 and Table 6.4. The worse performances correspond, as expected, to $-o0$ and versions with the least transformations, i.e., Baseline and SIFTv1. On the other hand, the loop transformations applied in SIFTv2 caused a slight speedup of 1.19. Lastly, SIFTv3, has a 1.85 speedup. This suggests that embedded systems can take advantage of Loop Peeling, Loop Splitting and Loop Normalization especially when no further optimizations are applied by the compiler. Moreover, Data Reuse is the most favourable transformation for this system, despite its not very powerful specifications. The results for the more powerful flags was identical, and this is justified by the vectorization log, which reports that none of the loops are vectorized in any version using any flag. Baseline, SIFTv1 and SIFTv2 have identical execution times, whereas SIFTv3 has a speedup of 1.25.

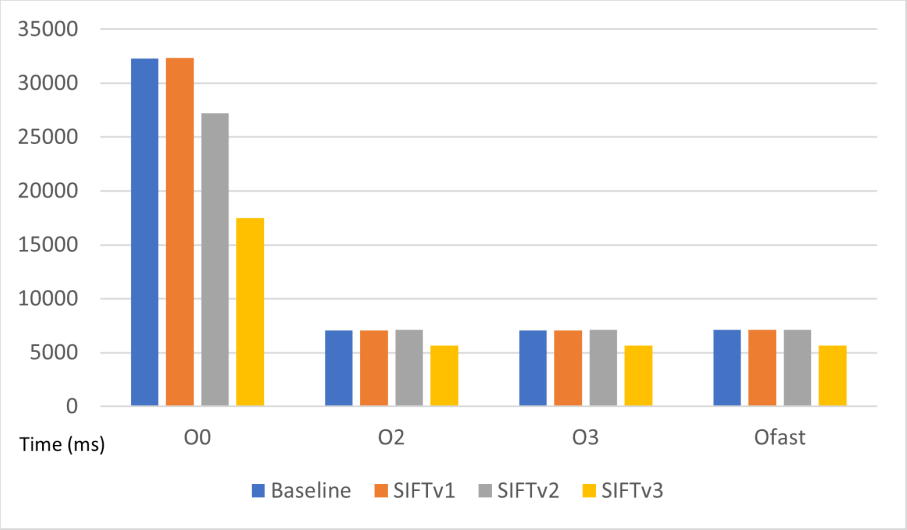
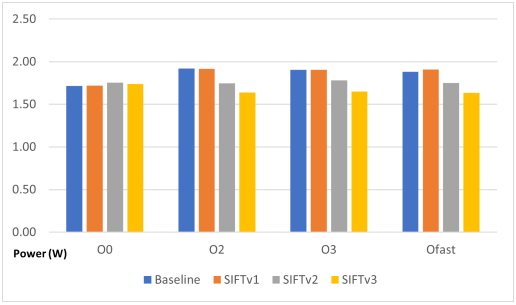


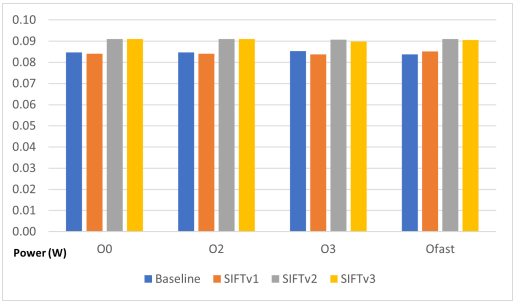
Figure 6.12: Execution times of SIFT sliding window size hotspots in Baseline, SIFTv1, SIFTv2 and SIFTv3 using ODROID

Speedups	SIFTv1	SIFTv2	SIFTv3
-o0	1.00	1.19	1.85
-o2	1.00	0.99	1.25
-o3	1.00	1.00	1.25
-ofast	1.00	1.00	1.25

