technische universität
dortmund

Bachelor thesis

Resource-aware optimization by trading locality, redundancy, and parallelism

Benjamin Gläser

April 2017

Supervisors:
Prof. Dr. Jian-Jia Chen
Dipl.-Inf. Ingo Korb

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12 (Eingebettete Systeme)
http://ls12-www.cs.tu-dortmund.de

Contents

1. Introduction **1**
1.1. Goal of the thesis . . . 2
1.2. Structure of the thesis . . . 3

2. Fundamentals of resource-aware programming in *Halide* **5**
2.1. The *Halide* language . . . 5
2.2. Locality . . . 7
2.2.1. Temporal Locality . . . 7
2.2.2. Spatial Locality . . . 7
2.2.3. Application of locality in *Halide* schedules . . . 8
2.3. Redundancy . . . 11
2.3.1. Favouring redundant computations over memory accesses . . . 11
2.3.2. Preventing redundant computations . . . 12
2.4. Parallelism . . . 14
2.4.1. Pitfalls of increasing parallelism . . . 14

3. The Halide auto-scheduler and its limitations **17**
3.1. Development of the auto-scheduler . . . 17
3.2. Parameters of the configurations . . . 18
3.3. Capabilities and limitations of the auto-scheduler . . . 18

4. Benchmarking the auto-scheduler **21**
4.1. Tested platforms . . . 22
4.2. Implementation of the benchmark . . . 23
4.2.1. Configuration-Switcher . . . 23
4.2.2. Pipelines to be inspected . . . 25

5. Evaluation of the benchmark **29**
5.1. Set up of the benchmark result generation . . . 29
5.2. Auto-scheduler performance on ARM-based devices . . . 30
5.3. Benchmark results . . . 31
5.3.1. Effects of the parallelism threshold . . . 31
5.3.2. Performance differences for computationally bounded pipelines . 33

5.3.3. Performance differences for memory bounded pipelines 36

6. Conclusion and outlook **39**

6.1. Summary 39

6.2. Outlook 40

Appendix **41**

A. Full benchmark results 41

B. Reviewed schedules 42

C. Extremely large execution time for first runs 45

List of figures and tables **47**

List of code samples and algorithms **49**

Bibliography **51**

Eidesstattliche Versicherung **53**

Abstract

This thesis analyses the possibility and shortcomings of automated resource-aware optimization by making use of locality, redundancy and parallelism. It is to be expected that different platforms and hardware will see benefits and drawbacks depending on the strategy used.
To study these effects further, a field of application is required, where the three aforementioned approaches can be utilized. Due to its nature and scalability, image processing represents an ideal candidate for this. The choice of image processing is the focus of this work, as it is also a domain that has been extensively researched and gaining importance in recent years.
Whilst most research on image processing acceleration mainly focuses on the ease of implementation and optimization for a specific platform, the thesis at hand aims to investigate the newest development in automatic resource-aware optimization. This requires the identification of patterns and subsets within the possible hardware configurations and image processing pipelines that exhibit similar behaviour, when faced with the challenge of trading locality, redundancy and parallelism. Using these characteristics as a basis, this thesis aims to study their effect on automatically generated schedules.

1. Introduction

Ever since the dawn of readily available computational processing power, a key observation by Gordon E. Moore has held true up until the recent years.

> The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. [9, p. 2]

This exponential growth of chip complexity has been extensively studied and put through the test time. It has shown that this prediction is largely accurate throughout recent computing history. When also accounting for ever increasing clock speeds, as David L. House (director at *Intel*) did, the overall performance gain can be measured as a twofold increase every 18 months [Cf. 3]. But it was without question that the aforementioned gains in clock speed would eventually come to a hold. The results obtained by the *American National Research Council* in "The Future of Computing Performance" [11, p. 9] mark 2004 to be that year, as growing limitations (like heat dissipation) severely hinder further increases in that regard. These induced a shift in computing advances, "[..] as Intel, AMD, and most other vendors turned away from emphasizing clock-based scaling, in favour of adding more CPU cores and improving single-threaded CPU performance [6]."

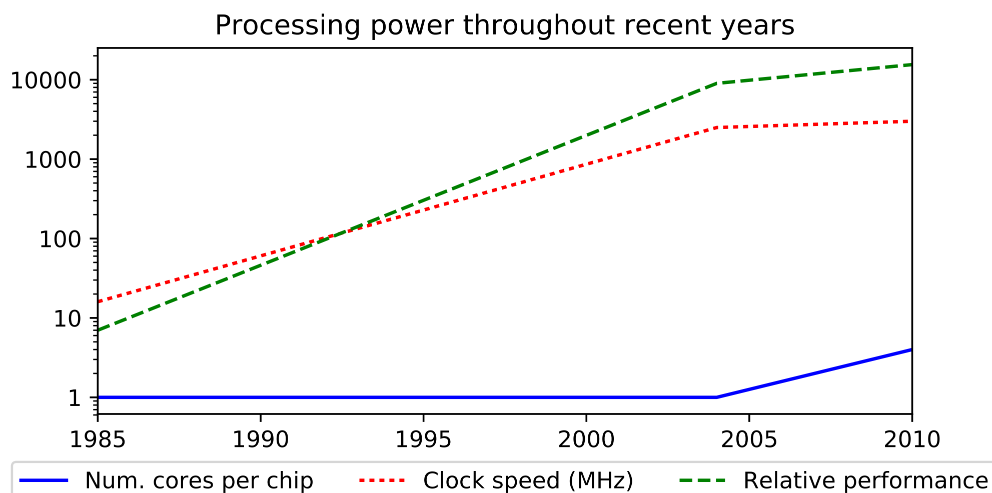

Figure 1.1.: Rise of parallel computing. Remodelled after [11, p. 55].

As the report from the *American National Research Council* lines out, parallel computing is one of the major contributors towards computational improvements. This technique has largely re-established the steady growth of relative performance as pictured by David L. House, albeit at a slower rate.
Yet, this advancement requires the actual usage of the available processing power through multiple cores - this is the main limiting factor, as software which makes use of parallelism is inherently more difficult to write.

Additionally, programmers have to factor in the advantage of using localized computations, as memory controllers have largely adopted advanced mechanisms which benefit from accesses to the same region in memory. This usually comes at the cost of redundant calculations or loss of parallelism. Therefore, adjustments favouring one technique over another have to be made cautiously, with strong consideration of the execution platform. Observing these effects requires an application domain where all of the three strategies (locality, redundancy and parallelism) can be effectively utilized and studied. The prime candidate for this is image processing, for two major reasons: Firstly, this field is one of the many at the forefront of newest computational progression. It is amongst the pioneers which are able to make use of newly available processing power and techniques. Some of the recent developments by early adopters are for example self-driving vehicles or medical image processors. Secondly, image processing is a domain that has been extensively studied, even in regards to possible applications of resource-optimizing techniques. It has therefore spawned many frameworks that simplify the task of implementing these features in your specific image processing pipeline.
One of the most popular and exhaustive frameworks is *Halide* [14] which allows decoupling of the image processing pipelines algorithm and scheduler. The developer is thus granted the ability to adjust and optimize the source-code suiting his hardware-configurations needs, by simply changing the *Halide* scheduling instructions.
Consequently, *Halide* is an excellent basis for further research on the topic of resource-aware optimization in the context of this thesis.

1.1. Goal of the thesis

Since *Halide* has been established as the foundation to implement resource-aware optimization techniques, it is only logical to pursue ways of automatically finding and utilizing the ones that are best applicable. Fortunately, this is one of the major areas of research by the *Halide* developer team, as they are developing an open-source *auto-scheduler* for Halide. The publication of this project in the later half of 2016 even includes extensive benchmark results for an *Intel Xeon E5-2620 v3* which dictate the goals of this thesis. The objective will be the investigation of limitations imposed onto the *auto-scheduler* by the

characteristics of various pipelines. Additionally, particular attention is paid to the reproduction of the original results and benchmarked schedules, which will be performed on devices featuring an ARM-architecture to see how these compare to the x86/x64-based machine.

1.2. Structure of the thesis

The remainder of this work is organized into the following chapters:
Chapter 2 introduces the *Halide* programming language, its purpose and the pertaining problems it is meant to solve. Furthermore, it covers essential knowledge about the concepts of locality, redundancy and parallelism, as well as the interaction and trade-off between them.
Chapter 3 describes the *auto-scheduler* which will form the main point of analysis for the thesis at hand. This requires a closer inspection of its parameters and potential before assessing its effectiveness and possible drawbacks.
Chapter 4 details how the benchmark was constructed in order to test the capabilities of the auto-scheduler. It will also explain the image processing pipelines to be benchmarked and relevant traits they possess.
Chapter 5 is devoted to analysing the generated data of the benchmark and further inspecting special cases found.
Finally, Chapter 6 concludes this paper with a summary of the most important results, whilst also offering avenues of further research.

2. Fundamentals of resource-aware programming in *Halide*

This chapter aims to introduce the essential knowledge required to properly discuss scheduling improvements made possible through *Halide*. But to do so, the first section (Section 2.1) starts with a general overview of the *Halide* programming language. Later on, the *Halide* auto-scheduler will be used to enhance specific image processing pipelines depending on their traits. This requires concepts that these adjustments are based on need to be well understood. Consequently, the next three sections cover the basics about locality (Section 2.2), redundancy (Section 2.3) and parallelism (Section 2.4).

2.1. The *Halide* language

Halide is a functional programming language which is embedded in C++. It has formerly been developed by members of the *MIT CSAIL*[1], *Adobe*, *Stanford University* and other contributors. Nowadays it is still being developed as an open source initiative, with support for many modern computing architectures. These currently include x86/SSE, ARM v7/NEON, CUDA, Native Client, OpenCL and OpenGL on OS X, Linux, and Windows [4].

The conceptual design behind *Halide* allows for easy specification of image processing pipelines. Therefore, it aims to tackle the previously mentioned problem of code that is increasingly difficult to write which has plagued developers trying to fully utilize the given platform resources. Especially when multi-platform compatibility is desired, code readability or performance had to be sacrificed without the use of *Halide*.

On the contrary, *Halides* purpose is to deliver simple, readable "high-performance code by separating the intrinsic algorithm from the decisions about how to run efficiently on a particular machine [13, p. 2]". In practice, this enables the programmer to only have one static image processing pipeline, whilst still maintaining the ability to adjust its execution depending on the platform. The former pipeline can accommodate for complex algorithms, "such as a camera raw pipeline, the bilateral grid, fast local Laplacian filtering, and image segmentation [13, p. 1]".

[1]Massachusetts Institute of Technology - Computer Science and Artificial Intelligence Laboratory

```
1 // Halide declarations
2 Halide::Func input(x, y, c) = cos(x + y + c);
3 Halide::Func blurX, blurY;
4 Halide::Var x, y, c;
5
6 // Actual blur pipeline
7 blurX(x,y,c) =(input(x-1, y, c)+ input(x, y, c)+ input(x+1, y, c)) / 3;
8 blurY(x,y,c) =(blurX(x, y-1, c)+ blurX(x, y, c)+ blurX(x, y+1, c)) / 3;
```

Algorithm 2.1: Sample implementation of 3x3 Blur Pipeline in Halide

As seen in Algorithm 2.1, a fast 3x3 blur code example in *Halide*, the processing pipeline contains no information about intermediate data storage, parallelism or execution order (yet). These instructions, referred to as the "schedule", have to be adapted manually to best fit the execution platform. In basic C++, this would require the manual inspection of multi-threading and vectorization possibilities, as well as the usage of advanced programming techniques, including unrolling, tiling, nesting, fusion and fission. Even experienced developers may struggle with this task and the resulting code will most likely be longer and less intuitive than the *Halide* equivalent. Additionally, changing the manually optimized C++ code to suit a different hardware requires changes on a scope way larger than simply adjusting a *Halide* schedule. How exactly these speed-up strategies are implemented in *Halide*, on top of the relation between them and the computing concepts of locality, redundancy and parallelism, will be covered in the following three sections.

