Automatic scheduling of image processing pipelines

Automatic Scheduling of Image Processing Pipelines

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op vrijdag 18 december 2020 om 16.00 uur

door

Savvas Sioutas

geboren te Athene, Griekenland

Dit proefschrift is goedgekeurd door de promotor en de samenstelling van de commissie is als volgt:

voorzitter:	prof.dr.ing. A.J.M. Pemen
1^e promotor:	prof.dr. H. Corporaal
2^e promotor:	prof.dr.ir. T. Basten
copromotor:	dr.ir. S. Stuijk
leden:	prof.dr. K.G.W. Goossens
	prof.dr.ir. H.E. Bal (Vrije Universiteit Amsterdam)
	dr. A. Adams (Adobe Research)
adviseurs:	dr. L.J.A.M. Somers

Het onderzoek dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Automatic Scheduling of Image Processing Pipelines

Savvas Sioutas

Doctorate committee:

prof.dr. H. Corporaal	Eindhoven University of Technology, 1^{st} promotor
prof.dr.ir. T. Basten	Eindhoven University of Technology, 2^{nd} promotor
dr.ir. S. Stuijk	Eindhoven University of Technology, copromotor
prof.dr.ing. A.J.M. Pemen	Eindhoven University of Technology, chairman
prof.dr. K.G.W. Goossens	Eindhoven University of Technology
prof.dr.ir. H.E. Bal	Vrije Universiteit Amsterdam
dr. A. Adams	Adobe Research
dr. L.J.A.M. Somers	Eindhoven University of Technology

This work was supported in part by Canon Production Printing Netherlands B.V.

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-5189-7

Summary

Automatic Scheduling of Image Processing Pipelines

Image processing applications are a vital part of many high-tech systems. Wide format printers, traffic surveillance cameras, as well as autonomous cars are examples of systems that execute complex image-processing algorithms that usually need to be highly optimized in an effort to meet real-time constraints. Due to the inherently intricate nature of these applications, as well as the ever increasing complexity of modern hardware architectures, code generation in the image processing domain remains a challenging task.

Even though modern state-of-the-art compilers attempt to automatically apply basic optimization techniques on the generated code, software developers with low-level knowledge and deep understanding of the underlying hardware are often still needed to ensure that the final implementation of the algorithm can achieve the required performance. Being a manual effort, this optimization process has to be repeated each time a new algorithm is developed or a new hardware platform is used.

This thesis tackles the aforementioned challenge by developing methods that enable automatic generation of efficient code in the image processing domain. First, an analytical model is proposed for the optimization of memory-bound kernels targeting multi-core CPU architectures [81]. The model leverages hardware prefetching behavior in order to exploit self-spatial and temporal reuse. Results show 40% higher performance on average compared to prior related work.

Second, an optimization algorithm that aims to maximize producer-consumer locality and reuse in multi-stage pipelines targeting multi-core CPU architectures is proposed through a set of heuristics and analytical modeling [82]. Experimental results show an average 40% performance improvement over previous automatic scheduling attempts, while being competitive to manually-tuned solutions.

Third, an auto-scheduling framework for GPGPU architectures that captures key platform specific parameters is introduced [79]. The auto-scheduler aims to maximize the amount of parallelism exploited in the implementation while minimizing the number of external memory accesses.

Throughout an extensive set of benchmarks and platforms the proposed framework is the first one to achieve performance competitive to manual expert-tuned solutions while limiting design time to the order of seconds.

Finally, a series of custom lowering compiler passes are designed that enable automatic, efficient code generation for the NVIDIA Tensor Core architecture through an abstract high-level description in the Halide domain - specific language [80]. Achieved performance is within 20% of the NVIDIA cuBLAS library kernels thus increasing the programmability of the NVIDIA Tensor Core units without severely limiting performance.

The optimization methods proposed in this thesis enable compilers to automatically generate high-performance code for applications in the image processing domain, without the need of manual optimizations or extensive auto-tuning. All of the above contributions have been implemented as tools or compiler passes to be used alongside the Halide domain specific language and compiler and are also available as open source software.

Contents

1 Introduction

1.1 Image processing applications

Digital signal and image processing applications can nowadays be found everywhere around us: from filtering and image enhancement techniques implemented by our smartphone cameras to autonomous driving technology and face detection or pattern recognition algorithms used by online social media and search engines. This is largely a result of the rapid advances in sensor technology which allows for an ever-growing, massive amount of data to be constantly available. The imaging and computer vision domains allow us to process and analyze that data and extract useful information. As a result, image processing is no longer strictly tied to computational photography but is widely used in many other domains of the digital world.

Throughout this thesis, we will be using the term "*image processing pipelines*" to describe all relevant applications in the image processing and computer vision domains. These "*pipelines*" refer to large graphs of feed-forward pixel computations such as stencil operations, reductions or computations with data-dependent access patterns over input images (or arrays of pixels). A single pipeline can often contain dozens of stages with various combinations of the above operations. As hardware capabilities and sensors continue to evolve, the opportunities for more sophisticated algorithms and applications increase. At the same time, as the complexity of these applications increases, so does the need for high-performance and

efficient code generation.

Artificial intelligence and machine learning is for example a domain that has rapidly evolved over the past decade due to the enormous amount of data that is nowadays constantly available. Deep learning neural network applications have proven to be capable of tackling many difficult problems in the computer vision domain. Image classification and localization, object detection and segmentation as well as super resolution and reconstruction of images are only some of the applications that neural networks excel at, while traditional imaging techniques struggle with or are completely incapable of solving those problems within the constraints of modern hardware specifications. However, these solutions often come at the cost of orders of magnitude higher computational needs and hardware resources than the equivalent classical methods. Deployment of such applications thus becomes a challenging problem especially in the context of embedded computing, where energy efficiency and performance are intertwined and data transmission towards offline processing is not always ideal, as it can be even more costly in terms of energy consumption.

1.2 Architectures and platforms

The specifications of the overall application often dictate the characteristics and nature of the deployment platform. Deep learning networks used for image super resolution and classification require an enormous amount of compute power and are therefore usually executed on highly-parallel GPU clusters. On the other hand, object recognition networks used in self-driving cars and image enhancement filters found on the entirety of modern smartphones require both high performance as well as low power either to ensure that the application fits specific real-time constraints, or to allow for prolonged battery usage, while in may cases the need for both performance and energy efficiency is also present.

As modern hardware architectures become more and more complex, deployment of applications becomes a critical and challenging task. State-of-the-art CPU architectures often contain multiple SIMD units in each of their respective processing elements, along with a highly sophisticated memory hierarchy system with various prefetching mechanisms. On the other hand, graphic processing units (GPUs) offer an enormous amount of parallelism that can be exploited in order to dramatically improve performance in applications where data-level parallelism is abundant. Highly-specialized architectures that aim to further speed up the main computation part of an application are also becoming more and more common. The NVIDIA Tensor Core units (TCUs) [58] and Google TPU [33] are such examples that offer specific instructions targeting generalized matrix multiplication kernels, which are the main workload of most deep learning

applications. These units can offer performance orders of magnitude higher than traditional CPU and GPU architectures (Figure 1.1).

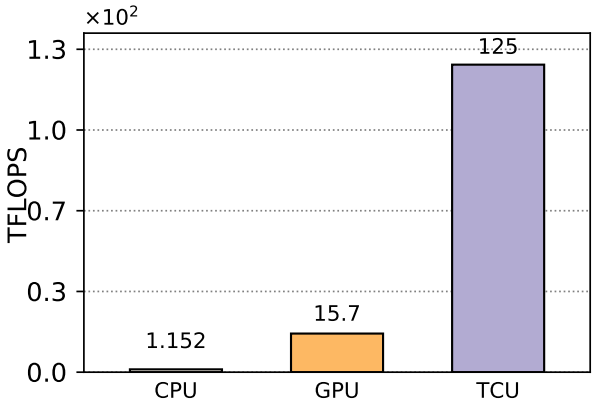


Figure 1.1: TFLOPS - theoretical peak performance comparison between CPU, GPU and TCU architectures: Intel i9-9900K, NVIDIA Tesla V100 (with / without TCU)

Current state-of-the-art hardware platforms usually contain both a CPU architecture which can act as a host, as well as a GPU (often with embedded Tensor Core units in case of NVIDIA), FPGA or ASIC architecture that can act as an accelerator. Developers thus need to decide on *where* and *how* to deploy their application on such a heterogeneous platform in order to properly utilize all of its resources. Answering this question is not as simple as choosing the hardware that can offer the highest peak performance, but instead requires extensive evaluation of the application's and the platform's specifications. For example, while a GPU can usually achieve performance orders of magnitude higher than a CPU in applications with high levels of data parallelism, it also has much higher power consumption. For mobile embedded applications such as health monitoring devices, prolonged high power usage can significantly limit the battery's lifetime, while it can also cause disastrous effects even on high-performance GPU clusters when specific thermal limits are being exceeded for a long period of time.

In order to maintain a balance and achieve both high performance, as well as energy efficiency, developers need to ensure that *all (and only) necessary hardware resources are used in an efficient way.* In a GPU architecture, that corresponds to all processing units being busy during execution while maintaining proper usage of the system's memory hierarchy. At the same time, fully exploiting the highly parallel nature of the platform can minimize the time during which the system operates at

full power. A similar argument can be made for CPUs as well as any other kind of hardware architecture. As a result, the need for efficient code generation and optimization is obvious and perhaps even more important when targeting embedded devices that feature a large number of constraints.

1.3 Platform-aware compilation

The initial design of an image processing algorithm is typically performed in a high-level language such as MATLAB or Python and has no relation to the platform the application will be deployed on. The final code then often has to be re-written in a highly efficient C/C++ implementation, which takes the underlying hardware and its resources into account during the optimization process. As different architectures require different optimization strategies to achieve sufficient performance, it is often the case that an optimized version of the same algorithm will look completely different depending on the platform that it will be deployed on.

Optimizing even simple applications requires an extensive number of code transformations to be applied in order to fully exploit all capabilities of the target platform. Vectorization through SIMD intrinsics, loop tiling and parallelization with OpenMP pragmas are only some of the optimizations commonly used when targeting a multi-core CPU architecture. In a similar fashion, executing the same application on an NVIDIA GPU would require rewriting all of it as a series of CUDA kernels and then applying a different set of optimizations and transformations. Hardware-specific parameters such as cache sizes, prefetching mechanisms, SIMD vector width and number of available registers and processing elements have a direct impact on *which combination of transformations will lead to optimal performance.*

As a result, even though some techniques such as tiling, unrolling, and loop fusion might be shared across architectures their usage and parameters such as tile sizes and unroll factors will again largely differ not just per platform but also between various models of the same architecture, depending on the above hardware specifications. Due to this reason, the final implementation of an algorithm will be completely different depending on whether the overall application runs on the high-performance CPU of a desktop computer or on the embedded graphics processing unit of a mobile smartphone, as well as on the manufacturer of said CPU and GPU: an Intel processor will require different optimization strategies and techniques compared to an ARM and even an altogether different programming model compared to an NVIDIA GPU.