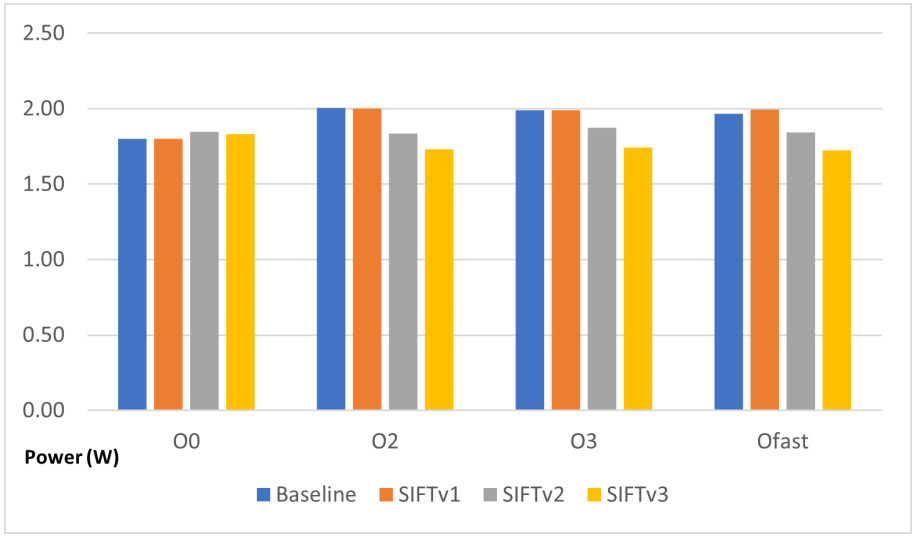
Table 6.4: Speedups of each SIFT benchmark version with performance flags compared to Baseline using ODROID



(a) Power consumed by the CPU executing SIFT benchmark versions

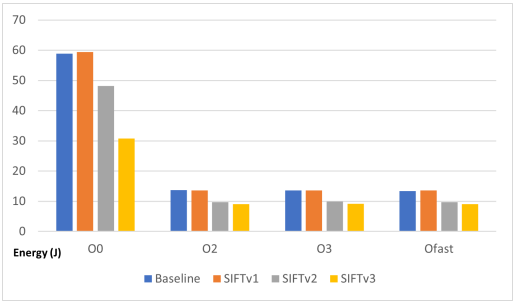


(b) Power consumed by the memory unit executing SIFT benchmark versions

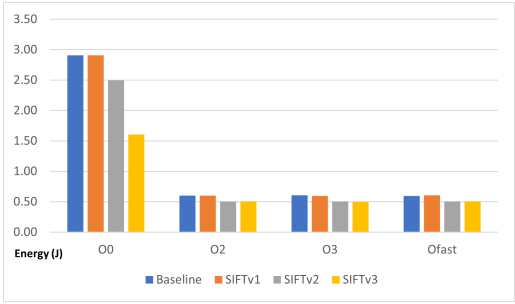


(c) Total power consumed executing SIFT benchmark versions

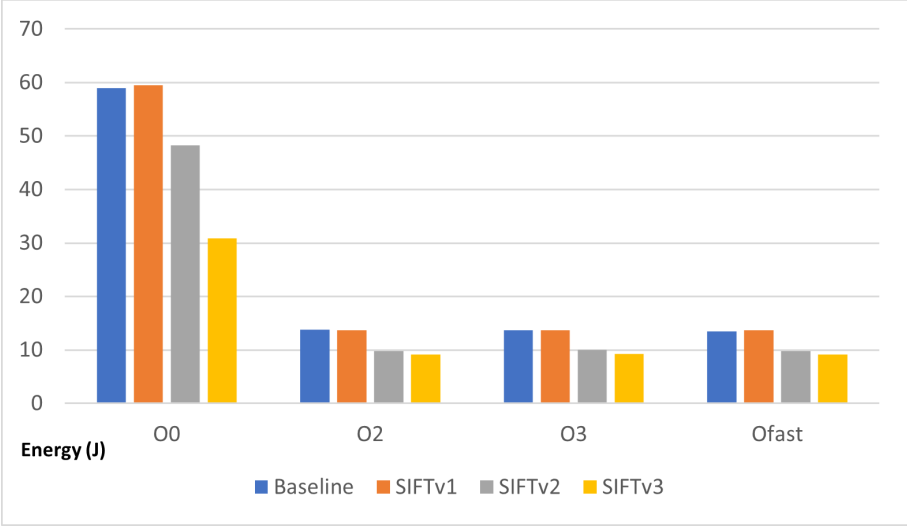
Figure 6.13: Power consumption of each SIFT benchmark version with performance flags using ODROID