2.2. Locality

The general strategy behind locality, in the context of this thesis, represents the utilization of speed-up techniques modern computing platforms offer in regards to memory accesses latencies. To take a more in-depth look at these, one has to start with a general approach to the principle in question. As *Peter J. Denning* observed,"the locality principle is useful wherever there is an advantage in reducing the apparent distance from a process to the objects it can access [2, p. 24]". It is widely accepted that the aforementioned "proximity between object-accesses" can be differentiated into two subgroups, temporal and spatial locality. In the following two sections, these concepts will be further discussed, together with the resulting applications for them in computer architectures.

2.2.1. Temporal Locality

The concept behind temporal locality describes multiple accesses to the same space in memory within a short time frame, as illustrated by Fig. 2.1. Modern computing platforms are optimized for this scenario by implementing a cache hierarchy: A small, fast cache closest to the CPU and one or more levels of cache beneath it, each larger in size and slower in access speed. This allows for the optimization of memory access times by keeping more frequently used values at the fastest level of cache.

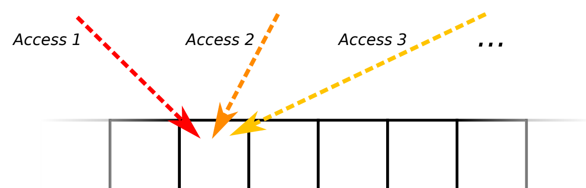

Figure 2.1.: Accesses with temporal locality

2.2.2. Spatial Locality

In contrast to temporal locality, where multiple accesses are only separated by time, spatial locality is the embodiment of multiple accesses within the same memory region. For example, as depicted in Fig. 2.2, an access pattern could load from or write to memory addresses in consecutive order. Optimizing for such patterns is especially important for the application domain in this thesis, image processing, since the image data is usually stored as two (or more) dimensional matrices in consecutive memory space. The most prominent technique used to hide cache latencies occurring during this access behaviour is called *(cache) pre-fetching*, aside from regular benefits achieved through cache line loading. "Cache pre-fetching is a mechanism to speculatively move data to higher levels in the cache hierarchy in anticipation of instructions that require this data [1, p. 2]." This can be realized on a software or a hardware level. Whereas the former

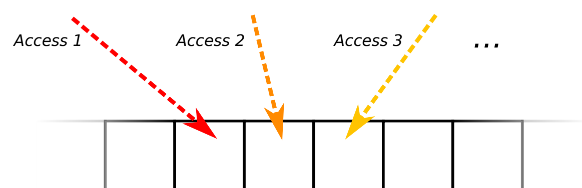

Figure 2.2.: Accesses with spatial locality

is mostly based on compile-time analysis of the source-code to identify memory accesses that can be pre-fetched, the latter uses heuristic algorithms that try to predict cache lines to load, based on recent accesses.

2.2.3. Application of locality in *Halide* schedules

By taking these modern cache features into account, developers can benefit from a significant performance gain, most notably in the domain of image processing. Specifically, *Halide* supports numerous programming techniques that can be applied to the image processing pipelines schedule in order to increase locality. The most effective strategies to accomplish this include- but are not limited to- loop reordering, tiling (sometimes referred to as "Loop blocking") [Cf. 8, pp. 219], as well as code inlining.[2]

Loop reordering is the simplest concept to improve locality. Depending on the architecture of the programming language used, the matrix containing the image data may be saved as row or column major (row first vs column first) in memory. If an algorithm performs a lot of accesses in the same column, a column-major data storage may be preferred to stay within the same cache line. According to this, the traversal order of the loop should also be changed, as shown in the following figure.

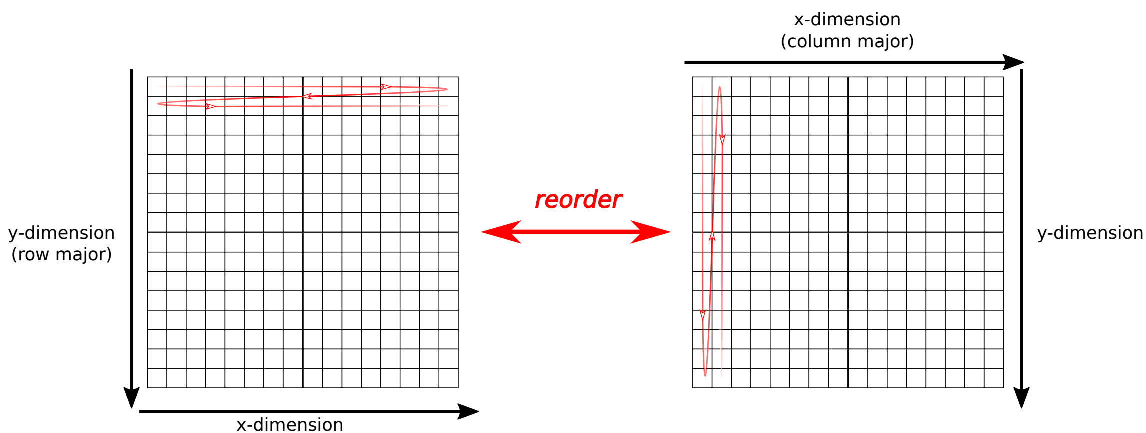

Figure 2.3.: Effects of reordering on loop traversal

Switching of the traversal order can also be accomplished in *Halide*, as shown in Code Sample 2.2.

[2]Since Code inlining majorly impacts the amount of re-computations, it is covered in Section 2.3.

```
1 [...]                      // Blur pipeline from Algorithm 2.1
2 blurY.reorder(y, x);       // New scheduling call to reorder
3 blurY.realize(16, 16);     // Execute the reordered pipeline³
```

Code Sample 2.1: Reordering traversal order in Halide

(Loop) tiling, in the context of image processing, describes the practice of splitting input data into smaller chunks that occupy neighbouring addresses in memory. Optimally, these tiles are small enough to fit into the smaller, faster levels of cache in the hierarchy. This allows pipelines, which use pixel data in close proximity to calculate a single output pixel, to quickly access the required data. We are hereby using the inherent temporal locality to our advantage. A typical example application of this is a blurring pipeline, as seen in 2.1. Consider Figure 2.4, which depicts a regular row major traversal by the algorithm. It is clear that the frequent swapping of the input data available in the fast (L1) cache negatively impacts loading times, due to the blur algorithm re-accessing previously used data which could be no longer present in it *(cache-miss)*.

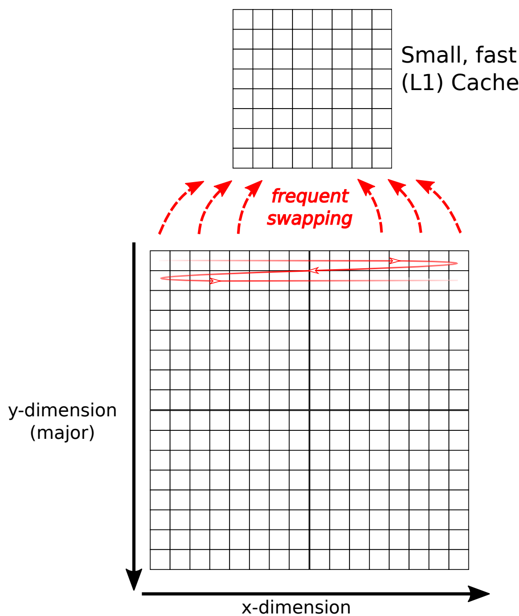

Figure 2.4.: Traversal without tiling

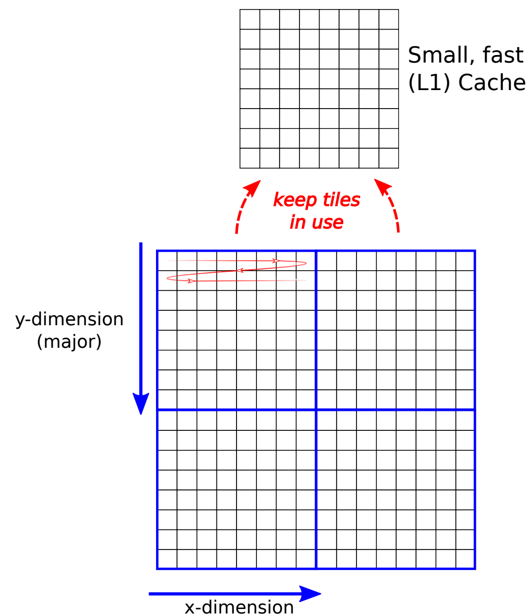

Figure 2.5.: Traversal with tiling

To counteract this, a tiled traversal approach is preferable. By splitting the input data into tiles where all accessed data points fit into one cache level, the amount of quickly accessible data increases (more *cache-hits*). This is illustrated in Figure 2.5, although it needs to be said that it is not always optimal to fit the tile size to the first level of cache. Depending on the pipeline it may be preferable to adjust for L1, L2 or any other number of cache level which is one of the adjustments this work aims to investigate more closely.

[3] The pipeline produces a 16 by 16 pixels large output image, hence the parameters "16,16"

Halide allows testing of these different schedules, as shown in the following Code Sample 2.2.

```
1 [...]                                    // Blur pipeline from Algorithm 2.1
2 Var xOuter, yOuter, xInner, yInner; // New variables for tile indeces
3 // Split transversal into 8x8 tiles
4 blurY.tile(x, y, xOuter, yOuter, xInner, yInner, 8, 8);
5 blurY.realize(16, 16);                   // Execute the tiled pipeline
```

Code Sample 2.2: Tiled traversal in *Halide*

Even though the tiling may yield a reduction in memory latency on its own, it is usually just a pre-requirement for utilizing vectorization and parallelization. The former describes the usage of SIMD (*Single Instruction, Multiple Data*) capability, offered by many modern computing architectures. The latter, parallelism through multi-threading, will be discussed more closely in the later Section 2.4.

2.3. Redundancy

Another resource-aware approach to optimize for the specific hardware configuration is to increase or decrease the amount of redundant calculations performed. This has to be done cautiously, as it is nigh impossible to make accurate predictions for when a re-computation should occur over a memory access. Whilst a re-computation is usually many times faster than a load from cache (especially at the larger, slower levels) or even main memory, it still depends on several other parameters, like CPU frequency, access latencies, cache hits/misses and the specific instruction to compare against. Additionally, possible pipelining of instructions increases this discrepancy in speed, whilst also contributing to the unpredictability. That is why, for the purposes of this thesis (and the soon to be discussed Halide auto-scheduler), this behaviour is parametrized as an average time of re-computation vs access latency at the last cache level (later referred to as *cost balance*). Sensible values will usually range from a factor of 5 up to about 60, as a consequence of the latency incurred due to cache misses.

2.3.1. Favouring redundant computations over memory accesses

In a lot of cases it may be preferable to simply recalculate a value instead of loading it from cache or main memory. This holds especially true for short image processing pipelines with high amounts of temporal locality and few dependencies - for example, the blur pipeline depicted in 2.1.