Furthermore, as newer architectures become more and more *specialized, the programming effort required to achieve high performance in-*

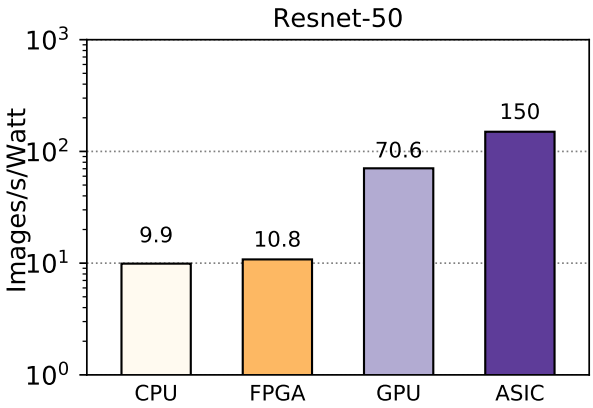


Figure 1.2: Comparison of computation efficiency (in images/s-Watt for network inference) for CPU, FPGA, GPU, and ASIC for deep learning [19]

creases dramatically. For example, consider the popular Resnet-50 deep learning network. Figure 1.2 shows a comparison between the computation efficiency (Number of images / s/Watt) for CPUs as well as FPGA, GPU and ASIC accelerators. As seen in the graph, both GPUs and ASICs achieve much higher efficiency than that of CPUs [19]. GPUs manage to achieve near-ASIC performance thanks to the specialized logic used by the embedded Tensor Core units. However, even though GPU programmability and flexibility is certainly increased compared to ASICs which are designed and fixed to a single application, significant effort by the programmer is still required in order to maintain an adequate implementation. Developers that aim to efficiently utilize all capabilities of such platforms have to extensively restructure and re-optimize their existing code-bases while dealing with data movement not just between the host CPU and accelerator-GPU devices but also the CUDA cores and Tensor cores.

Platform-aware code generation becomes even more important when one considers the fact that in an industrial context *the same application will have to be compiled multiple times* such that it is capable of running efficiently in various targets. Image processing frameworks such as Adobe Photoshop need to run adequately on systems with various hardware capabilities. As a result, the deployed application often contains multiple optimized implementations of the same filtering or enhancement algorithm, one of which gets finally executed depending on the client's platform. In the same way, other camera and image enhancement applications such as Instagram found in almost all current smart-phone devices need to provide high-performance implementations even though each client's device might

differ not just per manufacturer, but also per CPU type, GPU capabilities, or even battery and thermal limitations. This trend emphasizes the need for novel compilation and optimization techniques that enable efficient code generation for applications running on multiple possible hardware platforms.

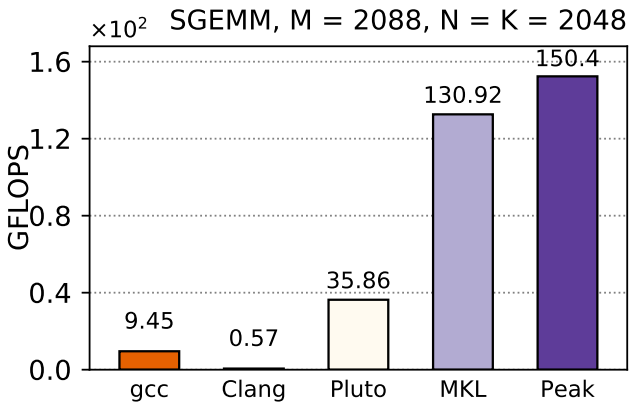


Figure 1.3: SGEMM single-core performance comparison across various compiler frameworks and Intel-MKL, Intel i7-8700K [11]

While modern compilers attempt to automatically apply some basic optimizations such as vectorization and unrolling to the generated code, other more sophisticated transformations fall outside their scope and they are often unable to achieve performance competitive to hand-tuned manual implementations. As a result, the translation-optimization phase remains a manual process that has to be repeated each time a newer or different platform is used, even when the initial algorithm is left unchanged. Figure 1.3 shows a single-core performance comparison across various compiler frameworks and the Intel-MKL library on a single-precision matrix-multiplication benchmark targeting an Intel i7-8700K processor. As seen in the graph, both gcc (v9.2.1) and Clang (v8.0.0), the current most popular general purpose C/C++ compiler frameworks, are unable to achieve performance even within 10% of the machine's capabilities (column Peak), even though both attempt to use optimizations such as vectorization and unrolling [1]. On the other hand, Pluto [12], an open-source polyhedral source-to-source compiler that can analyze the code before performing various polyhedral transformations (including interchange and tiling) achieves much higher performance. However, all

[1] Clang performs poorly on the above benchmark due to the fact that it only attempts to vectorize the innermost loop, while interchange would instead dramatically improve spatial locality [11]

three frameworks are unable to match the high efficiency of the manually optimized micro-kernels found within Intel-MKL [31].

Various domain-specific languages (DSLs) and compilers have been proposed over the years [53,73,87] in order to accelerate the design process without sacrificing performance and code readability. They typically make use of domain-specific knowledge to restrict the space of allowed programs, thus allowing them to apply transformations and optimizations that a general-purpose compiler could not easily do. Such languages often offer automatic schedulers which are integrated inside their compilers. These schedulers fall into two categories: analytical model-driven schedulers that generate quick solutions by restricting the design space considered [52, 72] and brute-force autotuning frameworks that extensively search the optimization space through iterative compilation [4,54]. Hybrid solutions have also been proposed over the years that attempt to limit the space evaluated by the autotuning frameworks and steer it without sacrificing performance of the final implementation [2]. In this thesis we will be focusing on the Halide DSL and compiler [73].

The Halide DSL [73] is perhaps the most prominent among the aforementioned image-processing DSLs. Its syntax allows developers to specify their implementations using a two-step approach: a high-level, functional description of their algorithm and a separate set of scheduling directives which dictate the various transformations and optimizations that should be applied on the final implementation. As a result, it enables faster exploration of the optimization space, improved code readability and maintainability. Furthermore, through various back-end code generators it can improve code portability, making it easier to port an existing algorithm to a new target platform. However, even with Halide or other similar languages, *expert knowledge is still required to ensure that the optimization schedule applied on the implementation will lead to near-optimum performance.* While there have been attempts at automatic schedule generation [43, 52], these are generally unable to consistently reach the performance achieved by manually tuned schedules.

1.4 Optimization challenges

As already mentioned, the optimization strategies that should be applied on the implementation are associated with both the nature of the stages that the pipeline consists of, as well as the architecture-specific parameters and constraints of the target hardware platform. Traditional optimizations attempts to focus on the vectorization and parallelization of the loop nests associated with the definition of each stage. However, most operations found inside inner loop computations feature low arithmetic intensity, and instead focus on extensive data transfer requirements which

grow as the dependencies between the stages become more complex. As a result, the majority of image processing applications are *memory bound* and their performance is *limited by the memory bandwidth of the underlying platform.*

Due to the above reason, global transformations such as loop/stage fusion and tiling are also necessary. Stage fusion in combination with tiling and inlining can dramatically increase producer/consumer locality and reduce memory usage at the cost of redundant computations and synchronization. The number of stages that can be merged together into a single loop nest is strongly correlated with the hardware resources and nature of the platform. For example, a CPU architecture contains a much larger cache compared to the shared memory found in most GPUs and thus allows for more computations to be stored into local memories. On the other hand, GPU architectures offer massive parallelism that can hide the latency caused by recomputing data.

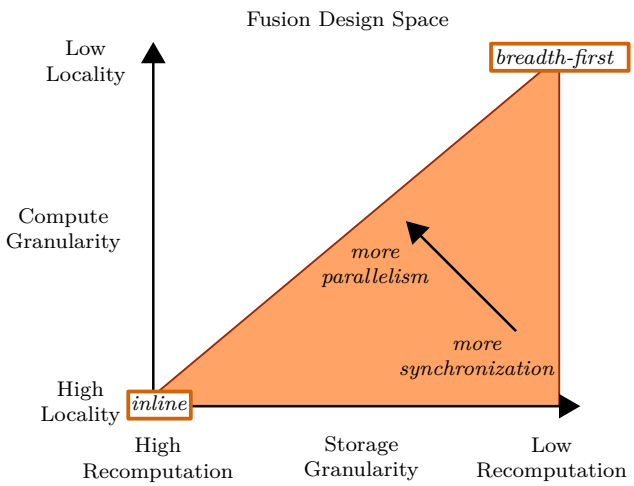


Figure 1.4: Optimization design space, adapted from [73]

Optimizing an image processing pipeline involves dealing with a *trade-off between parallelism, redundant computations and locality.* A visual representation of the design space is shown in Figure 1.4, where the vertical axis corresponds to the compute granularity while the horizontal axis corresponds to the storage granularity. Storage granularity refers to the amount of intermediate values that are stored while compute granularity refers to the time interval between production and consumption of said values. The various transformations that are applied on the generated code have a direct impact on both the parallelism that can be exploited, as well as the locality of intermediate results between stages. Finding

a point in this trade-off space which results in maximum performance is not a trivial task, especially considering the near infinite number of valid options as well as the fact that different hardware architectures will have their performance influenced by recomputation and parallelism in a different way. This figure will be explained in further detail in the following chapters.

Design space exploration is usually performed manually by domain experts with instruction-level knowledge and deep understanding of computer architecture. Manual optimization (or scheduling) is a very time-consuming process but usually offers the best results in terms of performance. Alternatively, various autotuning frameworks have been evaluated over the years which focus on iterative compilation techniques that exhaustively search the optimization space in order to achieve performance close to the maximum that can be achieved. Such frameworks can offer results that may even surpass manual optimization when targeting small-scale applications such as matrix multiplication and other linear algebra kernels, where the optimization design space is often limited to combinations of tiling, unrolling and parallelization/vectorization. However, their efficiency decreases as the complexity of the application rises. In the context of multi-stage imaging pipelines where more transformations are needed to achieve optimal performance, the search space quickly becomes too large to traverse within a specific time frame. Autotuners then need to constrain themselves to only a subset of the whole space or use a model as a steering mechanism in order to converge to a good solution. More importantly, *cross-compilation falls out of their scope* and they are unable to provide a solution when a single algorithm needs to be compiled for multiple targets, since they need to perform iterative execution of the application on the actual platform in order to choose a final optimal implementation. As a result, depending on the nature of the platform and the complexity of the algorithm, neither manual scheduling nor brute-force autotuning is always feasible, especially when targeting embedded edge devices with limited resources.

On the other hand, analytical modeling guided by heuristics is a quick and efficient way of generating optimization solutions. Interestingly, in the scope of linear algebra applications (such as Basic Linear Algebra Subprograms - or BLAS routines), they have even proved to be on-par with autotuned solutions [44], without the need for iterative compilation. Such analytical frameworks aim to estimate the performance and minimize the execution time of the application through carefully designed cost functions and heuristics that model the behavior of the target platform.

In order to eliminate both the need of time-consuming autotuning and extensive manual effort, the problem we aim to solve lies in implementing efficient optimization strategies and analytical models for applications in the image-processing domain that *integrally look at both architecture- and*

application-specific parameters in order to decide which transformations should be applied on the generated code. Design space exploration should therefore be driven by *a trade-off analysis that strives to maintain a balance between redundant computations, parallelism and locality.*

1.5 Thesis contributions

As already mentioned, the majority of image processing applications are dominated by memory-bound kernels and operations. To this end, modern CPU architectures employ various levels of memory hierarchy along with sophisticated prefetching mechanisms. Since complete image processing pipelines may feature hundreds such kernels as functional stages, proper utilization of the above components is vital during the optimization process. Moreover, when considering each pipeline as a whole, a new set of optimization opportunities become available due to data reuse opportunities across stages of the pipeline that feature a producer/consumer relation.