(a) Energy consumed by the CPU executing SIFT benchmark versions



(b) Energy consumed by the memory unit executing SIFT benchmark versions



(c) Total energy consumed executing SIFT benchmark versions

Figure 6.14: Energy consumption of each SIFT benchmark version with performance flags using ODROID

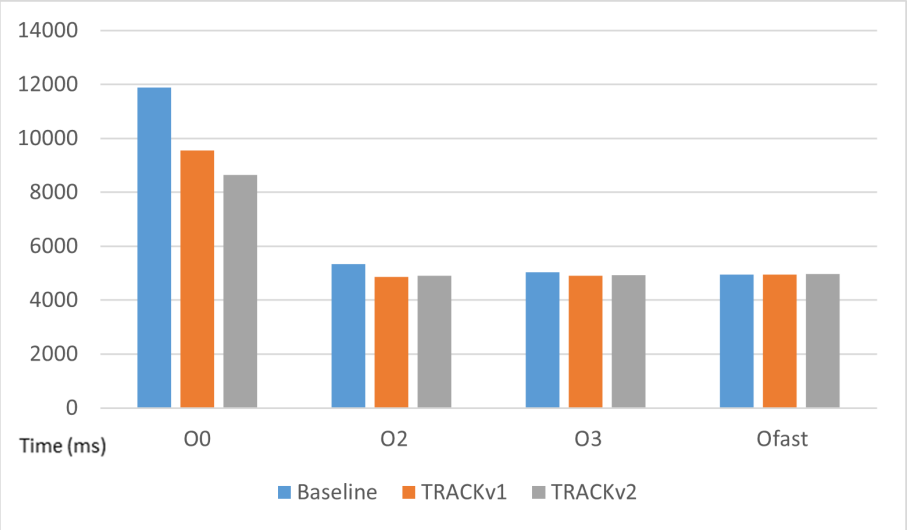


Figure 6.15: Execution times of Feature Tracking hotspots in Baseline, TRACKv1 and TRACKv2 using ODROID

The energy profiling of SIFT reveals slight increases in the power consumed by the CPU and memory components as more transformations are applied and, in cases where execution times are improved, the energy consumed by the whole algorithm can reduce significantly, as shown in Figures 6.13 and 6.14. The power and energy consumption of SIFTv1 is practically identical to Baseline, regardless of the optimization flag used. The power consumption results show that SIFTv2 and SIFTv3, using $-o0$, increase the power consumed by the memory unit in particular, as represented by the chart in Figure 6.13b. This increase is likely a symptom of a higher number of memory accesses, which makes us think this metric has a similar behaviour in ODROID to the one seen in ANTAREX. On the other hand, it has little impact in the total power consumed by the benchmark, as the power consumed by the CPU is much higher and remains approximately the same, culminating in a 1.91 energy consumption improvement overall, as shown in Figures 6.13c and 6.14c. Using $-o2$, $-o3$ and $-ofast$ when executing SIFTv2, the power consumed by the CPU improves by a 15-17% and power consumed by the memory unit deteriorates by a factor of 0.93-0.95, as occurred when using $-o0$. The energy consumed by this version using these flags is reduced by 47-50%, mostly due to the a similar reduction in the energy consumed by the CPU. The energy consumed by the memory unit reduces by 19-22%, which due to its smaller order of magnitude, has less impact overall.

Feature Tracking

The Feature Tracking execution time profiling results, represented in Figure 6.15 and in Table 6.5, further corroborate that the transformations applied improve performance in the absence of compiler optimizations. Using $-o2$, the performance across versions becomes more similar, as TRACKv1 and TRACKv2 generate an improvement of 9-10%. Using $-o3$ and $-ofast$ flags levels the performance even more, as the improvements seen in the transformed versions are of less than

Speedups	TRACKv1	TRACKv2
-o0	1.24	1.37
-o2	1.10	1.09
-o3	1.03	1.02
-ofast	1.00	0.99

Table 6.5: Speedups of each Feature Tracking benchmark version with performance flags compared to Baseline using ODROID

3%, which can be considered negligible. The vectorization logs explain this behaviour, as no loop vectorization is implemented in any combination of flags and versions. This behaviour further suggests that the ARM processing architecture in this system is not able to take full advantage of the transformations and apply vectorization.