Code inlining is a technique to replace a function call with its actual definition. If full inlining is desired, this is also done recursively for the calls within. Halide favours this behaviour by default and tries to apply it wherever possible. It is therefore enough to simply call *pipeline.realize(width,height)*, without any other redundancy affecting adjustments, to fully inline all feasible functions. This adjusts the code of the 3x3 blur pipeline during compilation in the following manner, by replacing all:

1. calls to *blurY(x,y,c)* with the corresponding *blurX*
2. *blurX* with the three *input* functions
3. *input* function calls with the resulting operation, in this case *cosine*

By compiling the pipeline down into simple calculation instructions, it naturally prevents having to wait on data calculated by other stages (or threads) and therefore increases locality to a maximum. This avoids the latency penalty of accesses to the cache hierarchy and eliminates possible cache misses. After compilation with *Halide*, the result is the following, fully inlined code for blurY (conceptually):

```
1 blurY(x, y, c) =
2 ((cos((x-1)+ (y-1)+ c) + cos(x+ (y-1)+ c) + cos((x+1)+ (y-1)+ c))/ 3) +
3  (cos((x-1)+    y   + c) + cos(x+    y   + c) + cos((x+1)+    y   + c))/ 3) +
4  (cos((x-1)+ (y+1)+ c) + cos(x+ (y+1)+ c) + cos((x+1)+ (y+1)+ c))/ 3))
      / 3;
```

Code Sample 2.3: Fully inlined blurY calculation

2.3.2. Preventing redundant computations

Contrary to the previous approach of raising the amount of redundant calculations performed, it is sometimes favourable to minimize these. By doing so, the given locality will inevitably be lowered, too, which may be desirable. This can be accomplished by calculating each stage of the pipeline in full before proceeding to the next one. Consequently, pipelines with high spatial locality are always able to reuse previously calculated data, if available.

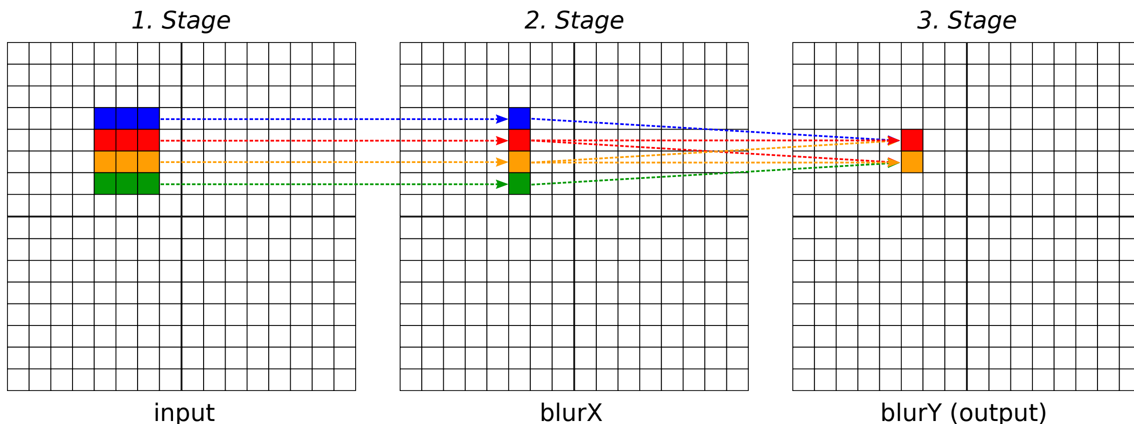

Figure 2.6.: Minimizing redundancy through stage-wise computation

Figure 2.6 illustrates this behaviour in an exemplary way for the 3x3 blur pipeline. From this figure it can be seen that two single (vertically) neighbouring data points already share one third of the data from previous stages of the pipeline, since either blurY calculation refers to one blurX data point that is being reused (of the three data points required for the output). This number further increases when taking all neighbouring points into account, especially since the same behaviour also appears with horizontal adjacency. We can force this style of computation (without any redundancy) in Halide by scheduling it as given in the following Code Sample 2.4.

```
1 [...]                                  // Blur pipeline from Algorithm 2.1
2 blurX.compute_root();                  // Compute blurX before blurY
3 blurY.realize(16, 16);                 // Execute the stage-wise pipeline
```

Code Sample 2.4: Forcing full stage-wise schedule in Halide

Even though this may seem mostly beneficial at first, one has to factor in the downsides of this strategy. Foremost, memory accesses are generally many times slower than a re-computation, as mentioned in the previous section. It therefore requires a very high amount of data re-usage to break even with the redundancy saved. Additionally, it is necessary to keep the previous stage of the pipeline in memory, therefore increasing the likelihood of cache-misses, especially when the computation is not tiled as shown in Section 2.2.3. Another downside is the limit on possible parallelism this technique forces on the pipelines schedule, due to the fact that for each stage to be calculated in full, all threads have to wait for the slowest one to be finished. This will be further discussed in the next section, 2.4 Parallelism.

For these reasons it is usually optimal to find a middle ground for the amount of redundant calculations performed. Nevertheless, a solution that yields the best overall performance, which is still affected by the other two factors (locality and parallelism), usually always features some redundant calculations.

2.4. Parallelism

To follow the trend of ever-increasing core counts in CPUs, it is essential to use multi-threading in any application to optimize for the available resources. It is crucial in the domain of image processing, as the task is highly parallelizable and easily scalable. Additionally, image processing pipelines usually process little to no interdependency between possible threads (in the same stage).
Nevertheless, it is still difficult to implement in plain *C(++)* based fashion and it requires sufficient experience to take full advantage of all possible parallelism options. Again, Halide is an exceptional option to effortlessly implement these multi-threading possibilities, since it provides an easy wrapper for pipeline parallelism.

```
1 [...]                                      // Blur pipeline from Algorithm 2.1
2 Var xOuter, yOuter, xInner, yInner; // New variables for tile indeces
3 // Split transversal into 8x8 tiles
4 blurY.tile(x, y, xOuter, yOuter, xInner, yInner, 8, 8);
5 // Fuse the outer, tile-traversal loop
6 blurY.fuse(xOuter, yOuter, tileIndex);
7 blurY.parallel(tileIndex);                 // Parallelize traversal over tiles
8 blurY.realize(16, 16);                     // Execute the tiled pipeline
```

Code Sample 2.5: Parallel, tiled traversal in Halide

There is one small caveat observable in Code Sample 2.5, the *Halide* implementation of a tiled, parallelized schedule for the 3x3 Blur pipeline: To iterate over the tiled chunks (outer loop) one would normally use two loops, but to allow them to be parallelized by Halide they need to be fused together into a single one.

2.4.1. Pitfalls of increasing parallelism

Whilst image processing is inherently a highly parallelizable field of application, there is a common misconception associated with multi-threading an application: Increasing the amount of threads available for the CPU to process will only ever yield a gain in performance or at least stay on the same level. But this is only sensible up to a certain point, as illustrated in the following Figure 2.7.

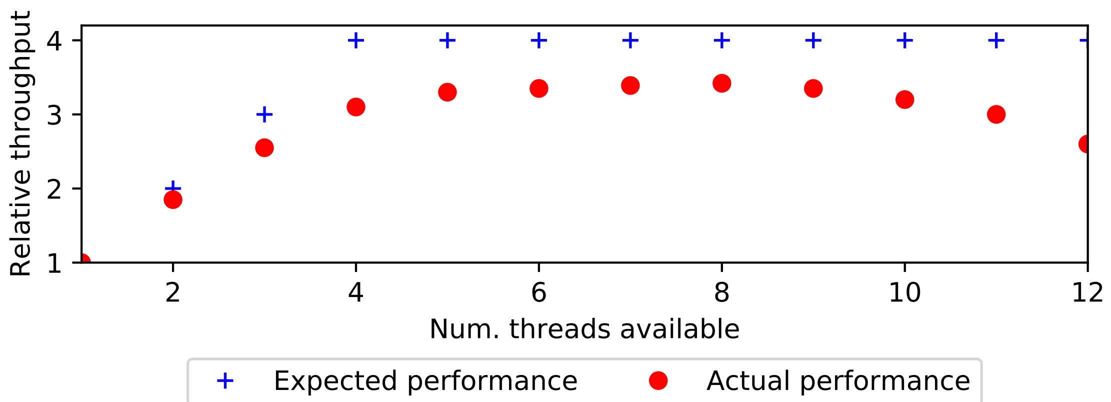

Figure 2.7.: Example throughput for increasing thread counts

As shown in the graph above, there is a significant increase in throughput up until the amount of threads matches the number of cores on the device, which is to be expected. Afterwards, this trend still continues albeit at a decreased rate. This behaviour can be explained by dependencies inside a thread. If one thread has to wait for a value of another task or a blocking memory access, the CPU could switch to the execution of another available thread to effectively use the waiting time. This justifies the continued performance, even after exceeding the number of available cores. Furthermore, the graph suggests that after a certain amount of threads there is no benefit from adding even more. This phenomenon is often overlooked and caused by the following two reasons:

Context switching, which occurs whenever the CPU switches to another available thread. It comes with a CPU cycles overhead, as the old thread data has to be stored and the data (of the thread being switched to) loaded. Increasing the amount of threads will therefore incur ever growing penalties to actual time spent on processing image data. The resulting effect is commonly called *(CPU) thrashing*.

Overlapping, redundant computations caused by the tiling of the input data into regions to be processed by each thread. It is foreseeable that the amount of these grows exponentially with the tiling factor and the number of neighbouring datapoints used. Figure 2.8 represents a visual illustration of the growth for the 3x3 Blur pipeline.

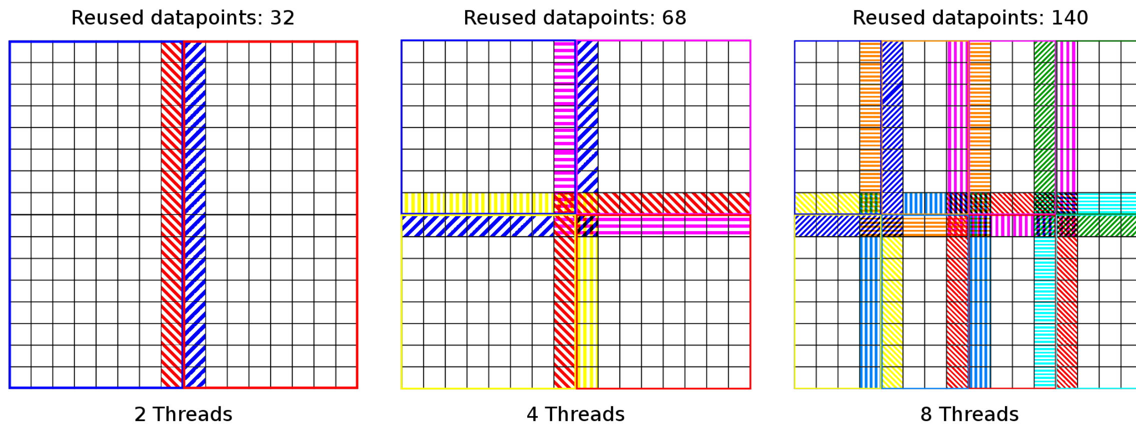

Figure 2.8.: Reuse of data with varying levels of parallelism

Each of the overlapping pixels exacerbates the decision already lined out in Section 2.3 on the programmer: Recompute the overlapping data needed through inlining, therefore wasting a lot of CPU cycles quickly, or wait until the data is available in memory. Either choice would result in slowdowns, as the latter would create a lot of dependencies between the threads, which induces stalling. It is clear that this dilemma can only be solved by limiting the amount of threads available for parallelization upfront. In conclusion, fine-tuning the parallelism to stay within an optimal range of speed-up is of utmost importance for image processing, which will be discussed later regarding the *Parallelism-threshold*.

Having explored various means of resource-aware optimizations in this chapter, it is now obvious that maximizing the performance on a given hardware platform depends on the utilization of all three strategies (locality, redundancy and parallelism). However, this requires in-depth expertise on the subject and their implementation in Halide, which is a daunting task for any programmer. This naturally brings up the question of possible means to automate these scheduling adjustments which will be discussed in the next chapter.

3. The Halide auto-scheduler and its limitations

The remainder of the thesis specifically aims to take a closer look at one of Halides newest developments, the recently proposed Halide auto-scheduler. More specifically, this chapter delivers the required background knowledge to further investigate it. First of all, the goal is to establish an overview on its development (Section 3.1) and how to configure the auto-scheduler (Section 3.2). Afterwards, its capabilities and shortcomings will be further inspected in Section 3.3. These establish the basis for the upcoming benchmark in the following chapter.