Equivalent techniques need to be developed for GPGPU architectures that accelerate highly parallel applications in heterogeneous platforms. Current state-of-the-art GPUs contain specialized units (such as the NVIDIA Tensor Core Units, present in all modern NVIDIA GPUs) capable of significantly increasing the performance of specific workloads such as matrix multiplications. Modern compilers and imaging DSLs need to integrate novel compiler passes which can take advantage of these units without sacrificing performance or programmability.

In order to tackle the main issues highlighted in the previous sections and achieve the approach outlined above, this thesis introduces a set of analytical models and heuristics that automatically schedule image processing applications on both CPU and GPU architectures without the need of expert low-level knowledge or extensive manual effort. All algorithms and frameworks presented throughout this thesis have been implemented as tools to be used alongside the Halide DSL and are available as open source software.

We show that our frameworks achieve performance competitive to manually tuned schedules while generating solutions in the order of seconds. To this end, our contributions can be summarized as follows:

1. An analytical model that optimizes memory-bound kernels targeting multi-core CPU architectures by modeling the behavior of the memory hierarchy and its various prefetching mechanisms [81].

2. An optimization algorithm composed of an analytical model and a set of heuristics that aim to maximize producer-consumer locality and data reuse in multi-stage image processing pipelines [82].

3. An automatic scheduling framework that targets GPU architectures and enables quick generation of optimization schedules while extending the search space considered in traditional state-of-the-art loop and kernel fusion techniques [79].

4. A series of novel compiler passes that facilitate the programming of NVIDIA tensor core units by enabling code generation through the Halide DSL and achieves performance competitive to cuBLAS implementations [80].

This work shows that through analytical modeling and heuristics, we can enable quick and efficient code generation in the image processing domain without the need of time-consuming autotuning, even for large and complex image processing applications.

1.6 Thesis overview

The rest of this thesis is organized as follows. Chapters 2 and 3 present necessary background information in order to facilitate the understanding of the next chapters, while Chapters 4-7 present our main contributions.

In detail, Chapter 2 provides details regarding application specific parameters in image processing pipelines, while Chapter 3 introduces Halide-specific knowledge necessary for the following chapters.

Chapter 4 presents the analytical model that classifies memory-bound kernels and enables optimization that target either temporal or self-spatial locality. Chapter 5 showcases the optimization algorithm that exploits producer-consumer locality and reuse in order to maximize performance in various image processing pipelines. Chapter 6 introduces an automatic scheduling framework for GPU architectures that captures key platform-specific parameters and achieves performance competitive to manual solutions. Chapter 7 presents a series of custom lowering compiler passes that enable automatic, efficient code generation for the NVIDIA Tensor Core architecture through the Halide DSL.

Finally, Chapter 8 concludes the thesis and discusses future-work possibilities. The procedure in order to gain access to the developed tools can be found in Appendix A.

2

Image processing pipelines

2.1 Algorithms

Algorithms in the image processing domain are typically defined as series of feed-forward pixel computations. The entirety of these operations along with all dependencies between them composes the image processing pipeline. In a similar way, each independent operation over an array of pixels defines the individual stage of the pipeline. Image processing pipelines can then be described as directed acyclic graphs (DAGs) where each node corresponds to a functional stage of the pipeline. Each stage performs a specific operation on one or more multi-dimensional input buffers before passing its output onto the next stage. Computations on input buffers are usually described using loop nests, the depth of which is defined by the number of dimensions required for the output buffer of each stage. Stencil computations usually make up the majority of such stages in a typical state-of-the-art image processing pipeline.

Stencil codes are iterative kernels that compute elements/pixels of an array based on a fixed pattern, which is called a '*stencil*'. They are common in simulations (fluid dynamic simulations), iterative differential equation solvers, cellular automata and of course image processing applications [29].

Other common operations found within the image processing and computer vision domains refer to up/down sampling, reductions and computations with data-dependent access patterns. Each of the above operations

requires individual analysis and separate optimization techniques in order to achieve high performance.

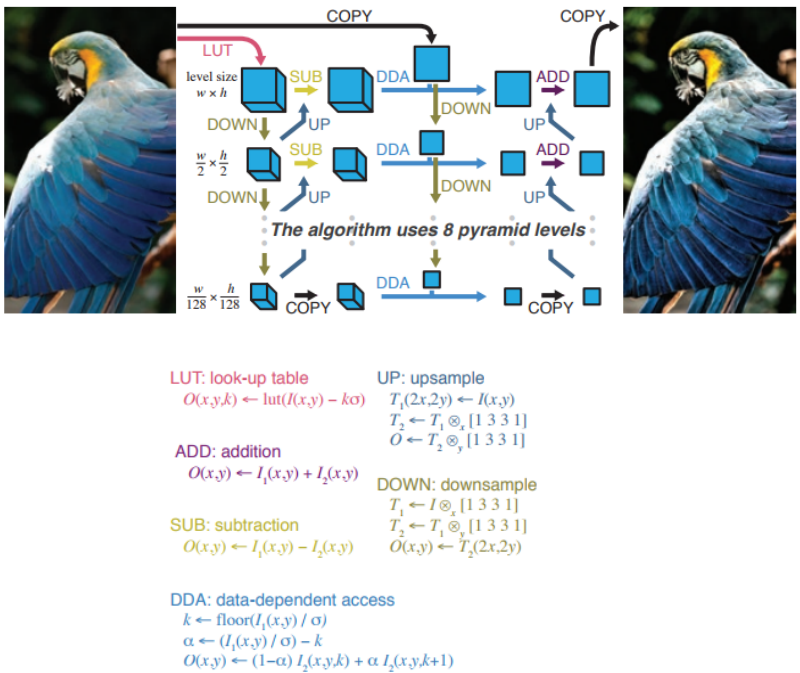


Figure 2.1: Local Laplacian Filters [73]

The dependencies between stages can be described through a producer/consumer relation. Stages that produce data which are needed in followup computations are called producers, while the ones that require that data as inputs are referred to as consuming stages. Analyzing these relations is a crucial part of the optimization process.

Common state-of-the-art pipelines often contain multiple combinations of the above operations, often across dozens of stages. Figure 2.1 shows the "local Laplacian filters" pipeline [62], used in Instagram. It is composed of 99 stages with complex dependencies between them and various combinations of the above mentioned operations. Manually optimizing such a pipeline can require months of work even for domain experts.

2.2 Optimization strategies

Implementations of the various algorithms in the image processing domain often need to be highly optimized in order to meet real-time constraints. An optimized implementation can be orders of magnitude faster than a naive version of the same algorithm. As a result, developers often employ various techniques to transform their code depending on the characteristics of both the overall application, as well as the architecture-specific parameters of the platform.

These optimizations usually refer to loop transformations that attempt to increase the performance of the implementation either by reducing the number of external memory accesses or by increasing the amount of parallelism exploited in the generated code. The transformations that we will be using in the following chapters are briefly introduced below:

Loop splitting. This transformation attempts to eliminate dependencies between loop iterations by splitting the original loop nests into multiple ones (usually inner and outer). The new loops iterate over different subranges of the original iteration space but contain the same body. Loop splitting can enable other optimizations such as tiling, vectorization and parallelization [48].

```
1 for i=0;i<Bi;i++
2  for j=0;j<Bj;j++
3   B[i][j]=A[j][i]
```

```
1 for ii=0;ii<Bi;ii+=Ti
2  for i=ii;i<ii+Ti;i++
3   for j=0;j<Bj;j++
4    B[i][j]=A[j][i]
```

where $B_i = k \times T_i, k \in \mathbb{N}$

Listing 2.1: Loop Splitting

An example of Loop Splitting can be seen in Listing 2.1, where the initial `i` loop on the left side has been split into an outer `ii` and an inner `i` loop by a factor equal to `Ti`. As is, this transformation does not influence the access pattern or computation on either array (neither on A nor B). However, when combined with the following transformations it can significantly improve spatial and temporal locality.

Loop interchange [48] reorders the dimensions of a loop nest in order to alter the memory access patterns on an associated array. Loop interchange can reduce cache misses and improve locality.

An example can be seen in Figure 2.2. The original loop nest that performs a simple array transposition is seen on the left. By interchanging the `i` and `j` loop indices (right side nest), the memory accesses on array `A` can

```
1 for i=0;i<Bi;i++
2  for j=0;j<Bj;j++
3   B[i][j]=A[j][i]
```

```
1 for j=0;j<Bj;j++
2  for i=0;i<Bi;i++
3   B[i][j]=A[j][i]
```

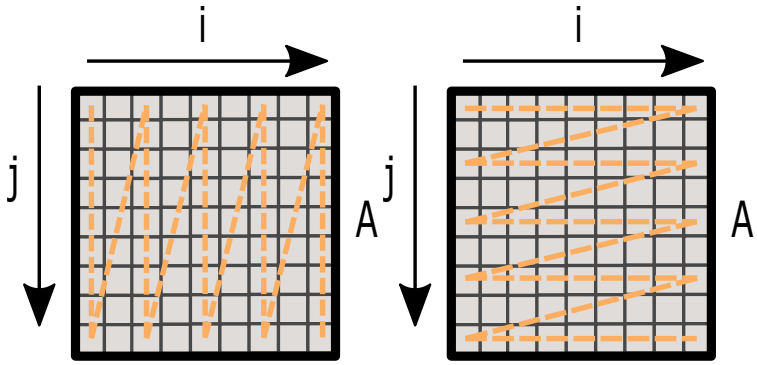


Figure 2.2: Loop Interchange

become more cache-friendly. A visual representation of the transformation is shown in the above figure. The left subfigure shows the access patterns on array A before interchange is applied. Input pixels are loaded in a column-major order, causing a different cache line to be fetched from the external memory into the local cache for each output pixel (since data is stored in a row-major layout in modern C-like compilers). Depending on the size of the global `j` dimension, these lines might be evicted before `i` moves to the next iteration and data from the same cache line are needed again. As a result, this loop ordering will cause a significant number of cache misses. On the other hand, the right subfigure shows the access patterns on the same array after interchange has been applied. In this case, since `i` is the innermost loop, data will be loaded *across* cache lines, which in turn will result in a dramatically reduced number of misses.

Loop tiling is a combination of interchange and splitting. Tiling [18,40] transforms a loop such that it iterates over blocks of data with carefully picked sizes that fit in the local memories of the platform (cache or shared memory of a GPU).

An example of loop tiling can be seen in Figure 2.3, where both `i` and `j` loops have been tiled. Splitting is first applied such that the initial `i` loop is split into `i` (inner) and `ii` (outer) (and `j` and `jj` for the original `j` dimension in a similar fashion) and then interchange is performed such that the new inner loops are the `j` and `i` loops while the outer ones become the `jj` and `ii`. The inner loops are called "*intra-tile*" loops, while the outer ones are referred to as "*inter-tile*". The above figure, which

```
for i=0;i<Bi;i++
 for j=0;j<Bj;j++
  B[i][j]=A[j][i]
```

```
for ii=0;ii<Bi;ii+=Ti
 for jj=0;jj<Bj;jj+=Tj
  for i=ii;i<ii+Ti;i++
   for j=jj;j<jj+Tj;j++
    B[i][j]=A[j][i]
```

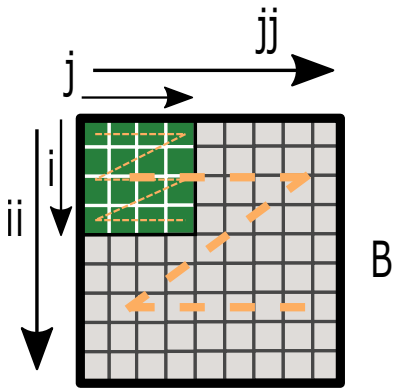


Figure 2.3: Loop Tiling

assumes tile sizes of 4 pixels (`Ti,Tj`=4) shows the order in which pixels of the output array `B` are computed. Pixels inside each tile are generated in a row-major order and after a whole tile is produced, computation moves to the next one, again in row-major order. Due to the massive amount of data needed to be transferred between stages, most applications in the image processing domain are memory bound, where performance of the final implementation is bound by the memory bandwidth of the underlying system. As a result, tiling is one of the most important transformations when optimizing imaging and vision pipelines.