Disparity

The Disparity benchmark's results, shown in Figure 6.16 and Table 6.6, present a performance deterioration in the absence of subsequent compiler optimizations. This behaviour can be explained by the large number of registers needed to perform Data Reuse, which is the main aspect that we expected would cause the low specifications of ODROID to yield, as described in Section 6.2. The performance using $-o2$ is identical for Baseline and DISPv1. Using $-o3$ improves upon $-o2$, but both versions still perform equally. Similarly to the other benchmarks, the vectorization log reports no instances of parallelization. This suggests that the the optimization flags can balance performance whether the implemented recipes improve or hinder it, given that in other benchmarks the speedups obtained using $-o0$ were also balanced when switching on compiler transformations.

6.4 Automated Transformations

After running the automated scripts explored in previous chapters on the studied benchmarks we executed the resulting code to understand if they had a relevant performance impact.

All transformations cause no improvement in terms of execution time, with all four having speedups of 1.00 relative to Baseline. This outcome is not unexpected, as the scripts involve transformations that change few aspects of an application or are used to enable other transformations. The maintained performance resulting from script $replaceDoublesWithFloats()$ suggests

Speedups	DISPv1
-o0	0.88
-o2	1.00
-o3	1.00

Table 6.6: Speedups of the refactored Disparity benchmark version with performance flags compared to Baseline using ODROID

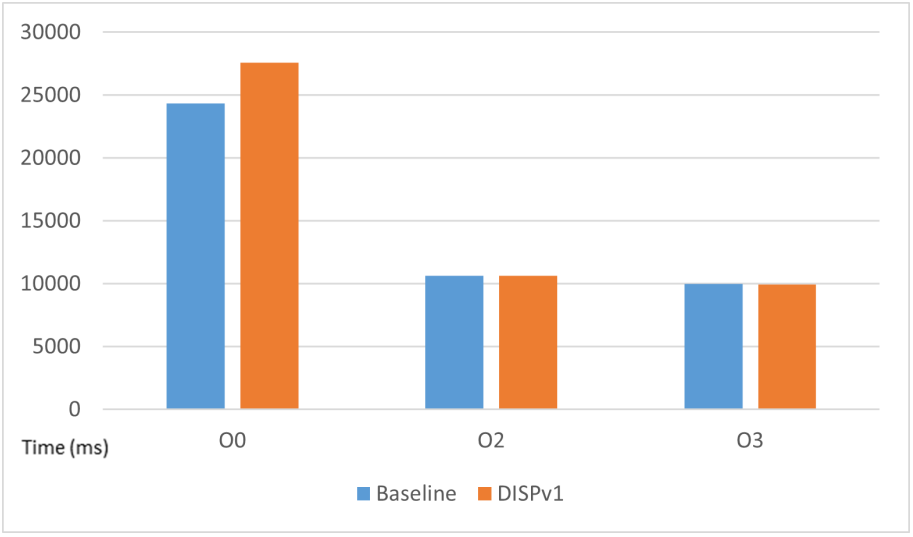


Figure 6.16: Execution times of Disparity hotspot *finalSAD*() in Baseline and DISPv1 using ODROID

that, more than 1000 executions of *float* literals instead of *double* literals are not enough to cause a noticeable reduction in the execution time of an application.

We consider, therefore, that *replaceDoublesWithFloats*() and *replacePowCalls*() did not have a relevant impact in the studied benchmarks. Algorithms looping over targetable code for these scripts would be more suitable for a measurement of their impact.

Conversely, despite maintaining performance, *loopSplitting*() and *loopSplittingFission*(), proved to be more useful, as they remove the risk of human introduced errors during the application of their respective recipes. Because they tend to be applied to enable other transformations, developers can focus on determining the ideal iteration domains for a given loop and using the scripts to refactor the code.

6.5 Summary

To understand the impact of the performance recipes we selected for our portfolio when applied to San Diego Vision Benchmark Suite and conducted a study on the execution times, vectorization logs and cache behaviour of each benchmark version.

The results obtained revealed a variety of effects caused by each benchmark. We have learned that the combination between source-to-source transformations, compiler transformations and available resources is a key factor in high-performance computing. Transformations applied in a source-to-source basis can benefit execution times by reducing the number of instructions, but powerful optimization flags can overshadow this improvement by equaling a Baseline version performance to the modified version's.