3.1. Development of the auto-scheduler

Ravi Teja Mullapudi et al. developed a novel algorithm [Cf. 10, p. 1] to tackle the previously mentioned problem that writing a performant image processing schedule requires in-depth expertise of resource-dependant optimization strategies, such as the ones depicted throughout Chapter 2. This is especially challenging on modern computing architectures, even when already using *Halide* itself to simplify this task. The algorithm, referred to as the **auto-scheduler**, aims to solve this task by automatically adjusting a given pipelines schedule for the best balance between the three core principles (locality, redundancy and parallelism).
It is one of the newest achievements by the Halide developer team, having its accompanying paper only just published in July 2016. Moreover, at the time of writing, the (open-source) repository is still bustling with activity, with updates on a near daily basis. It is therefore important to point out that the data gathered in Chapter 4 is merely a snapshot of performance during the time frame this thesis was conducted (early 2017). More specifically, the remaining parts of the thesis and benchmark are based on the revision from the 3rd of March 2017.[1]

[1] [Commit 2968a1c2b6f58a1bfc67c3abcbcd6f249a3fc8fb] available from `https://github.com/halide/Halide/commit/2968a1c2b6f58a1bfc67c3abcbcd6f249a3fc8fb`

3.2. Parameters of the configurations

The auto-scheduler creates a model to base the schedule on, through the usage of a *machine parameter* configuration. It is based on the following three variables:

- Parallelism threshold: The minimum amount of tiles, and therefore threads, to be created when splitting input data. According to Mullapudi et al., this value is best chosen as a small multiple of the available core count on the execution platform [10, p. 8 & 10].
- (Last level) cache size: As the name implies, this is utilized to create a model of the cache hierarchy. It is also used as a hard limit for the maximum tile size during loop tiling, to ensure that each tile fits into the cache. By varying this, it is therefore possible to adjust tiles to fit entirely into L1, L2 and higher levels of caches.
- Cost balance: An average factor of how many more times expensive a typical main memory access is, opposed to a re-computation on the platform

The auto-scheduler does not require a custom configuration, adjusted to the hardware platform, to be specified. If none is given, a default configuration is used. It utilizes a parallelism threshold of 16, cache size of 16 MB, and a cost balance of 40, which resembles a generic CPU architecture.

3.3. Capabilities and limitations of the auto-scheduler

By itself, the usage of the default configuration already grants a massive speed-up over a non-optimized schedule. Moreover, using a configuration that matches the execution platform, such as the one featured in the original publication, yields even better results. This has been shown in the original publication by comparing the performance against their auto-tuner schedule, which is generated by exploring the entire proposed search space in a brute-force approach. It spans values of 6 to 24 for the parallelism-threshold, 16 to 512 MB in cache size and 5 to 80 as the factor for cost balance. Thus a range of sensible options for higher-end machines are covered. In their benchmarks, the aforementioned schedules were pitted against one without any resource-aware optimizations and another created by an experienced Halide developer.
Their results show that the auto-scheduler is a promising technique to achieve the best possible overall performance on the platform, especially with the configuration tuned to the hardware parameters. In this case, "the auto-scheduler's generated code always remains within a factor of two for all benchmarks, and is within 25% of the best auto-tuned schedule in nine of 14 [10, p. 8]".
But most importantly, these efficient schedules were generated without the in-depth expertise usually required. The author believes that this is the biggest accomplishment,

since it allows programmers, even those new to Halide, to quickly write high performance image processing pipelines.
However, the paper published by the *Halide* developer team also shows that performance can heavily depend on the configuration in some benchmarks.

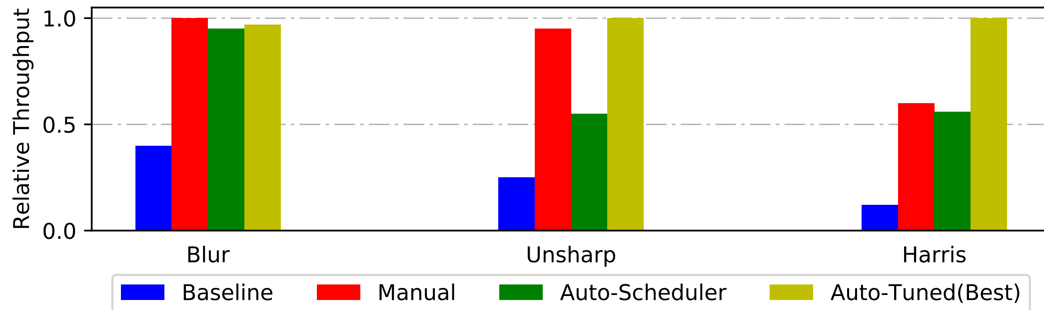

Figure 3.1.: Throughput of Halide schedules for the Blur, Unsharp and Harris pipeline. Remodelled after [10, Figure 6].

This thesis aims to more closely investigate those cases, where the throughput of the hardware-like configuration significantly diverges from the best possible one. The two worst offenders for this are the Harris corner detection and Unsharp mask benchmark, as shown in Figure 3.1. This naturally brings up the question of similarities and discrepancies between the Harris and Unsharp pipeline. To analyse these, another reference point is required, where the default configuration yields performance close to the best possible one. That is why a 3x3 Blur pipeline is also included in the following benchmark. During the implementation of the pipelines in Section 4.2, it is therefore important to discuss relevant characteristics, such as given locality and dependencies in the pipeline.
Furthermore, the goal is to possibly determine how the auto-scheduler incorporates these traits into the resulting schedules. Since those will form the main point of analysis, the author hopes to gain further insight into certain aspects of those schedules, like the preferred strategy between locality, redundancy and parallelism, depending on the hardware configuration.
To further examine this behaviour, an *auto-tuner* like algorithm is required, which allows the exploration of the search space desired for the best configuration by measuring the execution times. The construction of this and the pipelines to be inspected, will be covered in the following chapter.

4. Benchmarking the auto-scheduler

To further inspect the capabilities of the auto-scheduler created by the *Halide* developer team, a benchmark is required. Firstly, it should allow us to see if the auto-scheduler performs on other devices as promised by the research paper [Cf. 10]. Secondly, as mentioned earlier, it should give us the ability to run our own analysis of the pipelines for the Unsharp and Harris benchmark, where the performance of the default configuration is substantially worse than the best one. In overall, the goal should therefore be to gain an overview of feasible configurations for the auto-scheduler and possibly determine trends within those. This naturally leads to the following course of action:

1. Implementing pipelines in Halide for the Harris Corner Detection and Unsharp Mask. Additionally, a simple 3x3 blur pipeline to replicate results with good performance is also desirable.
2. Creating a benchmark wrapper that tests all permutations of possible configurations and feeds these to each pipelines auto-scheduler
3. Mapping each of the benchmarks and their results into graphs
4. Reviewing the differences in schedules between the best, worst and default configuration, as well as other notable outliers and unexpected cases.

Since step 4 requires the closer inspection of the formerly generated results, a second functionality is required for the benchmark: The inspection of previously generated schedules for a specific configuration. Only then it is possible to gain further insight into the differences between locality, redundancy and parallelism of the schedule.

4.1. Tested platforms

The publication released with the auto-scheduler already contains several benchmark results pertaining the throughput on a 6-core Intel Xeon CPU *(Intel Xeon E5-2620 v3)* [Cf. 10]. However, in-depth results for other supported platforms are rather limited. This is the primary reason why the focus for this benchmark is to study a different architecture, namely ARM-based devices. It should therefore be possible to confirm or disprove their claim that "the benefits of auto-tuning schedules for ARM follows similar trends as those reported for [the] Xeon [10, p. 10]".
The benchmark was conducted on the following hardware platforms available to the author, which utilize an ARM architecture: A *Raspberry Pi 2 B*, featuring a 4-core ARMv7 CPU, and an *ODROID-XU4*, with a heterogeneous 8-core ARMv7 CPU. The exact specifications of the devices can be found in the following table.

Device	**Raspberry Pi 2 B**	**ODROID-XU4**
CPU	Broadcom BCM2836	Samsung Exynos 5422
	ARMv7 4x Cortex-A7@900MHz	(heterogeneous) ARMv7 4x Cortex-A15@2.1GHz 4x Cortex-A7 @1.3GHz
Cache	8x 32 KB L1i-cache 8x 32 KB L1d-cache 512 KB L2-cache(shared)	8x 32 KB L1i-cache 8x 32 KB L1d-cache 2 MB L2-cache (shared) 512 KB L2-cache (shared)
Memory	1GB LPDDR2 SDRAM (shared with GPU)	2GB LPDDR3 RAM (shared with GPU)

Table 4.1.: Specification of the tested devices[1]

[1] SoC manufacturers usually do not release specifications, these were measured by third parties. Confer to [15], [16] for more details.

4.2. Implementation of the benchmark

The benchmark itself contains two major components: An algorithm to loop through all possible configurations to test the auto-scheduler with *(configuration-switcher)*, and the pipelines to be examined, implemented in Halide. These will be covered in their respective, following sections.

Aside from these, a wrapper program has been created which allows setting of various options for the benchmark, e.g. whether the input image is a pre-existing image or should be randomly generated, accompanied by the pertaining parameters height and width. Additionally, the specification of a seed number for the random image generator, setting explicit starting configuration parameters and the possibility to choose between different pipelines to test allow for repeatability of single test runs. This enables the user to inspect noteworthy schedules in more detail.

4.2.1. Configuration-Switcher

In order to be able to inspect all possible auto-scheduler configurations, the ability to loop through all parameter permutations is required (Parallelism-threshold, (Last) Cache size, Cost balance). The following pseudo-code, Algorithm 4.1, depicts such an implementation with C++ specific techniques omitted in favour of better readability.[2]
The configuration-switcher itself requires the range of the parameters to be given (lines 1-7), which are mainly based on the auto-tuner search space used by the original developers. For the purposes of this paper, the spectrum was extended and concentrated on the lower end of the possible configurations. This is due to the ARM-base devices possessing fewer cores and a smaller cache hierarchy than their counterpart used in the original publication[3]. The remaining functionality of the configuration-switcher is divided into two main functions, **grabConfig()** and **calculateNextConfig()**. The former (lines 9-15), as the name implies, returns the current configuration as machine parameters to be passed to the auto-scheduler. The latter, calculateNextConfig (lines 17-29), moves the current configuration to the next one. This is done in a fashion that first varies the parallelism-threshold, then cache size, and last the cost balance.

[2]Full source-code, along with the results, obtainable in Appendix A
[3]See previous Section 4.1 for more information on the hardware specifications.

```
1: ParallelismThresholds ← [1, 2, 3, 4, 8, 16]
2: CacheSizes ← [8, 16, 32, ..., 1024 * 1024]
3: CostBalances ← [5, 10, 15, 20, 30, 40, 60, 80, 120]
4:
5: StartingIndices ← [1, 1, 1]                        // Allow specific starting
                                                      // configurations for later review
6: CurrentIndices ← StartingIndices                   // Copy by value
7: ParamVariations = [ParallelismThresholds.length, CacheSizes.length, CostBalances.length]
8:
9: function grabConfig
10:     config ← new MachineParameters
11:     config.parallelism ← ParallelismThresholds[CurrentIndices[1]]
12:     config.cache_size ← CacheSizes[CurrentIndices[2]]
13:     config.balance ← CostBalance[CurrentIndices[3]]
14:     return config
15: end function
16:
17: function calculateNextConfig
18:     CurrentIndices[1] + +
19:     for IndexIterator ← 1, CurrentIndices.length do
20:         if CurrentIndices[IndexIterator] > ParamVariations[IndexIterator] then
21:             CurrentIndices[IndexIterator] ← 1
22:             if IndexIterator + 1 ≤ CurrentIndices.length : then
23:                 CurrentIndices[IndexIterator + 1] + +
24:             else
25:                 CurrentIndices ← StartingIndices   // Copy by value
26:             end if
27:         end if
28:     end for
29: end function
```

Algorithm 4.1: Pseudo-Code of Configuration-Switcher

4.2.2. Pipelines to be inspected

In this section we move on to the implementation of the pipelines to be inspected. As argued in Section 3.3, we focus on the Blur, Unsharp Mask and Harris Corner detection algorithms. Whilst the *Halide* sourcecode of the various pipelines will be shown, it is not the focus of this paper to analyse the specific implementation in detail. In the context of this thesis it is more important to gain an overview of characteristics and differences between the pipelines, in light of possibly determining how these impact the results which will be evaluated in the next Chapter.