Loop fusion combines the bodies of two separate loop nests. In the context of image processing, fusion also refers to the merging of separate stages that have a production/consumption relationship.

This transformation can significantly increase locality [97] between production and consumption of intermediate values but may increase the amount of redundant computations and synchronization requirements. Fusion can potentially also increase the complexity of dependencies between loop iterations and therefore limit the amount of parallelism that can be exploited. In its extreme case, total fusion is equivalent to stage inlining, where the producing statement is concatenated inside its consumers.

Examples of loop fusion are shown in Figures 2.4 and 2.5. The top left

```
allocate B[Bi][Bj];
for i=0;i<Bi;i++
 for j=0;j<Bj;j++
  B[i][j]=A[j][i]

for i=0;i<Bi;i++
 for j=0;j<Bj;j++
  C[i][j]=B[i][j]+D[i][j]
```

```
for i=0;i<Bi;i++
 allocate B[Bj]
 for j=0;j<Bj;j++
  B[j]=A[j][i]
 for j=0;j<Bj;j++
  C[i][j]=B[j]+D[i][j]
```

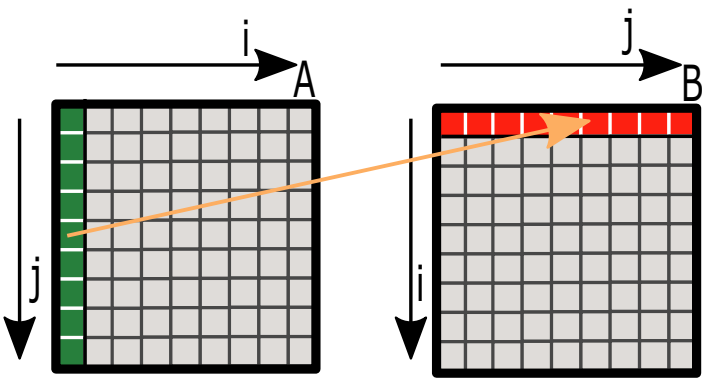


Figure 2.4: Loop Fusion - compute per output line

listing of Figure 2.4 shows two loop nests which can be described through a producer/consumer relation, as values of `B` are consumed to produce `C`. The right listing shows the transformed loop after the `i` loops have been fused/merged together. This allows for increased locality and reduced memory requirements (one scanline of size `Bj` of array `B` as opposed to a whole array of size `Bi*Bj` in the original loop nest). Figure 2.4 shows the dependencies between input array `A` and output `C` as well as how pixels are loaded from array `A` in order to produce and store the intermediate array `B` and finally the output `C`.

Two alternate options can be seen in the listings of Figure 2.5. The left one shows the transformed loop nest after both `i` and `j` loops of the producer `B` have been fused with its consumer `C`. Since pixels of `B` are computed as needed per output pixel of array `C`, only one element needs to be computed and stored before consumption. This further increases locality and reduces memory requirements compared to the previous case where only the `i` loops were merged. Figure 2.5 also shows the dependencies on array `A` for the production of a single pixel of the output.

Finally, the right listing shows an example of the most extreme case of loop fusion, which we call "*inlining*". In this case, computation of the intermediate array `A` has been completely inlined inside the compu-

```
for j=0;i<Bi;i++
 for i=0;j<Bj;j++
  allocate B[1]
  B[0]=A[j][i]
  C[i][j]=B[0]+D[i][j]
```

```
for j=0;i<Bi;i++
 for i=0;j<Bj;j++
  C[i][j]=A[j][i]+D[i][j]
```

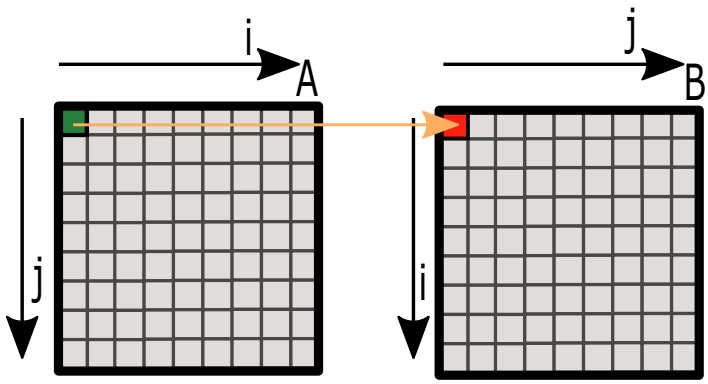


Figure 2.5: Loop Fusion - inline

tation of the output array, eliminating all intermediate allocations in the process. However, lower memory requirements are not always associated with higher performance. This type of "*point-wise fusion*" can interfere with both parallelism and vectorization or the efficiency of prefetching mechanisms. Moreover, in cases where the dependencies between stages grow more complicated, it may instead cause severe recomputation and synchronization costs. The following chapters investigate all of these trade-offs in greater detail.

Loop unrolling Loop unrolling attempts to decrease the overhead with branches and computations associated with a loop's exit condition by duplicating its body multiple times [97]. When the loop extent (iteration count) is known at compile time it can be completely unrolled, eliminating all such overhead in the process. This transformation affects register usage and instruction cache pressure and thus excessive unrolling may instead lead to performance degradation.

```
for j=0;i<Bi;i++
 for j=0;j<4;j++
  B[i][j]=A[j][i]
```

```
for i=0;i<Bi;i++
 B[i][0]=A[0][i]
 B[i][1]=A[1][i]
 B[i][2]=A[2][i]
 B[i][3]=A[3][i]
```

Listing 2.2: Loop Unrolling - completely unrolling the `j` loop

Listing 2.2 shows an example of loop unrolling, where the extent (range) of loop `j` is known to be a constant value of 4 and is unrolled as seen in the right loop nest.

Vectorization Vectorization is a form of data-level parallelism and usually refers to transformations that allow multiple loop iterations to be simultaneously executed on a system with SIMD (single instruction multiple data) extensions. Examples of such extensions are the SSE and AVX vector operations found in Intel x86 CPUs and the NEON equivalent operations found in ARM processors. The vectorization factor is typically set to be equal to the natural "*vector width*" of the corresponding architecture.

```
for j=0;i<Bi;i++
 for j=0;j<Bj;j++
  B[i][j]=A[j][i]
```

```
for i=0;i<Bi;i++
 for j=0;j<Bj/8;j+=8;
  B[i][j.vector.0.8]=A[j.vector.0.8][i]
```

Listing 2.3: Loop Vectorization

Listing 2.3 shows an example of loop vectorization, where the `j` dimension has been vectorized by a factor of 8. As a consequence, `j` within the computation inside the loop has been replaced by vector operations of size 8.

Parallelization Parallelization refers to transformations that enable task-level (or more specifically thread-level) parallelism such that the implementation exploits the multi-core capabilities a platform. Single or multiple loop iterations are assigned to a separate processing unit (corresponding to a core or thread in a multi-core processor). It is typically implemented using OpenMP pragmas or similar parallel programming paradigms.

2.3 Optimization space

As explained in the previous section, an efficient implementation of an image processing application requires a series of loop transformations and optimizations. However, choosing which specific combinations of transformations should be used as well as which parameters (i.e., tile sizes, loop ordering) is not a trivial task, as optimizations interact with each other and affect the behavior of the implementation in various ways.

Moreover, image processing pipelines are often dominated by stages with stencil operations. Optimizing such operations involves dealing with a trade-off between locality, parallelism and redundant computations of intermediate values (Figure 1.4). As a result, the overall optimization design space is also defined by these metrics. In detail:

Locality refers to the time interval (or iteration interval) between production and consumption of intermediate values. In the case of image processing pipelines, locality can be increased through loop/stage fusion and inlining. Increased locality can reduce or even eliminate external memory accesses and intermediate allocations. However, in most cases it comes at the cost of increased redundant computations and synchronization costs (and therefore decreased parallelism).

Parallelism refers to the ratio of the platform's processing units that can be kept active throughout the implementation. Task-level parallelism is usually associated with the number of cores/threads on a multiprocessor system, while data-level parallelism refers to the SIMD units and their vector width. The amount of parallelism that can be exploited is affected by the tile sizes applied on the loop nest. Small tile sizes can impact the efficiency of vectorization, while large tile sizes might cause cores/processing units to remain idle.

Recomputation or mainly refers to output values of stages being recomputed. The cost of such redundant computations varies depending both on the computation under question as well as on the nature and resources of the underlying platform.

A visual representation of the design space is shown in Figure 1.4. The storage granularity (horizontal axis) refers to the number of intermediate values that are stored, and thus controls the amount of recomputation in the final implementation. In a similar fashion, the compute granularity (vertical axis) refers to the interval between production and consumption of shared intermediate values and is associated with achieved locality of said values. Invalid are those situations where less data is stored than

is being computed (storage granularity is lower than the compute one). Implementations with low or zero recomputation and low locality are breadth-first solutions (we call them "*root*" solutions) and can exploit a high amount of parallelism. On the other hand, higher locality can lead to significantly higher synchronization costs due to more complicated dependencies between loop iterations and thus cause a decrease the amount of parallelism that will be exploited. In the other extreme of breadth-first solutions lie the "*inline*" ones, where intermediate values are recomputed every time they are needed by their consumers. In practice, and as we will see in the following chapters, the best solutions are often "*overlapping tiles*" and "*sliding window*" implementations that lie in the middle of the design space. The former refers refer to overlapping regions of data, which cause redundant computations but feature high parallelism. Sliding windows refer to situations where intermediate pixels are produced when first needed and are stored in circular buffers until no longer used. Such implementations feature higher producer/consumer locality at the cost of increased synchronization.

2.4 Summary

This chapter examined the key features of applications in the image processing domain, along with traditional optimization methods and techniques. The optimization space considered in the following chapters was also established.

3

The Halide language and compiler

Halide [73] is a Domain Specific Language (DSL) and compiler for high-performance image processing applications. Its key feature is the separation of algorithmic description and optimization schedule in an effort to provide increased code portability, readability and maintainability. In this thesis, we use Halide to evaluate our proposed optimization strategies and algorithms.

3.1 Functional representation

As a functional language, algorithms specified in Halide resemble the mathematical relation/equation that formulates the relationship between input and output. As seen in example listing 3.1, there are no explicit loops. Instead, developers specify variables, or "Vars", which correspond to the dimensions in the output domain of a functional stage.

Stages in Halide are specified as "Funcs" and refer to multidimensional rectangular domains. Each stage can perform any of the operations described in the previous chapter. The initial definition of a Func is called the "pure" definition. Subsequently, update definitions of a stage can be defined as independent passes over the output of the previous 'pure' definition or update. This allows for a form of recursion on the algorithms.