We observed that transformations with an enabling nature tend to maintain performance, especially in the presence of compiler optimizations. In the absence of said transformations, Loop

Peeling, in particular, has revealed a potential for performance improvement, which has been correlated to reductions in the number of instructions as well as accesses to memory. Using this transformation we were able to achieve speedups of up to 1.80.

Results in versions that implemented Data Reuse reveal a high efficiency of this technique in improving execution times, regardless of the available resources of each machine. Although, the degree to which this recipe is implemented is an important factor, as Disparity showed overall worse speedups, on relation to the other benchmarks. In the end, this transformation revealed that, under the right circumstances, this transformation can achieve speedups of up to 1.85 when combined with its enabling transformations.

The energy profiling performed on the ODROID embedded system revealed that the transformations applied to SIFT have very little impact on the energy consumed by the memory unit, as this component consumed very little energy from the start. We believe this behaviour has a correlation to an increase in the number of accesses to memory, assuming embedded system behaves similarly to ANTAREX in these metrics.

The two automated scripts we developed for Specialization showed a small adequacy for optimizing the studied benchmarks, but that does not completely rule out their use in the context of other applications which have more uses of literal doubles or calls to the *pow*() function. The two loop transformation automated scripts, on the other hand, proved to be more useful for removing data dependencies and enable other transformations, despite maintaining performance.

Chapter 7

Conclusions

In this chapter we overview the work done, the main contributions, and describe further improvements.

7.1 Summary

This study starts from a need to better understand the impact of performance recipes in applications. In some aspects the relationship between code structure and an application's execution time is unclear, more specifically what techniques should be favoured over others to maximize performance.

Our study compiles in a portfolio a collection of performance recipes that can be applied to various applications and describes the expected results. We also evaluate the advantages of automating each recipe, while stating the main steps required during this process. Understanding the constraints of code refactoring and the resulting improvements it can bring is valuable for any developer that aims to maximize application performance. Recipes such as replacing literal doubles with literal floats, Loop Permutation, Loop Normalization and Function Inlining can be automated to a point where no user input is required. On the other hand, transformations that can be implemented to a multitude of degrees depending on the program structure, such as Loop peeling, Loop Splitting and Data Reuse are possible to automate so that they ideally satisfy very specific conditions, but to universally detect the ideal factors without user input is practically impossible.

Additionally, each selected recipe is implemented and studied in the scope of computer vision, a domain in which performance is essential. The benchmarks chosen are transformed into different versions with different levels of performance recipes applied. The possible root causes found for performance improvements include aspects such as (i) how powerful the performance flags passed to the compiler are, (ii) what the level of parallelization achieved after compilation is, (iii) what structural changes are made to a program or (iv) how the cache behaviour changes according to the applied technique. The data obtained on this subject clarifies that machines with different architectures and computing power can take advantage of different transformations depending on

the amount of resources available to them. Having a clear definition of the characteristics of each benchmark before and after a transformation, along with empirical data on this subject is valuable to the field of high-performance computing as it illustrates correlations between changes and effects. This also enables for the obtained results to be properly validated, challenged or even replicated by interested parties.

7.2 Main Contributions

The main contributions of our work are:

Benchmark Analysis: we profile three benchmarks from the San Diego Vision Benchmark Suite, determine the performance hotspots in each benchmark and describe the code structure of each hotspot function.

Portfolio of Performance Recipes: we propose a portfolio of performance recipes that can be applied to code, describing how each can be implemented, the expected advantages and drawbacks and explore the possibility of automation.

Empirical Data: we apply the selected recipes to each benchmark and present execution time, vectorization data, cache behaviour and energy consumption. The results obtained are interpreted and linked to each transformation or set of transformations applied, giving us an understanding of the impact they had on the application.

7.3 Further Work

Our study leaves a few topics that can be picked up for future work. These are:

Automation: The more recipes are automated, the less developers have to transform code by hand, potentially introducing errors. Recipes such as Function Inlining could be automated to a point where no human input is needed, being applied to programs just before deployment to maximize performance and minimize storage space.

Benchmarks: Despite the three selected benchmarks being important staples in computer vision, they only represent 30% of the benchmarks available in the SVDBS. Conducting this study on more benchmarks would provide us even more insight into the subject of performance recipes.

Metrics: Our work analyses various parameters related to performance, before and after refactoring each benchmark, but there are other metrics that could be extracted and correlated to the performance results obtained, such as the energy profiling of more benchmarks and the speedups of more hotspot functions to complement the global speedups.