3x3 Blur

The first image processing pipeline to investigate is a simple 3x3 Blur algorithm. Of the three, Blur, Unsharp and Harris, it is the most straightforward one.

```
1 // Repeating boundaries for full evaluation of edge datapoints needed
2 Func inBounded = BoundaryConditions::repeat_edge(inputImage);
3
4 // Variable and Func declarations omitted for better readability
5 blurX(x, y, c) = (inBounded(x-1, y, c)  + inBounded(x, y, c) +
        inBounded(x+1, y, c))/3;
6 blurY(x, y, c) = (blurX(x, y-1, c)     + blurX(x, y, c)     +
        blurX(x, y+1, c))    / 3;
```

Algorithm 4.2: 3x3 Blur pipeline implementation with repeated boundaries

In the previous, and all subsequent algorithms, the input data is first put through a repeating-edge boundary condition (line 2). This repeats the edge data points further outwards, so that accesses by the pipeline outside of the given region are still computable which ultimately prevents shrinking the output image that would otherwise occur.

As seen in Algorithm 4.2 and its corresponding Figure 4.1, the blur pipeline depicted only features two stages. As there are no dependencies beyond the previous stage, it would therefore be possible to fully inline *blurX* into *blurY*. Another notable characteristic of the algorithm is the fair amount of locality, as each stage uses 8 neighbouring data points to compute a single output pixel.

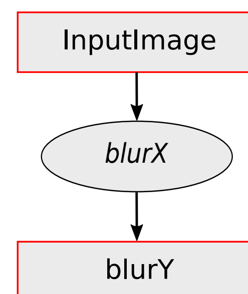

Figure 4.1.: DAG[4]Visualization of the Blur Pipeline[5]

[3]Directed acyclic graph
[5]The input and output stages have been highlighted to differentiate these from the rest of the pipeline.

Unsharp Mask

The next, more complex, pipeline for the benchmark is composed of the Unsharp Mask algorithm. It is used for computation of an artificially sharpened image by weighting each pixel with the difference between a grayscale version of the input and its resulting blurred variant.

```
1 // Variable and Func declarations omitted for better readability
2 // The grayscale weighting stems from the fact
3 // that humans do not perceive all colors equally6
4 gray(x, y)       = 0.3f*inputBounded(x,y,0) + 0.59f*inputBounded(x,y,1) +
      0.11f*inputBounded(x, y, 2);
5 blurX(x, y)      = (gray(x- 1, y)  + gray(x, y)  + gray(x+ 1, y))  / 3.0f;
6 blurY(x, y)      = (blurX(x, y- 1) + blurX(x, y) + blurX(x, y+ 1)) / 3.0f;
7 sharpen(x, y)    = 2.0f*gray(x, y) - blurX(x, y);
8 ratio(x, y)      = sharpen(x, y)   / gray(x, y);
9 unsharp(x, y, c)= ratio(x, y)      * inputImage(x, y, c);
```

Algorithm 4.3: Unsharp pipeline implementation with repeated boundaries

This implies that, at its core, the Unsharp Mask pipeline depicted in Algorithm 4.3 reuses the method of the previous 3x3 Blur on the grey-scale version of the input image. Therefore it is to be expected that access patterns for the Unsharp Mask feature an similar amount of spatial locality as the previous pipeline.

Moreover, the algorithm introduces a set of dependencies as illustrated by Figure 4.2: The *gray* computation (and the inputImage to a lesser extent) is re-accessed at later stages by the pipeline, specifically during *sharpen* and *ratio* (as well as the final stage, *unsharp*, for the inputImage). Whilst this does grant a degree of temporal locality, it also requires the previous stages to be kept in memory and severely hinders the feasibility of recomputations. In return, this puts a damper on parallelizability, due to the fact that reused data points (as explained in Section 2.4) can not simply be recomputed.

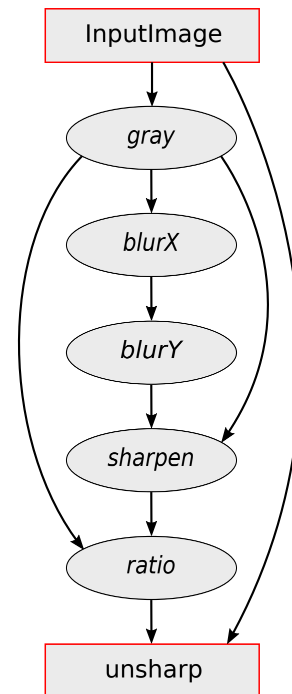

Figure 4.2.: DAG Visualization of the Unsharp Pipeline

[6] Further Information available in [7].

Harris Corner Detection

The last algorithm to be benchmarked, under utilization of the Halide auto-scheduler, is the commonly abbreviated Harris Corner Detection. It is named after Chris Harris, one of the original inventors, and is actually intended as "a combined corner and edge detector [5, p. 1]".
The following Algorithm 4.4 shows an implementation of Harris in Halide, based on the original paper by Chris Harris and Mike Stephens.

```
1  // Variable and Func declarations omitted for better readability
2  gray(x, y)        = 0.3f*inputBounded(x, y, 0) + 0.59f*inputBounded(x, y,
       1) +
3                      0.11f*inputBounded(x, y, 2);
4  intensityX(x, y)= gray(x-1, y-1)* (-1.0f/12)+ gray(x+1, y-1)* (1.0f/12) +
5                    gray(x-1, y)   * (-2.0f/12)+ gray(x+1, y)   * (2.0f/12) +
6                    gray(x-1, y+1)* (-1.0f/12)+ gray(x+1, y+1)* (1.0f/12);
7
8  intensityY(x, y)= gray(x-1, y-1)* (-1.0f/12)+ gray(x-1, y+1)* (1.0f/12) +
9                    gray(x, y-1)   * (-2.0f/12)+ gray(x, y+1)   * (2.0f/12) +
10                   gray(x+1, y-1)* (-1.0f/12)+ gray(x+1, y+1)* (1.0f/12);
11
12 intensityXX(x, y) = intensityX(x, y) * intensityX(x, y);
13 intensityXY(x, y) = intensityX(x, y) * intensityY(x, y);
14 intensityYY(x, y) = intensityY(x, y) * intensityY(x, y);
15
16 sumXX(x, y)=intensityXX(x-1,y-1)+intensityXX(x-1,y)+intensityXX(x-1,y+1)+
17             intensityXX(x, y-1) +intensityXX(x, y) +intensityXX(x, y+1) +
18             intensityXX(x+1,y-1)+intensityXX(x+1,y)+intensityXX(x+1,y+1);
19 sumXY(x, y)=intensityXY(x-1, y-1) + [...] + intensityXY(x+1, y+1);
20 sumYY(x, y)=intensityYY(x-1, y-1) + [...] + intensityYY(x+1, y+1)
21
22 determinate(x, y)= sumXX(x, y) * sumYY(x, y) - sumXY(x, y) * sumXY(x, y);
23 trace(x, y)       = sumXX(x, y) + sumYY(x, y);
24 //k has to be determinated per run, typical k value here (k = 0.09)
25 harris(x, y)      = determinate(x, y) - 0.09f * trace(x, y) * trace(x, y);
```

Algorithm 4.4: Harris pipeline implementation with repeated boundaries

From the source code it is inferable that this algorithm would be the most computationally expensive of three, due to the high amount of calculations in each stage. Whilst it will be shown to hold true later, there are factors to consider which positively affect the generated schedule's performance and thus brings the throughput on a level close to the previous pipeline.

Foremost, Harris Corner detection does not include over-arching dependencies, like the previous Unsharp Mask algorithm. In comparison, Figure 4.3 shows that only the *determinate* stage has computation results that get used beyond the following stage. This should allow for a great deal of code-inlining and thus parallelizability, since the remaining stages follow a strictly sequential order. The only requirement is the full computation of *intensityX* and *intensityY*, since it is a prerequisite for *intensityXY* (and subsequently *determinate*).

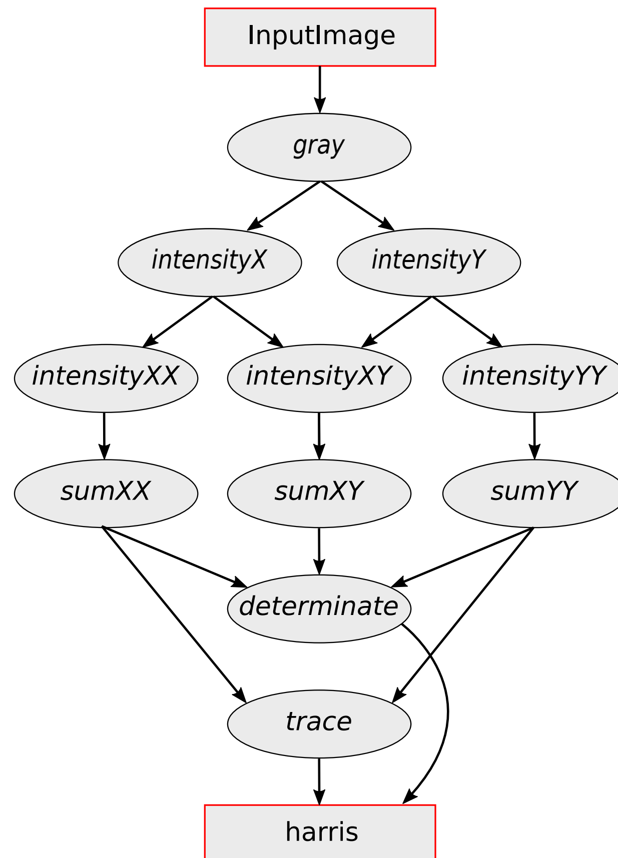

Figure 4.3.: DAG Visualization of the Harris Pipeline

Furthermore, the various stages which require accesses to neighbouring data points (*gray, intensityX/Y, sumXX/XY/YY*) all use the same eight surrounding pixels. This behaviour in spatial locality allows keeping the tiling (and parallelism) options through-out the stages the same.

5. Evaluation of the benchmark

Having previously covered the pipelines' implementation and their characteristics, as well as the benchmark, this chapter aims to put these to the test. The first section of this chapter covers the benchmark parameters which were used to generate the results. Using those, the following section aims to evaluate the results by inspecting schedules that show significance.

5.1. Set up of the benchmark result generation

To produce the results evaluated in this chapter, a randomly generated image sized 4096 by 4096 pixels (8192 wide for the ODROID-XU4) was supplied as the input for each pipeline tested. The auto-scheduler was then utilized to generate a schedule with varying parameters. As previously explained in Section 4.2, the configuration switcher is used to permutate through all sensible options for the auto-scheduler, which in return creates the schedules for the three pipelines tested. For every pipeline and each configuration, the algorithm resulting from the auto-scheduler was then executed 25 times. In overall, running the entire benchmark set on the Raspberry Pi 2 and ODROID-XU4 (with the larger input image) took several days each, due to more than 700 possible permutations in the configuration search space.

The resulting data of the benchmark was then converted into comprehensible diagrams. To do so, the top and bottom outliers in each execution time dataset were removed first, to account for unintended inaccuracies[1]. Afterwards it was observable that the variation in execution time for each configuration is minuscule (< 1% difference between best and worse). Therefore, opting to represent each configuration through its average execution time in the diagrams is a viable option. This allows for additional, condensed views of the data. The full benchmark results and all diagrams generated in this fashion can be obtained from Appendix A.