In the example of Listing 3.1, the `producer` stage performs a simple stencil operation on a 3-dimensional input buffer. The three dimensions correspond to the width `x`, height `y` and depth (or channel) `c` dimensions

```
1  Var x, y, c;
2  //A simple stencil code
3  Func producer;
4  producer(x,y,c)= input(x-2,y,c) + input(x+1,y,c) + input(x+3,y,c)
5                  +input(x,y+1,c) + input(x,y-2,c) + input(x,y+3,c);

7  //A small reduction
8  Func consumer;
9  consumer(x,y)= producer(x,y,0) + producer(x,y,1) + producer(x,y,2);

11 //and a scaling of the output
12 consumer(x,y)= consumer(x,y) * 10;
```

Listing 3.1: A Halide algorithm example

of the input buffer/image respectively with the width being innermost (in terms of storage). Its output values are then used to produce the `consumer` based on the relation of line 9 (summation across channels). The definitions in lines 4 and 9 respectively are the "pure" definitions of stages `producer` and `consumer` respectively. Finally, an example update definition can be seen in line 12 as an independent pass over the pixels of the output of line 9.

Given an algorithmic description such as the one above, Halide uses interval analysis to derive the dependencies between production and consumption of values. For the above example, the compiler will automatically infer that in order to produce one output pixel/value of stage `consumer`, it first needs to compute three pixels of the `producer` stage, as well as 18 pixels loaded from `input`.

3.2 Scheduling

An algorithm description written in Halide has no notion of execution order, allocation or connection to the hardware the code will finally run on. The order in which pixels of a stage are computed, as well as the allocation sizes for its buffers are all specified through a separate schedule definition.

Schedules in Halide are defined through a series of " *scheduling directives*" that dictate the various transformations that the compiler should apply on the generated code. Most of these directives correspond to common optimizations used in the image processing domain, similar to the ones presented in Section 2.2.

As an example, consider the schedule seen in Listing 3.2. Loops `x` and `y` are split by a factor of 128 and 8 such that the inner loops become

```
Var xi("xi"), yi("yi"), xo("xo"), yo("yo");

consumer.compute_root()
        .split(x, xo, xi, 128)
        .split(y, yo, yi, 8)
        .reorder(xi, yi, xo, yo)
        .vectorize(xi)
        .parallel(yo);

consumer.update().split(x, xo, xi, 128)
                 .reorder(xi, xo, y)
                 .vectorize(xi)
                 .parallel(y);

producer.compute_at( consumer, xo )
        .reorder(c,x,y)
        .vectorize(x,8)
        .unroll(c);
```

Listing 3.2: Example CPU schedule

`xi` and `yi` and the outer ones `xo` and `yo`. The dimensions of the loop nest are then reordered with the new permutation being `xi, yi, xo, yo`, where `xi` is the innermost. These two transformations are equivalent to tiling the original loop nest with tile sizes 128 and 32 in the `x` and `y` dimensions. After performing loop interchange, the innermost intra-tile loop is vectorized, issuing the compiler to inject SIMD intrinsics and replace the iterations with vector operations and the outermost loop `yo` is parallelized across threads. Similarly, the update definition of `consumer` is scheduled such that the innermost dimension `xi` is vectorized. The producing stage is then computed as needed per tile of its consumer using the `compute_at` directive. In this case, each time `consumer` starts an iteration of `xo`, the compiler is asked to first calculate and store all pixels of `producer` that will be consumed during this iteration. Finally, its dimensions are reordered with the channel dimension being innermost, which is then unrolled.

The previous schedule assumes that the code will run on a multi-core CPU architecture with SIMD extensions. However, Halide also supports code generation for GPU architectures. An example schedule for the same pipeline which instead targets a GPU architecture is found in Listing 3.3.

The schedule of Listing 3.3 will map the implementation onto the GPU of the platform. Most steps are identical to the previous CPU implementation with the exception of the `gpu_threads` and `gpu_blocks` directives. `gpu_threads` tells the compiler which dimensions should be

```
Var xi("xi"), yi("yi"), xo("xo"), yo("yo");

consumer.compute_root()
        .split(x, xo, xi, 128)
        .split(y, yo, yi, 8)
        .reorder(xi, yi, xo, yo)
        .gpu_threads(xi, yi)
        .gpu_blocks(xo, yo);

consumer.update().split(x, xo, xi, 128)
                 .split(y, yo, yi, 8)
                 .reorder(xi, yi, xo, yo)
                 .gpu_threads(xi, yi)
                 .gpu_blocks(xo, yo);

producer.compute_at(consumer, xo)
        .reorder(c,x,y)
        .gpu_threads(x,y)
        .unroll(c);
```

Listing 3.3: Example GPU schedule

assigned as CUDA threads (or OpenCL's work-items), while `gpu_blocks` indicates which dimensions should be mapped as blocks (or OpenCL's notion of work-groups). As a result, production of values from `producer` will happen before each 'block' iteration of the consumer.

Below is a list of the most important scheduling directives, which will be used in the following chapters.

`reorder`: Performs interchange on the dimensions of a loop.

`split`: Splits a dimension into an inner and outer loop.

`tile`: Splits and reorders dimensions in one statement, i.e. equivalent to loop tiling

`vectorize`: Translates a loop into vector operations.

`unroll`: Unrolls a loop by replicating its body.

`parallel`: Parallelizes each iteration of a loop across threads.

`compute_root`: Specifies that the whole stage should be computed and stored before its consumption

`compute_at`: Specifies that a stage should be computed "as needed" by a specific dimension of its consumer.

`store_at`: Specifies that the allocation of a stage should be moved at a specific dimension in the consumer's loop nest. Can control reuse of intermediate values and exploit sliding window optimizations.

`compute_inline`: Inlines the computation of a stage into all of its consumers.

`gpu_threads`: Assigns a dimension to correspond with CUDA's notion of threads.

`gpu_blocks`: Assigns a dimension to correspond with CUDA's notion of blocks.

`gpu_tile`: Equivalent to the tile directive, but also assigns dimensions to threads and blocks.

3.3 Compilation flow and code generation

Given an algorithm and an optimization schedule, the Halide compiler will internally lower the initial code into an Intermediate Representation (IR). Halide can target numerous back-ends including traditional CPU architectures (Intel x86, ARM) as well as GPU architectures (NVIDIA PTX, Intel OpenCL). After a series of lowering and optimization passes, the Halide IR is translated to the LLVM IR before the final code generation. LLVM [41] is a compiler infrastructure used by multiple general purpose as well as domain specific languages as a back-end code generator in order to support various instruction set architectures.

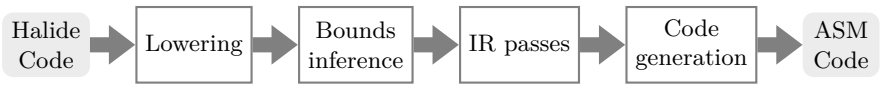


Figure 3.1: Basic compilation flow of Halide: Source code gets lowered into an intermediate representation (IR). Bounds inference determines the dependencies and bounds between consumers/producers. Various IR passes apply optimizations on the lowered AST (vectorization, scheduling directives, flattening etc). Final code generation is achieved by translating Halide IR to LLVM/NVVM IR and finally to machine instructions (ASM).

Figure 3.1 shows a simplified view of the compilation flow of the Halide compiler. Each of the individual steps are discussed below in more detail.

Lowering: The first step of the compilation flow is responsible for lowering the functional representation of the algorithm to an IR. The initial algorithm gets transformed into an imperative C-like loop nest

using a specific schedule. The storage and computation of producers are recursively injected inside the loop nest of the consumer as specified by the same schedule. All bounds and allocation sizes are symbolic at this point.

Bounds inference: Halide uses interval arithmetic to compute the regions required by each producer in order to calculate the values/pixels needed by its consumers. During this step the compiler uses interval computations to derive all bounds and allocation sizes, replacing the previous symbolic relations.

IR passes: A series of IR optimization passes are then used to apply various transformations on the lowered abstract syntax tree (AST). These passes are responsible for most of the optimizations used in the scheduling part of the code. Optimization passes involve: sliding and storage folding optimizations that remove redundant computations while replacing allocations with circular buffers, reusing intermediate data in the process. Storage flattening optimizations convert multidimensional buffers into one-dimensional ones by re-formulating array indices relative to the base of each buffer. Other passes focus on unrolling loops by duplicating their body, or vectorizing them by replacing iterations with vector operations.

Code generation: At this point, the Halide IR is translated into LLVM IR. LLVM is then used for back-end code generation, which emits machine instructions (ASM) for the scheduled pipeline. For NVIDIA architectures the equivalent NVVM IR is used before producing PTX code.

3.4 Summary

This chapter briefly introduced all necessary background knowledge related to the Halide DSL and compiler in order to facilitate the understanding of the following chapters. In this thesis, we use Halide to implement and evaluate our scheduling algorithms targeting both multi-core CPU architectures as well as GPGPUs. The first contributions of the thesis will focus on the front-end scheduling directives which operate on the Halide lowered IR, while the last one on the code generation part of the compiler and the various extensions that were made upon it.

4

Scheduling memory-bound kernels in multi-core CPU platforms

4.1 Introduction

The ever-growing gap between processor and memory speed in modern architectures is currently a severe drawback in the efficiency of applications in domains where high performance is necessary. Memory-intensive (or memory-bound) applications are affected the most by this problem, since they usually contain loop nests with a large number of memory accesses and relatively few computations. As a result, they are bound by the memory bandwidth of the system [102].

Developers often employ various optimization methods and techniques in order to mitigate the effects of this memory bandwidth problem and increase the performance of their implementations. Loop tiling [1,9,13,18, 40,46,83,84,96,100] is a common loop transformation that aims to improve temporal locality thereby enabling data reuse. Tiling paired with vectorization and parallelization can have a huge impact on the performance of an application. However, picking the proper tile dimensions that will minimize external memory accesses without interfering with the SIMD unit or the hardware prefetching mechanism present in most modern architectures is not a trivial task. Due to these reasons, manually optimizing a target algorithm is a time-consuming and error-prone process, where numerous architecture and application-specific parameters need to be considered.

In the past, most approaches to automatic tile size selection have

mainly focused on analytical models [8, 45, 50, 64] that only consider loop nests that fit into specific patterns while relying on the compiler to decide on the optimal loop ordering. These methods may quickly generate efficient code when the loop nest fits into the expected pattern but may produce sub-optimal results in other case. Furthermore, they usually ignore the hardware prefetching mechanisms found in modern architectures and as a result, the proposed optimizations may actually lead to a deterioration in the performance of the final implementation.

Other approaches employ dynamic auto-tuning frameworks [5, 93] that exhaustively search the optimization space in order to optimize the target application. In general, these frameworks are able to produce near-optimal results. However the time needed to converge to that solution is usually unknown, thus making them inadequate for fast design space exploration and debugging. Furthermore, the fact that they need to run on the target platform can also be a limitation for some architectures.

In this work, we propose an optimization algorithm and analytical model for memory-bound applications that aims to minimize external memory accesses, while taking necessary architecture and application-specific parameters into account. The model first classifies the application by detecting patterns in the definitions which are derived by the statements in the innermost level of the loop nest. We use these patterns to determine whether the applications should be optimized with emphasis on spatial or temporal locality in order to better exploit the hardware prefetching mechanisms, as well as to determine which other optimizations (i.e. vectorization, non-temporal instructions, multi-threading) may improve the performance of the final implementation. The algorithm then invokes an analytical model that based on the previous classification decides which levels of the cache hierarchy to optimize for and then chooses the tile dimensions as well as the final loop nest order.