Parallelization: Despite all the improvements obtained, we believe that continuing to pursuit parallelization of key loops could increase performance even more.

Tuning: Develop a tuning strategy using Open Tuner to determine the combination of compiler flags that improves the performance of each benchmark the most.

References

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 303–316, New York, NY, USA, 2014. Association for Computing Machinery.

[2] Arm. Arm® cortex®-a76 software optimization guide. Available at https://developer.arm.com/documentation/swog307215/a/, July 2020.

[3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[4] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, July 1994.

[5] João Bispo and João M.P. Cardoso. Clava: C/c++ source-to-source compilation using lara. *SoftwareX*, 12:100565, 2020.

[6] João M.P. Cardoso, Tiago Carvalho, José G.F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz, and Zlatko Petrov. Lara: An aspect-oriented programming language for embedded systems. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*, AOSD '12, page 179–190, New York, NY, USA, 2012. Association for Computing Machinery.

[7] João M. P. Cardoso and Pedro C. Diniz. Modeling loop unrolling: Approaches and open issues. In Andy D. Pimentel and Stamatis Vassiliadis, editors, *Computer Systems: Architectures, Modeling, and Simulation*, pages 224–233, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[8] João M. P. Cardoso and Pedro C. Diniz. *Compilation Techniques for Reconfigurable Architectures*, pages 67–107. Springer US, Boston, MA, 2009.

[9] João M.P. Cardoso and Markus Weinhardt. Xpp-vc: A c compiler with temporal partitioning for the pact-xpp architecture. In Manfred Glesner, Peter Zipf, and Michel Renovell, editors, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 864–874, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[10] João Cardoso, José Coutinho, Tiago Carvalho, Pedro Diniz, Zlatko Petrov, Wayne Luk, and Fernando Gonçalves. Performance-driven instrumentation and mapping strategies using the lara aspect-oriented programming approach. *Software: Practice and Experience*, 46, 12 2014.

[11] João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. Chapter 2 - high-performance embedded computing. In João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz, editors, *Embedded Computing for High Performance*, pages 17 – 56. Morgan Kaufmann, Boston, 2017.

[12] João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. Chapter 3 - controlling the design and development cycle. In João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz, editors, *Embedded Computing for High Performance*, pages 57 – 98. Morgan Kaufmann, Boston, 2017.

[13] Hardkernel Co. Odroid. Available at https://www.hardkernel.com/, December 2020.

[14] Jason Cong, Peng Zhang, and Yi Zou. Combined loop transformation and hierarchy allocation for data reuse optimization. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, page 185–192. IEEE Press, 2011.

[15] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009.

[16] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 10 pp.–, 2001.

[17] Julian Dolby. Automatic inline allocation of objects. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, page 7–17, New York, NY, USA, 1997. Association for Computing Machinery.

[18] J. J. Dongarra and A. R. Hinds. Unrolling loops in fortran. *Software: Practice and Experience*, 9(3):219–226, 1979.

[19] Ozana Silvia Dragomir. *K-loops: Loop Transformations for Reconfigurable Architectures*. PhD thesis, Politehnica University of Bucharest, Bucharest, 2011.

[20] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, January 2001.

[21] J.R. Fonseca. gprof2dot. https://github.com/jrfonseca/gprof2dot, 2013.

[22] A. Fraboulet, K. Kodary, and A. Mignotte. Loop fusion for memory space optimization. In *International Symposium on System Synthesis (IEEE Cat. No.01EX526)*, pages 95–100, 2001.

[23] GNU. 3.11 options that control optimization. Available at https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, September 2020.

[24] GNU. Gcc, the gnu compiler collection. Available at https://gcc.gnu.org/, September 2020.

[25] GNU. Gprof. Available at https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_chapter/gprof_toc.html#TOC1, September 2020.

[26] GNU. Valgrind. Available at https://www.valgrind.org/, December 2020.

[27] GNU. Valgrind user manual - 6. callgrind: a call-graph generating cache and branch prediction profiler. Available at https://valgrind.org/docs/manual/cl-manual.html, December 2020.

[28] Zoltan Herczeg, Daniel Schmidt, Ákos Kiss, Norbert Wehn, and Tibor Gyimóthy. Energy simulation of embedded xscale systems with xeemu. *Journal of Embedded Computing*, 3:209–219, 08 2009.