[1]Earliest version of the benchmark had a fault where the first execution took much longer than the remaining runs. This was solved, as explained in Appendix C.

5.2. Auto-scheduler performance on ARM-based devices

The first comparison to be made is between the general behaviour of the auto-scheduler on ARM devices, as opposed to the results of the *Intel Xeon E5-2620 v3* depicted in Section 3.3. To do this, each device had three datasets extracted for all of the benchmarks. These datasets corresponded to the three most notable configurations:

- Default: Running the auto-scheduler without any customized parameters
  (Parallelism threshold:16, cache size:16 MB, cost balance:40)
- Real: A configuration as close as possible to the hardware specification needs (Compares to *Auto-Scheduler* in Figure 3.1)
  (Raspberry Pi 2: Parallelism threshold 8, cache size 512 KB, cost balance 10;
  ODROID-XU4: Parallelism threshold 16, cache size 512 KB, cost balance 10)
- Best: The configuration parameters which delivered the best performance, determined by the full exploration of the search space through the benchmark in this thesis. (Compares to *Auto-Tuned(best)* in Figure 3.1)

To determine the values of the parallelism threshold and cost balance for the "real" configuration, the recommendation by Ravi Teja Mullapudi et al. was followed. They advise a choice of "a small multiple of the core count" (in this case 2 times) for the former, and a value of 10 for the latter in their short, ARM-based benchmark [Cf. 10, p. 10]. The cache size parameter reflects the actual L2 cache capacity.

The Figure 5.1 shows the resulting plot for each of three datasets per configuration and

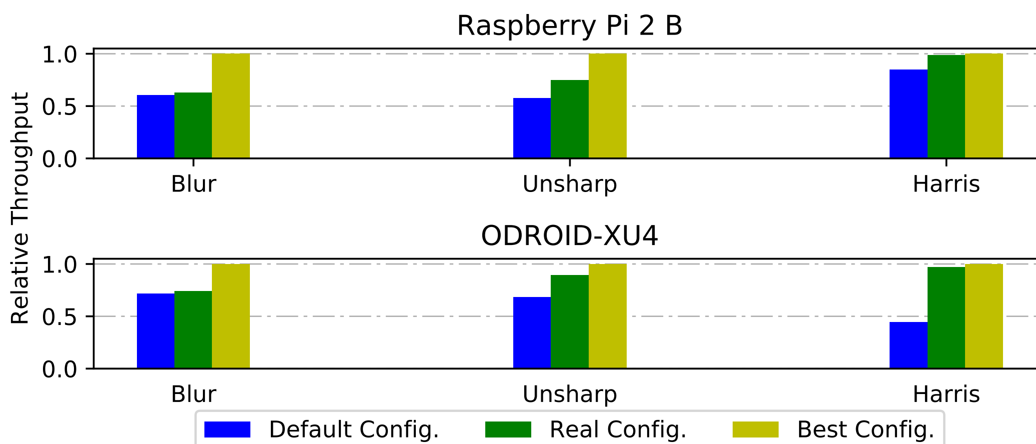

Figure 5.1.: Throughput of the auto-scheduler on ARM-based devices

device. The figure indicates that it is always beneficial to use a configuration that more closely resembles the hardware specification instead of the default, which was to be expected.

But, when comparing these graphs to Figure 3.1, our data paints a different picture for the discrepancy between the Hardware-adjusted *(real)* configuration and the best possible one. The correlation between these two cases is actually inverted on ARM-devices, compared to the x86/64 machine in the original benchmark. Performance results for the hardware-specific configuration during blur are significantly worse than the original data would suggest. On the opposite end, Unsharp and Harris show a much smaller gap between the *real* and best configuration. This discovery is another interesting point of observation, which will be explained by investigating the underlying schedules in the next sections.

5.3. Benchmark results

After carefully reviewing the previously generated results from the benchmark, the author believes that the following order of evaluation for the schedules is the most sensible one: Firstly, discussing the behaviour of the auto-scheduler in regards to the *parallelism-threshold* and how this affects the further examination of the data. Secondly, inspecting the differences between pipelines and how the parameter choices affect the execution times. This requires the categorization of the algorithms into computationally and memory bounded pipelines, as well as the review of schedules in specific case studies.

5.3.1. Effects of the parallelism threshold

The most striking behaviour for the auto-scheduler concerns the first parameter, the *parallelism-threshold*. There are two observations to be made:
First of all, the poor performance when choosing the value for this parameter smaller than the core count. This is not in line with the explanation given for the purpose of the parallelism threshold in the original paper by Ravi Teja Mullapudi et al:
In their pseudo-code, the input tiling for each *Halide* function is adjusted to minimize the amount of accesses required for the computation [Cf. 10, p. 6 - Listing 1, lines 17–19]. This is supposedly done by checking through all possible tiling sizes and dismissing the ones that do not result in a sufficient amount of parallelism (the *parallelism threshold* describes this minimum). Unsurprisingly, this would result in one huge tile (and no parallelism) given *parallelism-threshold=1*, since it has the minimal amount of redundant loads. But, the secondary condition of a viable tiling size should prevent this: The tile is required to use less memory than the parameter *cache size* allows. The example in Figure 5.2 shows that even with a choice of just 32 KB for this parameter, the unfavourable tiling with little to no parallelism still gets picked. This behaviour is replicable on all devices and benchmarks with any choice of *cache size*, which indicates faults or differences within the actual implementation compared to the pseudo-code given in the original paper. Due to

time constraints, it was not possible to further investigate this. Nevertheless, choosing a threshold smaller than the core count should therefore be avoided at all costs.

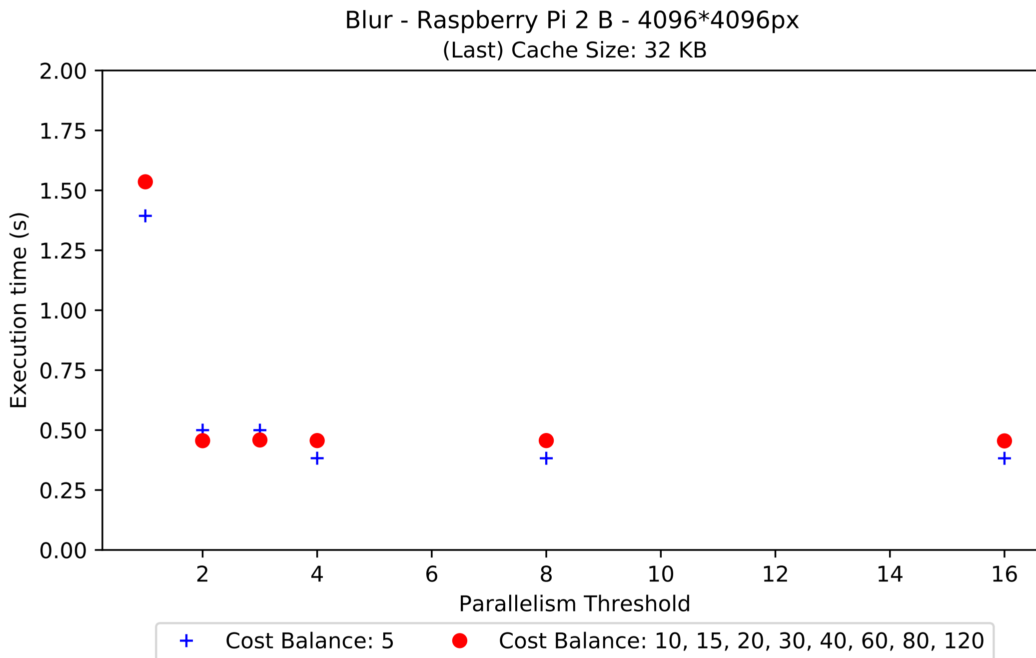

Figure 5.2.: Effects on the auto-scheduler by low levels of parallelism threshold[2]

Furthermore, if the *parallelism threshold* is at least equal or greater than the core count of the device, then it does not have any effect on the schedule generated (they are identical, as seen from Algorithm B.1 and Algorithm B.2). This suggests that the mechanism described earlier does manage to always find the same, feasible tiling solution, as long as the parallelism threshold value is not too low.

In retrospect, this indicates that the search space for the parallelism threshold would have been better chosen in a higher range, possibly with maximum values so large that CPU thrashing can be shown. Quick, single configuration tests show that these effects start to appear at 64 and more threads. This amount of multi-threading can be forced by the use of the minimum parallelism threshold. For now this invariance in sensible ranges means it is sufficient to investigate cases with thresholds of 8 or 16.

[2] In this and all following diagrams, datasets which barely show any difference (for example, cost balance: 10,15...,120) have been grouped together for visual clarity.

5.3.2. Performance differences for computationally bounded pipelines

As pointed out in Section 4.2.2, the various pipelines show characteristics which are ultimately reflected in the resulting benchmarks. The most interesting connection can be made in regard to the *cache size* parameter: Pipelines that possess few dependencies, such as **Blur** and **Harris corner detection**, benefit from code inlining and parallelism the most, which is why these are **computationally bounded**. Due to this, most values are computed as needed and little time is spent waiting on cache accesses. Small benefits are noticeable though, when the *cache size* is geared towards the L2 cache capacity. Figure 5.3 illustrates this for Harris, on the faster *ODROID-XU4*.

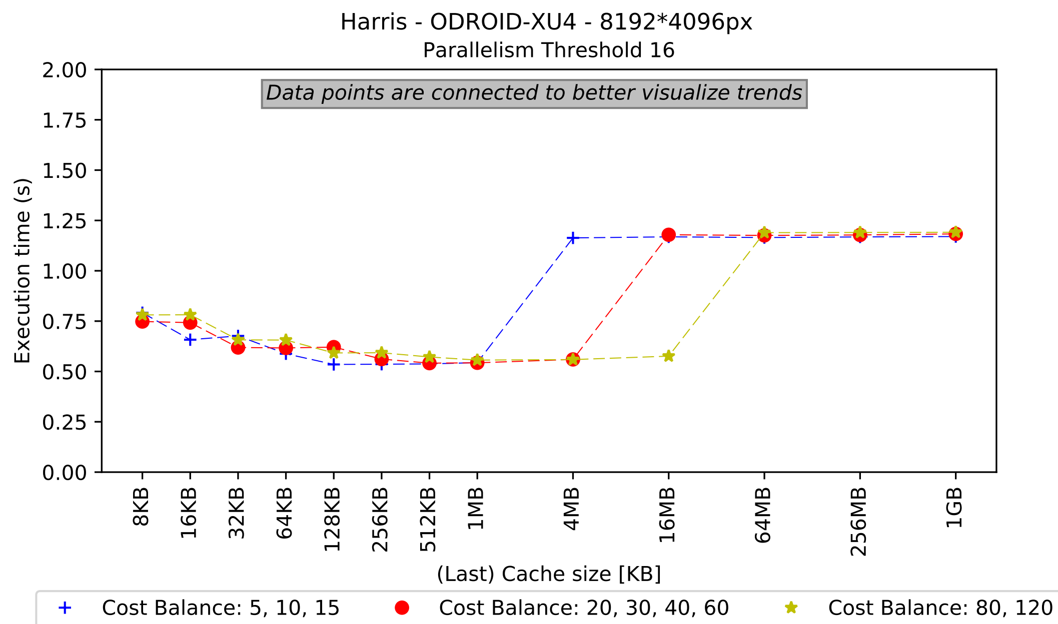

Figure 5.3.: Harris benchmark on the ODROID-X04 for varying cache sizes

As soon as the *cache size* parameter exceeds actual available cache capacity, performance takes a massive hit, since main memory loads start to weight into the execution time. This very same behaviour can be observed in the other computationally bounded benchmark (blur), even on the less performant Raspberry Pi 2 B. This is demonstrated in Figure 5.4
Here, the gap between *cache size* choices that fit into the L2 and those that do not, is not nearly as large as the previous diagram would suggest. This can be explained by the inherently slower hardware specification of the platform.[3] As a result, the computations take up a larger part of the execution time, whilst the memory access latencies stay roughly the same.
Therefore the additional main memory access penalty does not affect the overall execution time as much, when the *cache size* is larger than the actual capacity available.