We implement our algorithm as a tool to be used along with the Halide language and compiler [73] (introduced in Chapter 3 in order to automatically generate optimization schedules for Halide functions often within milliseconds. We extend the Halide compiler with the ability to generate non-temporal stores by adding a new scheduling directive to the language's front end. We test our method on various target applications and compare its results with previous analytical as well as dynamic empirical (auto-tuning) models. We find that our method achieves an average performance improvement of 40% compared to the aforementioned analytical models targeting the Halide DSL. Performance is also better in terms of quality to the exhaustively auto-tuned implementations, where the results using our approach are usually achieved in a matter of milliseconds instead of hours (in terms of optimization run-time) when using the Halide autotuner.

The remainder of this chapter is organized as follows: subsection 4.2 discusses related work. subsection 4.3 presents the proposed model and

analysis technique. subsection 4.4 shows the implemented Halide tool, while subsection 4.5 demonstrates experimental results and a comparison with similar frameworks. Concluding remarks are discussed in subsection 4.6. This work was published in [81].

4.2 Related work

The problem of optimizing memory-intensive applications has been considered many times in the past. Most of that work has focused on tile size selection algorithms. These algorithms usually employ analytical models that aim to determine the optimal tile dimensions in order to exploit temporal locality. The authors in [18] take cache parameters into account when generating tile sizes, but are only considering one level of cache hierarchy and no interaction with other optimizations or cache associativity. In [63] the authors propose the block data layout as a solution to bypassing the issues associated with memory-intensive applications and also provide a corresponding analysis. However specific hardware and software support is needed in order take advantage of their approach, which limits the application scope. [101] proposes a combination between machine learning techniques and synthetic kernels to calculate the tile size for a specific class of applications, but limits their search to cubic tiles and only takes one level of cache hierarchy into account. The authors in [64] propose an analytical model to optimize nested loops with a combination of tiling and interchange. However, their work is focused on embedded accelerators and thus all interaction with the cache is ignored, leading to suboptimal results of the model in cache-based systems.

In [49] the authors consider multi-level cache hierarchies in order to exploit reuse in both L1 and L2 cache levels while taking associativity into account. We use a similar analysis for our temporal locality optimizer that exploits reuse in L1 and L2 cache, but extends it in order to also take the hardware prefetching mechanisms and multi-core aspects of current architectures into account and to generate the loop nest permutation that takes advantage of those features. In [50], interaction with the hardware prefetching mechanisms is considered in order to achieve reuse in the L3 cache. However, both techniques rely on the compiler in the back-end to find the optimal loop order before performing any analysis. To this end, we propose a combined approach that considers loop tiling and loop ordering at the same time. Furthermore, they only consider tiling for applications with some form of temporal locality, which may lead to suboptimal results in situations where tiling should be focused only on self-spatial reuse.

Other approaches have focused on empirical autotuning methods that exhaustively try to optimize an application [5, 21, 92, 93]. In [93] an example of such a method is presented, that generates an optimized BLAS

library for a target platform on the Pluto framework. However, such approaches usually require a large amount of time in order to explore the entire design space and converge to an efficient solution and furthermore cannot be used without access to the target architecture.

Hybrid methods have also been presented where both analytical models and exhaustive searches are used [23,38,70,78]. For example, in [78] an analysis is conducted to obtain bounds on the search space that should be explored. The authors consider data reuse in multiple levels of the cache hierarchy but ignore cache associativity.

As already mentioned in Chapter 3, Halide enables the separation of a target algorithm from its optimization schedule. Due to this reason, it is a good target environment for testing our optimization algorithm. Similar to our approach, the Halide Auto-Scheduler [52], attempts to automatically generate an optimization schedule for a given function by using a heuristic based optimization algorithm. However, the authors' approach focuses on finding the best loop fusion options in image processing pipelines with numerous stages and thus the cache and tiling analysis it employs is limited (considering only one level of cache hierarchy). This leads to suboptimal results in small memory intensive applications. Moreover, it uses the bounds inference information provided by the back-end compiler regarding memory accesses and footprint and is thus unable to discern patterns in the source code. The Halide autotuner implemented alongside the autotuning framework Opentuner [5], is another method that automatically generates optimization schedules by iteratively running an application using different optimization configurations. The autotuner needs a large amount of time to search the design space, while providing no guarantee regarding the quality of the final solution. Furthermore, part of the design space is sometimes actually excluded from the search space, and thus the framework may be incapable of finding the optimal solution altogether.

Other, more recent schedulers were proposed (after publication of this work in [81]) which also attempt to schedule Halide applications [2, 82]. Due to this reason, the evaluation section of this chapter only compares to solutions available at the time of publication.

4.3 Proposed method

In this section we present the general optimization flow of our proposed method and demonstrate the analysis involved. Figure 4.1 shows the generic procedure that takes an algorithmic description of a loop nest as input, classifies it in order to decide whether to apply loop transformations and if so which combinations of them. It finally performs parallelization and vectorization (if supported by the target architecture) in order to

produce an optimization schedule for it. Furthermore, if the optimizer detects that the output data is not used in future loop iterations, then non-temporal store instructions are used in order to bypass the cache and reduce cache pollution. Non-temporal stores can help assure that data fetched into the cache by the hardware prefetchers do not get evicted before they can actually be used.

Table 4.1: Architecture and application Parameters

Notation	Description
Li_{CLS}	Li cache line size
Li_{way}	Li cache associativity
Li_{CS}	Li cache size
B_i	Problem size in ith Dimension
D_{TS}	Data Type Size
N_{Cores}	Number of Cores
$N_{threads}$	Threads per Core

Table 4.1 lists all the application and architecture-specific parameters that will be required throughout the optimization process. The former category considers the problem size (loop bounds in each dimension) as well as the size of the data type as the main parameters of interest. As for architecture-specific parameters, most of the information we need refers to the memory hierarchy of the system, such as the size, line size and associativity of each cache level. Other parameters include the native vector width of the architecture, and the number of available processing units.

Classification

The first step in the optimization flow involves the classification of the algorithm definition. The main purpose of this step is to decide whether transformations should be applied on the target loop nest, and if so whether they should focus on optimizing for temporal or spatial locality. The reason behind this distinction is twofold. Firstly, tiling (and therefore altering the stride of most load operations of a program) a loop nest with only contiguous memory accesses or no temporal locality may interfere with the efficiency of the streaming hardware prefetching unit and lead to suboptimal results. Secondly, tile size selection for (self-) spatial locality requires a different analysis than tiling that aims to exploit temporal reuse. The only notion of reuse in applications that only benefit from increased self-spatial locality would refer to cache line reuse, or more specifically to input data that belong to the same cache line.

The classification process that specifies whether to transform the loop nest and optimize for temporal or spatial locality can be seen in Figure 4.2.

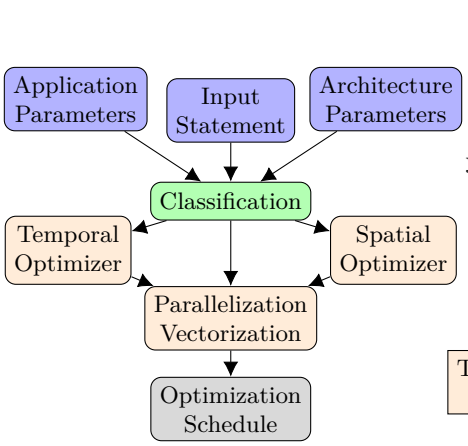


Figure 4.1: Optimization Flow

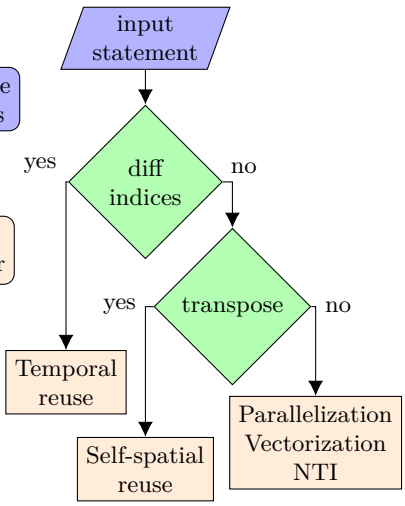


Figure 4.2: Classification Process

We first check if the unique indices in the input arrays (which appear on the right-hand-side of the algorithmic relation) of the algorithm description are different from the ones in the output array (which appear in the left-hand-side). If that's the case, then our algorithm contains multiple cache line references with temporal reuse possibilities. If we do not detect such a pattern, then it either means that only self-spatial reuse may be exploited, or that the algorithm only contains contiguous memory accesses and applying any loop transformation may alter the stride of the load operations and therefore interfere with the efficiency of the prefetching mechanisms. When optimizing for spatial reuse we check whether any arrays appear transposed in the statement. In this case, we transform the loop in order to ensure that useful data fetched to the L1 cache due to prefetching will not get evicted before they can be used. If none of the above patterns exist in the statement, then no further analysis is needed and no loop transformations are deemed beneficial. This decision is also supported by the work in [34], which explains that tiling may not be effective for stencil computations (even though they might reference multiple cache lines and therefore have some form of temporal reuse) due to uniform access patterns that can be easily exploited by the hardware prefetchers in modern architectures which can achieve the same level of reuse without the loop overhead of tiling.

Optimizations for temporal reuse

This section presents the analytical model, as well as the procedure that is followed in order to determine both the dimensions of the tile as well as the final loop permutation.

In general, our goal is to exploit reuse in both L1 and L2 caches in order to minimize the overall number of (cache) misses. More specifically, we pick tile dimensions such that L1 reuse is achieved in the outermost intra-tile loop and L2 reuse in the innermost inter-tile loop. The shared cache (L3) is also implicitly considered during the optimization procedure; modern hardware prefetching units are also capable of detecting non-unit strides in load operations in which case they fetch the expected data to the last-level-cache (and usually to the L2 as well). To better exploit this feature, we also aim to minimize the inter/intra-tile distance of each loop, therefore minimizing the stride of the equivalent load operations as well.

Table 4.2: Basic notation

Notation	Description
l_c	Amount of data that fits in one cache line
N_{sets}	Number of sets in cache
T_{width}	Tile width
B_c	Loop Bounds in the leading (column) dimension
max_{Ti}	Maximum tile size in ith dimension
T_{dims}	Number of tile dimensions
T_i	Tile size in ith dimension
ws_{Li}	Li cache working set
a_i	Li access time cost
C_{Li}	Estimated misses in Li cache
C_{order}	Loopnest permutation cost
$L2_{pref}$	L2 cache prefetches per access
$L2_{maxpref}$	Maximum prefetch distance
Lie_{way}	Effective associativity of Li cache

As an example, consider the code in Listing 4.1 which shows a simple C implementation of tiled matrix multiplication. In this case the classifier will recognize that different indices appear in the left and right side of the statement and thus will determine that temporal reuse should be exploited. We first want to achieve L1 cache reuse at the outermost intra-tile loop level (i).