[29] Robert V. Hogg and Elliot A. Tanis. *Probability and Statistical Inference*. Pearson, 7th edition, 2005.

[30] Jang-Eui Hoing, Ilchul Yoon, and Sang-Ho Lee. Code refactoring techniques for reducing energy consumption in embedded computing environment. *Cluster Computing*, 21, 03 2018.

[31] Intel. Intel® 64 and ia-32 architectures optimization reference manual. Available at https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf, September 2019.

[32] Intel. Intel® vtune™ profiler. Available at https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html, September 2020.

[33] Ken Kennedy and Kathryn S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, page 407–416, Washington, DC, USA, 1990. IEEE Computer Society Press.

[34] Bongjae Kim, Sangho Yi, Yookun Cho, and Jiman Hong. Impact of function inlining on resource-constrained embedded systems. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, page 287–292, New York, NY, USA, 2009. Association for Computing Machinery.

[35] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, July 1996.

[36] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol. Earmo: An energy-aware refactoring approach for mobile apps. *IEEE Transactions on Software Engineering*, 44(12):1176–1206, 2018.

[37] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Monitoring energy hotspots in software. *Automated Software Engg.*, 22(3):291–332, September 2015.

[38] Preeti Ranjan Panda, Luc Semeria, and Giovanni de Micheli. Cache-efficient memory layout of aggregate data structures. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, page 101–106, New York, NY, USA, 2001. Association for Computing Machinery.

[39] Peng Chang, D. Hirvonen, T. Camus, and B. Southall. Stereo-based object detection, classi?cation, and quantitative evaluation with automotive applications. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Workshops*, pages 62–62, 2005.

[40] Tutorials Point. C library - <math.h>. Available at https://www.tutorialspoint.com/c_standard_library/math_h.htm, January 2021.

[41] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, page 249–258, New York, NY, USA, 2006. Association for Computing Machinery.

[42] Qubo Hu, M. Palkovic, and P. G. Kjeldsberg. Memory requirement optimization with loop fusion and loop shifting. In *Euromicro Symposium on Digital System Design, 2004. DSD 2004.*, pages 272–278, 2004.

[43] Ken Kennedy Randy Allen. In *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2001.

[44] Cagri Sahin, Lori Pollock, and James Clause. From benchmarks to real apps: Exploring the energy impacts of performance-directed changes. *Journal of Systems and Software*, 117:307 – 316, 2016.

[45] Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R. Beccari, Luca Benini, Loïc Besnard, João Bispo, Radim Cmar, João M. P. Cardoso, Carlo Cavazzoni, Daniele Cesarini, Stefano Cherubin, Federico Ficarelli, Davide Gadioli, Martin Golasowski, Antonio Libri, Jan Martinovič, Gianluca Palermo, Pedro Pinto, Erven Rohou, Kateřina Slaninová, and Emanuele Vitali. The antarex domain specific language for high performance computing, 2019.

[46] Cristina Silvano, Giovanni Agosta, Stefano Cherubin, Davide Gadioli, Gianluca Palermo, Andrea Bartolini, Luca Benini, Jan Martinovič, Martin Palkovič, Kateřina Slaninová, João Bispo, João M. P. Cardoso, Rui Abreu, Pedro Pinto, Carlo Cavazzoni, Nico Sanna, Andrea R. Beccari, Radim Cmar, and Erven Rohou. The antarex approach to autotuning and adaptivity for energy efficient hpc systems. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, page 288–293, New York, NY, USA, 2016. Association for Computing Machinery.

[47] Byoungro So, Mary W. Hall, and Pedro C. Diniz. A compiler approach to fast hardware design space exploration in fpga-based systems. *SIGPLAN Not.*, 37(5):165–176, May 2002.

[48] Irwin Sobel. An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*, 02 2014.

[49] Graphviz Graph Visualization Software. Graphviz - graph visualization software. Available at http://www.graphviz.org/doc/info/lang.html, September 2020.

[50] Special Purpose Computing Systems. Special purpose computing systems, languages and tools. Available at http://specs.fe.up.pt/, September 2020.

[51] Special Purpose Computing Systems. Lara documentation. Available at http://specs.fe.up.pt/tools/clava/doc/#, January 2021.

[52] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. Sd-vbs: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, 2009.