[3]Confer to Section 4.1 for further details.

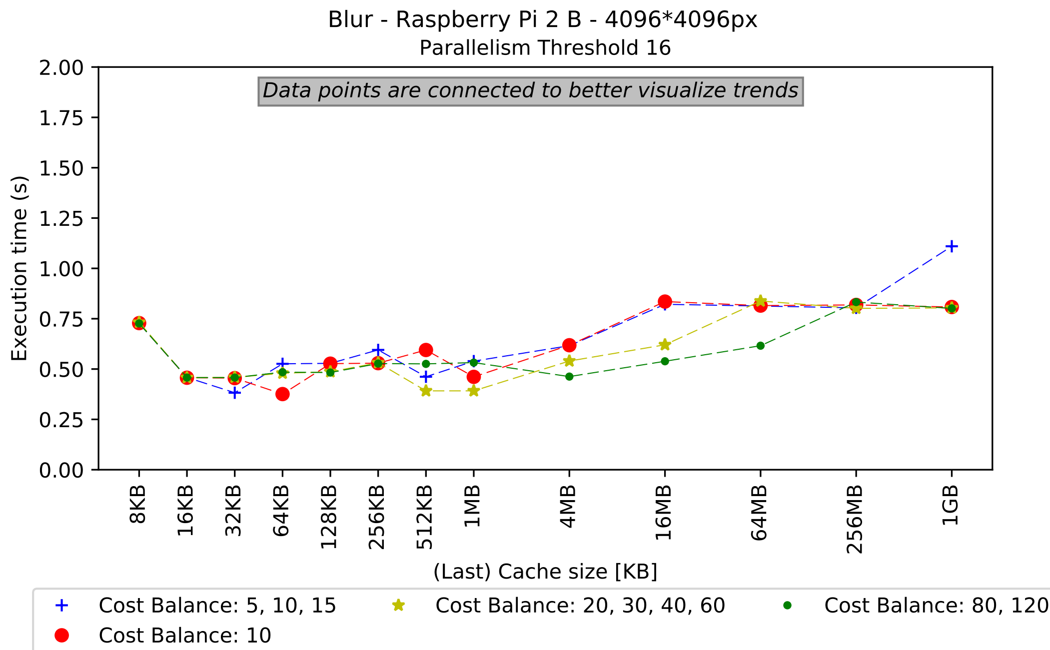

Figure 5.4.: Blur benchmark on the Raspberry Pi 2 B for varying cache sizes

When trying to maximize the throughput, the choice of fitting the *cache size* parameter to the L2 capacity delivers the best performance. Here the unusually poor results of the "real" configuration (*parallelism threshold* 8/16, *cache size* 512 KB, *cost balance* 10) in Figure 5.1 can also be explained. When comparing the schedule with a cost balance of 10 (Schedule B.4) against the ones with the surrounding values, 5 (Schedule B.3) and 15 (Schedule B.5), one key difference is observable: These feature an additional layer of parallelism for the first stage of computation.[4] It is therefore safe to say that this result is an outlier, since the behaviour for low *cache sizes* seems to be rather inconsistent for this benchmark.

Another noteworthy point of investigation deals with the impact of the *cost balance* on these computationally bounded schedules when using large *cache size* parameters. It is very striking that the *cost balance* datasets were assignable to distinct groups and their resulting tiers of performance. Upon further exploration of the schedules generated by the auto-scheduler, it turns out that *cost balances* within the same groups actually have identical schedules.
Additionally, it can be seen that these tiers of *cost balance* directly correlate to the chosen *cache size* in performance. The larger the *cost balance* value is, the stronger the incentive to

[4]Schedule B.5 also uses smaller tiles, which makes this schedule slightly more performant than Schedule B.3. The majority of speed-up is gained through the parallelism though.

avoid the use of main memory - even if the *cache size* parameter was chosen to be larger than the capacity available.

5.3.3. Performance differences for memory bounded pipelines

On the flip side, **memory bounded** pipelines are rich in dependencies and offer little room for full code inlining. The **Unsharp Mask** algorithm matches these criteria, as discussed in Section 4.2.2.
It is plain to see that these pipelines spend most of their time waiting for memory loads, since the dependencies arching over multiple stages do not allow this data to be kept in cache.[5] Therefore it is not surprising, that on a sufficiently fast machine, like the ODROID-XU4, the performance is largely unaffected by the choice of *cache size* and *cost balance*.

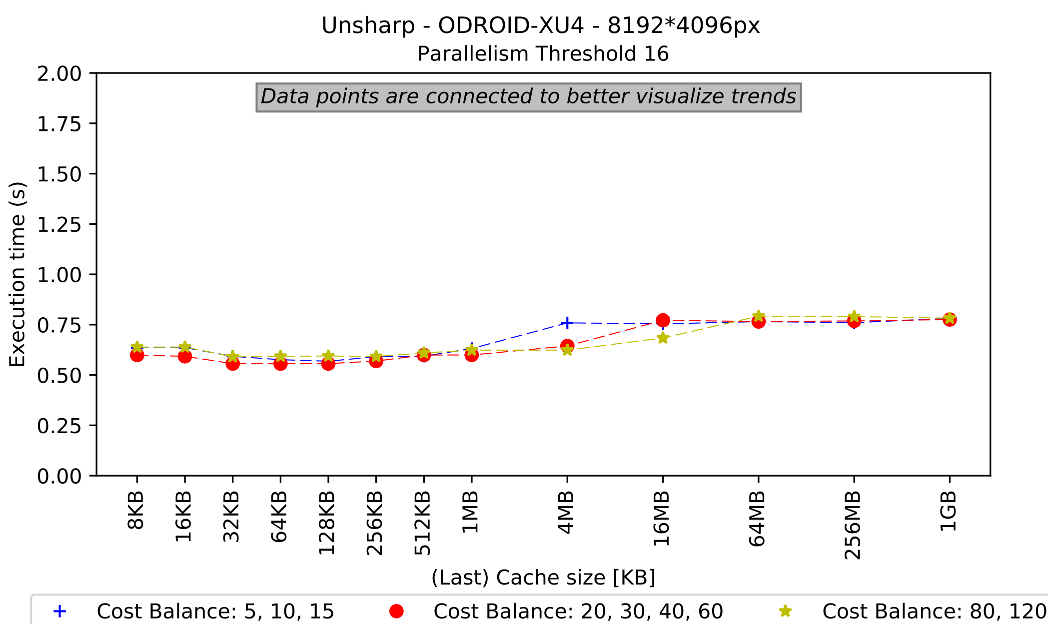

Figure 5.5.: Unsharp benchmark on the ODROID-XU4 for varying cache sizes

There is a small trend observable in Figure 5.5 though, that in contrast to the computationally bounded pipelines, the best results are yielded by adjusting the *cache size* to the L1 cache capacity. Moving onto the less performant platform, this trend is confirmed to hold true to an even greater extent. As the following diagram shows, performance is always guaranteed at small levels of *cache size* (for this memory bounded pipeline) no matter the choice of *cost balance*.

[5]Assuming the input image is larger than any typical cache sizes - which is to be expected when applying resource-aware techniques to image processing pipelines

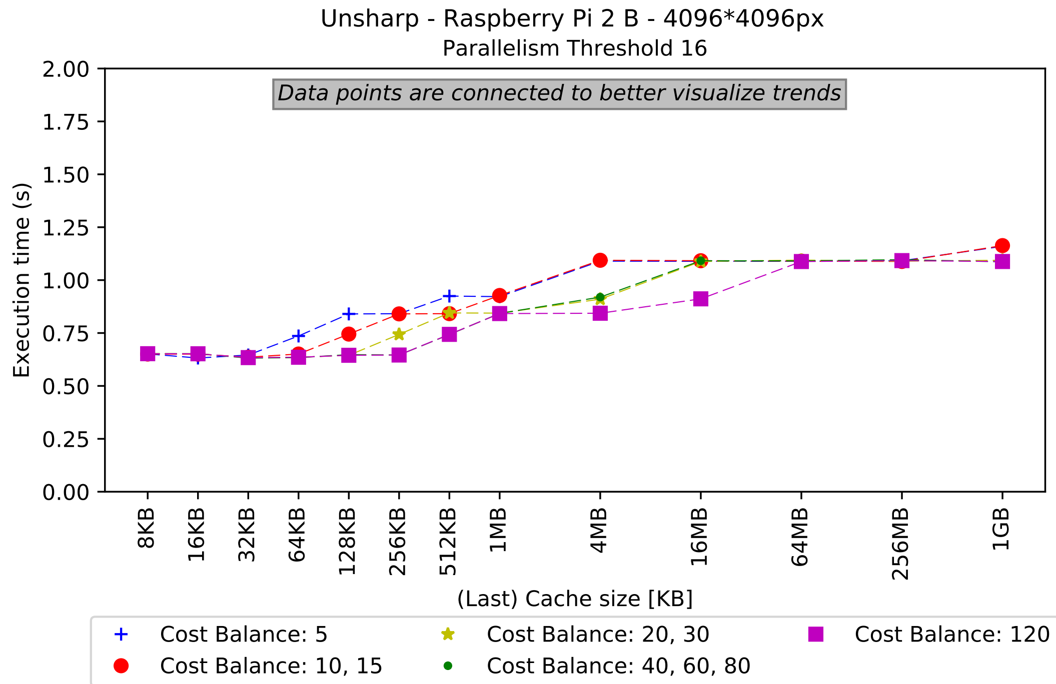

Figure 5.6.: Unsharp benchmark on the Raspberry Pi 2 B for varying cache sizes

The latter is not without overall impact though. Just like all the previous diagrams, especially Algorithm 5.3, showed, the *cost balance* mostly acts as an offset for memory load execution time increases. The larger the parameter value is chosen, the bigger the range of maximum *cache size* possible, before memory access related penalties start to appear.

Even though the *cost balance* behaves in predictable patterns for the memory bounded pipelines in this section, the same could not be said for computationally bounded ones. Especially in combination with a small *cache size* parameter, the behaviour was very erratic, as seen in Figure 5.4. It is therefore problematic to make conclusive statements about this particular parameter, whilst it was possible to do so for the other ones. These results will be summarized in the next, final chapter.

6. Conclusion and outlook

The goal of this bachelor thesis was to analyse the trade-off between locality, redundancy and parallelism, which has been applied to the field of image processing. This offered a basis of investigation, as various image processing pipelines exhibit different characteristics which impact the resource-aware optimization techniques possible. By further investigating into the Halide auto-scheduler, a recent development for automatic resource-aware optimization, the aim was to study the effects and relations between pipeline traits as well as parameters given. From the outcome of the conducted research it is possible to conclude that such correlations exist, which will be discussed in Section 6.1 Concluding, the final section intents to provide some avenues of further research offered by this thesis.

6.1. Summary

Based on the results, it can be concluded that the research into automatic resource-aware optimization by the Halide developer team has been largely successful. It has even been possible to confirm their claim that ARM-based devices show similar performance gains as their test platform did. Furthermore, the gap between the best possible configuration and the one simply using the hardware-platform specification has been significantly closed. This indicates that since the release of the auto-scheduler several improvements have been made to the schedule generation process.

Yet, the findings in this thesis reveal that some manual adjustments of the input parameters must still be performed to achieve the maximum performance possible depending on the pipelines' characteristics. Namely, memory bounded algorithms (such as Unsharp Mask), which feature dependencies and few calculations per stage, suffer from the inability to recalculate as needed. Therefore, the amount of possible parallelism is limited. Memory access penalties thus make up the bulk of the execution times, which can be optimized by reducing the *cache size* parameter for the auto-scheduler to the smallest level of cache capacity available. Pipelines that do not possess these traits are in return better off choosing the maximum cache size available per core.