For the loop nest of Listing 4.1, an iteration of the i loop accesses/loads a row from array C of width T_j, a row from array A of width T_k and a tile of size $T_k * T_j$ from array B. Thus the working set for the L1 cache in this case is:

$$ws_{L1} = T_j + T_k + T_j T_k \tag{4.1}$$

```
for ii=0;ii<Bi;ii+=Ti
 for kk=0;kk<Bk;kk+=Tk
  for jj=0;jj<Bj;jj+=Tj
   for i=ii;i<ii+Ti;i++
    for k=kk;k<kk+Tk;k++
     for j=jj;j<jj+Tj;j++ // vector loop
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
```

Listing 4.1: Tiled matrix multiplication

The total estimated cold misses in the L1 cache for one iteration of the i loop will be:

$$\frac{T_j}{l_c} + \frac{T_k}{l_c} + \frac{T_j T_k}{l_c} \tag{4.2}$$

However, due to the streaming prefetchers present in the L1 and L2 cache which fetch the next cache line after every reference, the estimated cold misses will be:

$$1 + 1 + T_k \tag{4.3}$$

Furthermore, for Ti iterations of the i loop, Equation 4.3) becomes:

$$T_i + T_i + T_k \tag{4.4}$$

Finally, the total number of estimated misses in the L1 cache after taking the inter-tile loop nest iterations into account will be:

$$C_{L1} = (T_i + T_i + T_k)(\frac{B_i B_j B_k}{T_i T_j T_k}) \tag{4.5}$$

Similarly, we want to achieve L2 reuse at the innermost inter-tile loop level (jj). One iteration of the jj loop will access a whole tile of arrays A, B and C. In this case the working set for the L2 cache will be:

$$ws_{L2} = T_j T_i + T_k T_i + T_j T_k \tag{4.6}$$

Moreover, just like for the L1 cache, the estimated number of cold misses for one iteration of the jj loop will be:

$$\frac{T_j T_i}{l_c} + \frac{T_k T_i}{l_c} + \frac{T_j T_k}{l_c} \tag{4.7}$$

Which after eliminating the prefetched references becomes:

$$T_i + T_i + T_k \tag{4.8}$$

Which in turn for $\frac{B_j}{T_j}$ iterations of the jj loop :

$$T_i \frac{B_j}{T_j} + T_i + T_k \frac{B_j}{T_j} \tag{4.9}$$

And finally after taking the other two inter-tile loops (kk,ii) into account we can compute the total estimated cost for the L2 cache:

$$C_{L2} = (T_i \frac{B_j}{T_j} + T_i + T_k \frac{B_j}{T_j}) \frac{B_i}{T_i} \frac{B_k}{T_k} \tag{4.10}$$

After computing both (4.5) and (4.10) we can compute the final cost function:

$$C_{total} = a_2 C_{L1} + a_3 C_{L2} \tag{4.11}$$

We use a weighted cost function where the a_2 and a_3 are the relative access times of L2 and L3 cache respectively. We assume that the hardware prefetching unit can follow the strides of the memory references and therefore fetch the equivalent data into the L2 and L3 cache.

C_{order} is the cost function that describes the total distance in terms of iterations between the "equivalent" intra and inter tile loops, that belonged to a single original dimension before tiling was applied. In detail the partial costs for Listing 1 are: $T_i T_k$, $\frac{B_j}{T_j} T_i$ and $\frac{B_j}{T_j} \frac{B_k}{T_k}$ for the j, k, i original loops respectively. The total loop permutation cost would be:

$$C_{order} = (\frac{B_j B_k}{T_j T_k} + \frac{B_j T_i}{T_j} + T_i T_k) \tag{4.12}$$

It is obvious that for a different inter-tile or intra-tile permutation, a different loop would be at the outermost intra-tile loop level (or innermost inter-tile), which in turn would lead to a different L1/L2 working set and a different number of estimated misses. This explains why we evaluate all possible permutations.

Algorithm 1 is used to acquire an upper bound on the dimensions of the tile, such that no interference misses occur. In detail, it emulates the behavior of the cache, by fetching tile rows into the array emu_{cache} and testing whether the set that the new data will be mapped to is already full, at which point the interference flag ($intr_{flag}$) becomes true and the upper bound max_{Ti} is returned. Furthermore, the algorithm keeps track of the prefetched data that might cause interference misses in the following way: If we are optimizing for the L1 cache, then we also need to consider the fact that for every cache line that is fetched, the next one is also brought into the cache by the hardware prefetcher. When optimizing for the L2 cache, we need to take into account that more than one prefetching requests may be issued at once, usually with a maximum distance between the actual reference and the prefetched data (usually 20 for Intel processors). For this reason we also fill the array with these extra lines in order to detect situations where the prefetched data might cause useful data to be evicted. To accomplish this, we track the total number of prefetched lines (s), as well as the distance between the actual reference and the prefetched line

$(s - p)$. Finally, in the case of L2, we limit the effective number of sets to half the original size. In other words, we reduce the effective cache size by half (and thus size of ws_{L2}) to account for the data that are fetched by the constant stride prefetchers. As the experiments show in subsection 4.5, this leads to efficient results, especially in the case of processors without L3 cache where data is only brought to L2. All the relevant notation can be found in Table 4.2.

Algorithm 2 shows the procedure that is followed in order to optimize a loop nest for temporal locality. The first step is to obtain the proper tile dimensions that minimize misses in the L1 and L2 cache. To achieve this we evaluate all possible tile sizes, as constrained by the bounds returned by the cache emulation algorithm (Algorithm 1) (for the first three dimensions) and problem size (for loop nests with four or more levels) for all valid intra-tile and inter-tile permutations. Invalid permutations are considered those where the loops that correspond to column indices are outermost. For each possible tile we calculate the size of the working set in the L1 and L2 cache to ensure that the tile fits in the cache (in order to minimize capacity misses) and finally if the dimension that corresponds to the outermost intertile loop (the one that we plan to parallelize over cores/threads) fulfills the following constraint:

$$\frac{B_{outer}}{T_{outer}} \geq N_{threads/core} * N_{cores} \tag{4.13}$$

This constraint ensures that each core/thread can execute at least one iteration of the inter-tile loop nest in order to better distribute the computation load among the processing units. The tile dimensions that correspond to the minimum total cost are chosen for the final tile size. To better utilize the prefetching units, we introduce a second step in our procedure where we try to minimize the distance between the inter and intra-tile loops that correspond to the same original loop in the original nest. This way we minimize the reuse distance of the equivalent data, as well as the stride of the load operation that will occur on the next inter-tile reference. Finally, after tiling and reordering the loop nest, we merge the outer inter-tile loops when possible to reduce loop overhead and further exploit parallelism.

Optimizations for spatial reuse

Optimizing for spatial locality is important in applications with no temporal reuse possibilities. This subsection presents the analytical model and the procedure to obtain tile dimensions that take advantage of the streaming hardware prefetching units in applications with complex strides like transposed arrays.

Algorithm 1: Cache emulation Algorithm (emu)

Input: $L1_{CLS}, Li_{CS}, D_{TS}, T_{i-1}, Li_{way}, B_i, N_{threads}, addr, L2_{pref}, L2_{maxpref}$

Output: max_{Ti}

Initializations :

$l_c = \lfloor \frac{L1_{CLS}}{D_{TS}} \rfloor$

$N_{sets} = \lfloor \frac{Lics}{Liway * D_{TS}} \rfloor$

$Lie_{way} = \frac{Li_{way}}{N_{threads}}$

$max_{Ti} \leftarrow 0, s \leftarrow 0$

$intr_{flag} \leftarrow False$

if optimizing for L2 **then**

  $T_{i-1} = \lceil \frac{max(T_{i-1}, l_c)}{l_c} \rceil$

  $N_{sets} = \frac{N_{sets}}{2}$

else

  $T_{i-1} = \lceil \frac{max(T_{i-1} + l_c, 2 * l_c)}{l_c} \rceil$

end if

$emu_{cache}[N_{sets}] = 0$

repeat

  $set \leftarrow \lceil \frac{addr + max_{Ti} * B_i}{l_c} \rceil$

  for $i = 0$ *to* T_{i-1} **do**

    if $emu_{cache}[(set + i)] = Lie_{way}$ **then**

      $intr_{flag} \leftarrow True$

    else

      $emu_{cache}[(set + i)] + +$

      $s + +$

    end if

    if $s - i <= L2_{maxpref}$ **then**

      for $p = 0$ *to* $L2_{pref}$ **do**

        if $emu_{cache}[(set + i + p)] = Lie_{way}$ **then**

          $intr_{flag} \leftarrow True$

        end if

      end for

    end if

  end for

  if $intr_{flag} = False$ **then**

    $max_{Ti} + +$

  end if

until $intr_{flag} = True$ **OR** $max_{Ti} = B_i$

Return max_{Ti}

Algorithm 2: Temporal Reuse Optimizer

Input: $L1_{CLS}, L2_{CLS}, L1_{CS}, L2_{CS}, L1_{way}, L2_{way}, N_{cores}, B_0, .., B_n, D_{TS}$

Output: *Tile size, Loop order*

Step 1: *Loop Tiling:*

1: $i \leftarrow 0$
2: **for** Every inter-tile loop permutation **do**
3: **for** Every intra-tile permutation **do**
4: **if** Column index is outermost **then**
5: Skip to next permutation
6: **end if**
7: **repeat**
8: $PickT_i \leq B_c$
9: $i \leftarrow i + 1$
10: $max_{Ti} = emu(L1_{CLS}, L1_{CS}, D_{TS}, L1_{way}, B_c, N_{threads}, addr, 0, 0)$
11: Pick $T_i \leq max_{Ti}$
12: **if** $(T_{dims} > 2)$ **then**
13: $i \leftarrow i + 1$
14: $max_{Ti} = emu(L2_{CLS}, L2_{CS}, D_{TS}, L2_{way}, B_i, N_{threads}, addr, L2_{pref}, L2_{maxpref})$
15: Pick $T_i \leq max_{Ti}$
16: **if** $(T_{dims} > 3)$ **then**
17: **for** i=3 to T_{dims} **do**
18: Pick $T_i \leq B_i$
19: **end for**
20: **end if**
21: Calculate ws_{L2}, Estimate C_{MissL2}
22: **end if**
23: Calculate ws_{L1}, Estimate C_{L1}
24: **if** (ws_{L1}, ws_{L2} fit in cache **and** iterations per thread ≥ 1) **then**
25: CostFunction=$(a_2 C_{L1} + a_3 C_{L2})$
26: **end if**
27: **until** all valid tile sizes evaluated
28: **end for**
29: **end for**

Step 2: *Reorder Loop:*

30: **for** Every valid inter-tile loop permutation **do**
31: **for** Every valid intra-tile permutation **do**
32: Calculate C_{order}
33: **end for**
34: **end for**

As an example, consider the C code in Listing 4.2 which shows a tiled implementation of a transposition and masking algorithm.

```
for yy=0;yy<By;yy+=Ty
 for xx=0;xx<Bx;xx+=Tx
  for y=yy;y<yy+Ty;y++
   for x=xx;x<xx+Tx;x++
    out[y][x]=A[x][y]&B[y][x]
```

Listing 4.2: Tiled Transposition and Masking

In this case the classifier detects that the indices are the same in the input (left-hand-side) and output (right-hand-side) arrays, and that one array (A) appears transposed in the statement. As a result the algorithm is optimized targeting spatial locality, using Algorithm 3.