Unfortunately, such statements can not be made in regards to the *cost balance* parameter, as there seems to be no discernible pattern in combination with realistic cache size choices, especially on non-memory bounded pipelines (Figure 5.3 and Figure 5.4 showcase this well). To find the maximum performance, currently there seems to be no way

around sampling a small subset of possible *cost balance* choices (for example: 5, 10, 20) by running the generated schedule. If a small loss of performance is acceptable, then the gathered data suggests that choosing an unrealistically large value for this parameter (e.g. 120 in our tests) still yields acceptable results in all cases. On top of not requiring sample executions, this additionally allows for a wider space of performant *cache size* parameter choices, leaving the programmer with a wide margin for possible errors - after all, the auto-scheduler's purpose is to automate the resource-aware optimization process, not complicate it.

6.2. Outlook

There are some points of discussion and further possible research which appeared during the course of this thesis.
Most prominently, the unexpected behaviour for low *parallelism threshold* parameter values, as explained in Section 5.3.1. Further investigation into this area could determine why the results differ from what the pseudo-code in the original auto-scheduler paper would suggest.
Another interesting avenue of analysis could be the - at this moment - unpredictable behaviour for several *cost balance* choices combined with small *cache size* parameters. This is mainly observable on non-memory bounded pipelines. A deep examination of the underlying auto-scheduler source code would be required to determine why substantially different, sub-optimal schedules are generated for some values of the *cost balance* parameter.

Appendix

A. Full benchmark results

The sourcecode of the benchmark, execution times and their resulting diagrams are available to download from:

`http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2017-glaeser-thesis-figures.zip`

Considering the amount of possible permutations between all configurations and ways of displaying them, the author opted to digitally publish these. The results include boxplot diagrams for each configuration per benchmark, as well as combined views with one parameter variation. Since the execution times have very little variance between their repetitions, they were averaged out and grouped together for presentation in this paper. Some of these plots have already been featured and the remaining ones can be obtained from the previous download.

B. Reviewed schedules

The following schedules were specifically reviewed during the evaluation of the benchmark results. Whilst a full inspection of all loop boundaries and calculation steps is also possible, a more condensed version is presented here.
The most important aspects about these schedules, in regards to characteristics discussed in Section 4.2, are the consumer-producer relationships. These are directly caused by dependencies in the original pipeline, as they indicate necessary pre-requirements for a following computation.
Furthermore, loop tiling which is directly affected by *cache size* can be observed here (for example in line 13), as well as utilized parallelization (e.g. line 3) and vectorization (e.g. line 10 & 15).

```
1  // Parallelism threshold: 4     Cache size: 32KB    Cost balance: 10
2  produce blur_y:
3      parallel v7.v7_o:
4          for v6.v6_o:
5              produce blur_x:
6                  for v8:
7                      for v7:
8                          for v6.v6_vo:
9                              vectorized v6.v6_vi in [0, 3]:
10                                 blur_x(...) = ...
11             consume blur_x:
12                 for v8:
13                     for v7.v7_i in [0, 7]:
14                         for v6.v6_i.v6_i_vo:
15                             vectorized v6.v6_i.v6_i_vi in [0, 3]:
16                                 blur_y(...) = ...
```

Schedule B.1: Auto-scheduled pipeline for blur (parallelism 4)

```
1  // Parallelism threshold: 16    Cache size: 32KB    Cost balance: 10
2  produce blur_y:
3      parallel v7.v7_o:
4          for v6.v6_o:
5              produce blur_x:
6                  for v8:
7                      for v7:
8                          for v6.v6_vo:
9                              vectorized v6.v6_vi in [0, 3]:
10                                 blur_x(...) = ...
11             consume blur_x:
12                 for v8:
13                     for v7.v7_i in [0, 7]:
14                         for v6.v6_i.v6_i_vo:
15                             vectorized v6.v6_i.v6_i_vi in [0, 3]:
16                                 blur_y(...) = ...
```

Schedule B.2: Auto-scheduled pipeline for blur (parallelism 16)

```
1  // Parallelism threshold: 16    Cache size: 512KB   Cost balance: 5
2  produce blur_y:
3      parallel v8:
4          parallel v7.v7_o:
5              for v6.v6_o:
6                  produce blur_x:
7                      for v8:
8                          for v7:
9                              for v6.v6_vo:
10                                 vectorized v6.v6_vi in [0, 3]:
11                                     blur_x(...) = ...
12                 consume blur_x:
13                     for v7.v7_i in [0, 63]:
14                         for v6.v6_i.v6_i_vo:
15                             vectorized v6.v6_i.v6_i_vi in [0, 3]:
16                                 blur_y(...) = ...
```

Schedule B.3: Auto-scheduled pipeline for blur (cost balance 5)

```
// Parallelism threshold: 16    Cache size: 512KB    Cost balance: 10
produce blur_y:
    parallel v7.v7_o:
        for v6.v6_o:
            produce blur_x:
                for v8:
                    for v7:
                        for v6.v6_vo:
                            vectorized v6.v6_vi in [0, 3]:
                                blur_x(...) = ...
            consume blur_x:
                for v8:
                    for v7.v7_i in [0, 63]:
                        for v6.v6_i.v6_i_vo:
                            vectorized v6.v6_i.v6_i_vi in [0, 3]:
                                blur_y(...) = ...
```

Schedule B.4: Auto-scheduled pipeline for blur (cost balance 10)

```
// Parallelism threshold: 16    Cache size: 512KB    Cost balance: 15
produce blur_y:
    parallel v8:
        parallel v7.v7_o:
            for v6.v6_o:
                produce blur_x:
                    for v8:
                        for v7:
                            for v6.v6_vo:
                                vectorized v6.v6_vi in [0, 3]:
                                    blur_x(...) = ...
                consume blur_x:
                    for v7.v7_i in [0, 31]:
                        for v6.v6_i.v6_i_vo:
                            vectorized v6.v6_i.v6_i_vi in [0, 3]:
                                blur_y(...) = ...
```

Schedule B.5: Auto-scheduled pipeline for blur (cost balance 15)

C. Extremely large execution time for first runs

During the review of the gathered results from the benchmark developed in this thesis, it was observable that the first execution of a auto-scheduled pipeline always took about 1 to 2 seconds longer than all of the remaining ones. This was very disconcerting, since the usual variation between runs never exceeded 1%, as explained in Section 5.1.
Further investigation revealed that this was caused by the way *pipeline.realize(...)* is implemented in *Halide*. It also acts as a short-cut which includes a call to *pipeline.jit_ compile(...)*. This compilation is responsible for compiling the Halide program into usable machine code and the jit (just-in-time) version outputs this directly into the host memory [Cf. 12]. By specifically implementing a call to *jit_ compile* before any performance measuring was conducted, the discrepancy between the first execution time and all the others was fully dissolved.

List of Figures

1.1. Rise of parallel computing. 1

2.1. Accesses with temporal locality 7
2.2. Accesses with spatial locality 7
2.3. Effects of reordering on loop traversal 8
2.4. Traversal without tiling 9
2.5. Traversal with tiling 9
2.6. Minimizing redundancy through stage-wise computation 12
2.7. Example throughput for increasing thread counts 15
2.8. Reuse of data with varying levels of parallelism 16

3.1. Throughput of Halide schedules for the Blur, Unsharp and Harris pipeline. 19

4.1. DAG visualization of the Blur pipeline 25
4.2. DAG visualization of the Unsharp pipeline 26
4.3. DAG visualization of the Harris pipeline 28

5.1. Throughput of the auto-scheduler on ARM-based devices 30
5.2. Effects on the auto-scheduler by low levels of parallelism threshold 32
5.3. Harris benchmark on the ODROID-X04 for varying cache sizes 33
5.4. Blur benchmark on the Raspberry Pi 2 B for varying cache sizes 34
5.5. Unsharp benchmark on the ODROID-XU4 for varying cache sizes 36
5.6. Unsharp benchmark on the Raspberry Pi 2 B for varying cache sizes . . . 37

List of Tables

4.1. Specification of the tested devices 22

List of Code Samples

2.1. Reordering traversal order in Halide 9
2.2. Tiled traversal in *Halide* 10
2.3. Fully inlined blurY calculation 12
2.4. Forcing full stage-wise schedule in Halide 13
2.5. Parallel, tiled traversal in Halide 14

List of Algorithms and Schedules

2.1. Sample implementation of 3x3 Blur Pipeline in Halide 6

4.1. Pseudo-Code of Configuration-Switcher 24
4.2. 3x3 Blur pipeline implementation with repeated boundaries 25
4.3. Unsharp pipeline implementation with repeated boundaries 26
4.4. Harris pipeline implementation with repeated boundaries 27

B.1. Auto-scheduled pipeline for blur (parallelism 4) 42
B.2. Auto-scheduled pipeline for blur (parallelism 16) 43
B.3. Auto-scheduled pipeline for blur (cost balance 5) 43
B.4. Auto-scheduled pipeline for blur (cost balance 10) 44
B.5. Auto-scheduled pipeline for blur (cost balance 15) 44

Bibliography

[1] S. G. Berg, "Cache prefetching", Department of Computer Science & Engineering University of Washington, Tech. Rep., 2002.

[2] P. J. Denning, "The locality principle", *Communications of the ACM*, vol. 48, no. 7, pp. 19–25, 2005.

[3] P. J. Denning and T. G. Lewis, "Exponential laws of computing growth", *Communications of the ACM*, vol. 60, no. 1, pp. 54–65, 2016.

[4] (Jan. 2017). Halide/readme.md, [Online]. Available: `https://github.com/halide/Halide/blob/master/README.md` (visited on 04/04/2017).

[5] C. Harris and M. Stephens, "A combined corner and edge detector.", in *Alvey vision conference*, vol. 15, 1988, pp. 147–151.

[6] J. Hruska. (Apr. 2015). Moore's law is dead, long live moore's law, ExtremeTech, [Online]. Available: `https://www.extremetech.com/extreme/203490-moores-law-is-dead-long-live-moores-law` (visited on 12/13/2016).

[7] C. Kanan and G. W. Cottrell, "Color-to-grayscale: Does the method matter in image recognition?", *PloS one*, vol. 7, no. 1, 2012.

[8] M. Kowarschik and C. Weiß, "An overview of cache optimization techniques and cache-aware numerical algorithms", in *Algorithms for Memory Hierarchies — Advanced Lectures, volume 2625 of Lecture Notes in Computer Science*, Springer, 2003, pp. 213–232.

[9] G. E. Moore, "Cramming more components onto integrated circuits", *Electronics*, vol. 38, no. 8, Apr. 1965.

[10] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines", *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, 2016, Proceedings of ACM SIGGRAPH 2016.

[11] National Research Council, *The Future of Computing Performance: Game Over or Next Level?*, S. H. Fuller and L. I. Millett, Eds. Washington, DC: The National Academies Press, Jan. 2011, ISBN: 978-0-309-15951-7.

[12] J. Ragan-Kelley. (Mar. 2014). Static vs. jit compilation, [Online]. Available: `https://github.com/halide/Halide/wiki/Static-vs.-JIT-compilation` (visited on 04/05/2017).

[13] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines", *ACM Transactions on Graphics (TOG)*, vol. 31, no. 4, Jul. 2012, In Proceedings of SIGGRAPH 2012.

[14] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines", *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[15] (Feb. 2015). Raspberry pi 2 - colated faq's, [Online]. Available: `http://jamesrandominfo.blogspot.de/2015/02/raspberry-pi-2-colated-faqs.html` (visited on 04/04/2017).

[16] (Jan. 2015). Samsung exynos 5422 - linux exynos, [Online]. Available: `http://linux-exynos.org/wiki/Samsung_Exynos_5422` (visited on 04/04/2017).

Eidesstattliche Versicherung

______________________________ ____________________

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

____

____

____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

__________________________ _______________________

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____________________________ _________________________

Ort, Datum Unterschrift