We again assume the presence of a streaming prefetcher in both levels of the cache hierarchy, which means that the processor will fetch the next cache line for memory references in A and B. Just like in the previous subsection, the cost of accessing one tile of the transposed array A will be equal to T_x which after taking the inter-tile loops into account becomes:

$$T_x \frac{B_x B_y}{T_x T_y} = \frac{B_x B_y}{T_y} \tag{4.14}$$

The total cost for array A will be:

$$C_{partial} = (\frac{B_x B_y}{T_y}) \frac{T_x}{l_c} \tag{4.15}$$

where we refer to the factor $\frac{T_x}{l_c}$ as the prefetching efficiency for array A which represents the efficiency of the constant stride prefetching unit in the L2 cache. This factor gets minimized for $T_x = l_c$ (assuming that all tiles have a minimum of l_c size in every dimension). In other words, the transposed array favors tiles that have the maximum height and the minimum width. Similarly, for array B the cost of accessing one tile be equal to T_y which after taking the inter-tile loops into account becomes:

$$T_y \frac{B_x B_y}{T_x T_y} = \frac{B_x B_y}{T_x} \tag{4.16}$$

The total cost for array B will be:

$$C_{partial} = (\frac{B_x B_y}{T_x}) \frac{T_x}{l_c} \tag{4.17}$$

Finally the working sets for the two levels of cache:

$$ws_{L1} = l_c T_x + T_x \tag{4.18}$$

Algorithm 3: Spatial Locality Optimizer

Input: $L1_{CLS}, L2_{CLS}, L1_{CS}, L2_{CS}, L1_{way}, L2_{way}, B_0, .., B_n, D_{TS}$
Output: *Tile size*
Initializations :
1: $l_c = \lfloor \frac{L1_{CLS}}{D_{TS}} \rfloor$,
2: **repeat**
3: $C_{Total} \leftarrow 0$
4: Pick $T_{width} \leq B_c$
5: $i \leftarrow i + 1$
6: $max_{Ti} = emu(L2_{CLS}, L2_{CS}, D_{TS}, L2_{way}, B_i, N_{threads}, addr, L2_{pref}, L2_{maxpref})$
7: Pick $T_i \leq max_{Ti}$
8: Calculate ws_{L2}
9: Calculate ws_{L1}
10: **for** Every input array **do**
11: **if** (ws_{L1}, ws_{L2} fit in cache **and** iterations per thread ≥ 1) **then**
12: Calculate $C_{partial}$
13: $C_{Total} += C_{partial}$
14: **end if**
15: **end for**
16: **until** all valid tile sizes evaluated

$$ws_{L2} = 2T_xT_y \tag{4.19}$$

Algorithm 3 shows the pseudocode for the spatial locality optimizer. Just like in the previous subsection, we use Algorithm 1 to obtain an upper bound for the tile dimensions (tile height for 2 dimensional arrays). We calculate the working sets and if the tile height also fulfills equation (4.6), then for each input array we calculate the partial cost ($C_{partial}$) as explained in the previous example (equations (4.16),(4.17)). The final cost function C_{Total} is equal to the sum of all $C_{partial}$ costs. We evaluate all valid tile sizes as constrained by the bounds returned from Algorithm 1 and the problem size in the leading (column) dimension, and the tile that corresponds to the minimum C_{total} is chosen as the final tile size.

Parallelization, vectorization, non-temporal instructions

Standard optimizations include performance optimizations that can be applied after properly transforming the loop nest. These optimizations usually include vectorization and parallelization. Another possibility is the usage of non-temporal stores in applications with no temporal reuse in the output data. Non-temporal instructions can bypass the various

levels of the memory hierarchy in order to avoid cache pollution. For applications with contiguous memory accesses, tiling is unnecessary, since the streaming hardware prefetching units are already capable of fetching the next cache line along with the data that will be needed in the near future. This is why in such cases we bypass all tiling transformations during the optimization flow.

4.4 Experimental framework

In this section we present the experimental framework that was developed for the Halide DSL and compiler [73]. The red box in Figure 4.3 highlights the optimizer that is described in this work and which is implemented as a tool to be used with Halide. Listing 4.3 gives an example implementation of a matrix multiplication implementation in the Halide language, along with an optimization schedule. We should emphasize that our proposed optimization flow can be used with any other compiler/back-end but the Halide DSL was chosen in order to make use of the scheduling directives that enable quick application of various loop transformations and optimizations as seen in Listing 4.3. Such a framework is especially useful in many applications in the image processing domain, where most parameters are fixed and known at compile-time.

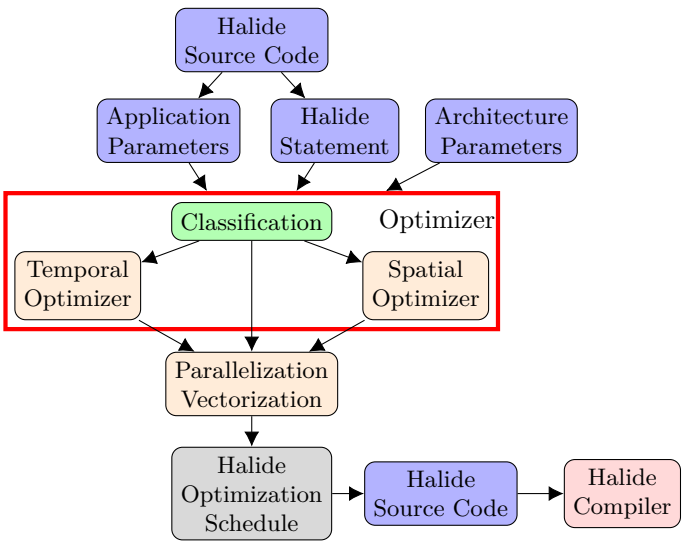


Figure 4.3: Experimental Halide Optimization Flow

As already mentioned in subsection 4.2, there are currently two ways to generate optimization schedules for Halide functions: the Halide Auto-

```
1 //Algorithm Definition
2 C(j,i)=0;
3 C(j,i)=C(j,i)+A(k,i)*B(j,k);
4 //Optimization Schedule
5 C.update().split(j,j_o,j_i,512)
6  .split(i,i_o,i_i,32)
7  .reorder(j_i,i_i,j_o,i_o)
8  .vectorize(j_i,8)
9  .parallel(i_o);
```

Listing 4.3: Matrix Multiplication in Halide & example optimization schedule for an Intel-based x86 architecture.

Scheduler [52] uses a heuristics-based algorithm to decide on the tile size and final loop permutation, while the autotuner [5] iteratively searches the design space with various schedule configurations in order to minimize the execution time of the final application. We use those two approaches as references for comparison with our framework.

Our framework requires the definition of a Halide function, along with the application and architecture specific parameter as input to the optimizer. The Halide statement is then processed during the classification step, and depending on the information that is derived and the patterns that can be recognized, a different optimization technique is used, as explained previously in subsection 3.

Furthermore, since the Halide compiler cannot generate non-temporal instructions, we extend it with a new scheduling directive - `.non_temporal()` - that produces non-temporal stores in the generated code when used in the optimization schedule of a function. To this end, we introduce a new optimization to the Halide front-end that allows the compiler to mark a function and the subsequent Halide Intermediate Representation (IR) store nodes as non-temporal in order to internally use that information to generate specific instructions (both scalar and vector variants) with non-temporal hints during the LLVM code generation pass in the back-end. Examples of such instructions (and the ones generated by the compiler in the following experiments) in Intel platforms with SSE/AVX support are the vector operations `(v)movntdq, (v)movntps` for integer and single precision floating point data types respectively.

4.5 Experimental results

Comparison to Halide approaches

This subsection presents the results that were obtained for a variety of benchmarks. All experiments were conducted multiple times measuring the average execution time of 100 runs for each benchmark. The run-time difference between runs of the same experiment was less than 1%.

We compare our results with the equivalent that the Halide Auto-Scheduler and autotuner generate on the same platform. Table 4.3 shows the hardware specifications of the target architectures that were used throughout the experimental process, while Table 4.4 lists the benchmarks, the problem size used in each of them along with the average execution time of the best implementation (in terms of execution time) for each benchmark to be used as reference for the following graphs. We chose three different architectures to showcase the flexibility of our approach in platforms with different architectural parameters. Specifically, the two Intel platforms differ in the number of cores and therefore may lead to different tile sizes (Equation 4.13), while the ARM architecture operates on a completely different memory hierarchy and utilizes one thread per core. Finally, Table 4.5 shows the runtime of our framework for each benchmark. In most cases, the tool is able to provide solutions within milliseconds, with the only exception being the convolution layer benchmark due to the large number of nested loops present in the tiled version of the algorithm and therefore the large number of possible loop permutations.

Table 4.3: Experimental Platforms

	Intel i7 5930k	**Intel i7 6700**	**ARM Cortex A15**
L_{CLS}	64B	64B	64B
$L1_{way}$	8	8	2
$L1_{CS}$	32KB	32KB	32KB
$L2_{way}$	8	8	16
$L2_{CS}$	256KB	256KB	512KB
N_{Cores}	6	4	4
$N_{threads}$	2	2	1

Figure 4.5 shows the throughput (1/s) relative to the fastest implementation for the two Intel platforms. The autotuner bar refers to the schedule that the Halide autotuner converges to after one hour of runtime. The Baseline bar corresponds to the most basic optimization a developer may perform, which usually includes parallelization of the outer loop and vectorization of the inner one. Finally, in order to make the comparison clearer, and since neither the autotuning nor the autoscheduling methods are able to generate non-temporal instructions, we separate the results

Table 4.4: Benchmarks

Benchmark & Problem Size	Average execution time (ms) Best implementation Intel i7 6700	Intel 5930K	ARM A15
`convlayer` 3x3x64x64 Convolution Layer 256x256x64x16	887.12	503.80	8897.29
`doitgen` Multiresolution Analysis Kernel 256x256x256	233.29	143.77	2824.87
`matmul` Matrix Multiplication 2048x2048	298.97	182.24	2080.58
`3mm` Three Matrix Multiplications 2048x2048	310.97	178.90	1564.15
`gemm` Generalized Matrix Multiplication 2048x2048	286.12	183.00	1503.06
`trmm` Triangular Matrix Multiplication 2048x2048	199.44	131.76	1295.14
`syrk` Symmetric rank k update 2048x2048	742.57	364.80	3575.62
`syr2k` Symmetric rank 2k update 2048x2048	1442.41	992.61	7269.75
`tpm` Matrix Transposition and Masking 4096x4096	10.02	6.00	41.87
`tp` Matrix Transposition 4096x4096	7.23	4.50	39.00
`copy` Array Copy 4096x4096	5.49	3.18	-
`mask` Array Mask 4096x4096	8.32	4.67	-

where the classifier decides to use non-temporal (streaming) stores. The first nine benchmarks (`convlayer`, `doitgen`, `matmul`, `3mm`, `trmm`, `gemm`, `syrk`, `syr2k`) have been optimized for temporal reuse, while the transposition (`tp`), transposition and masking (`tnm`), `copy` and `mask` kernels have been optimized for spatial reuse. Non-temporal instructions can also be used for the four last algorithms.

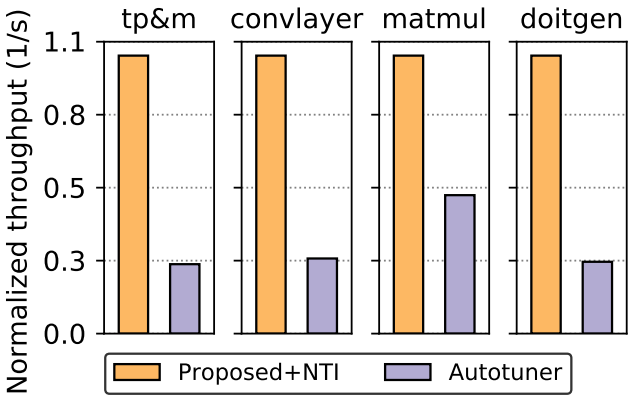
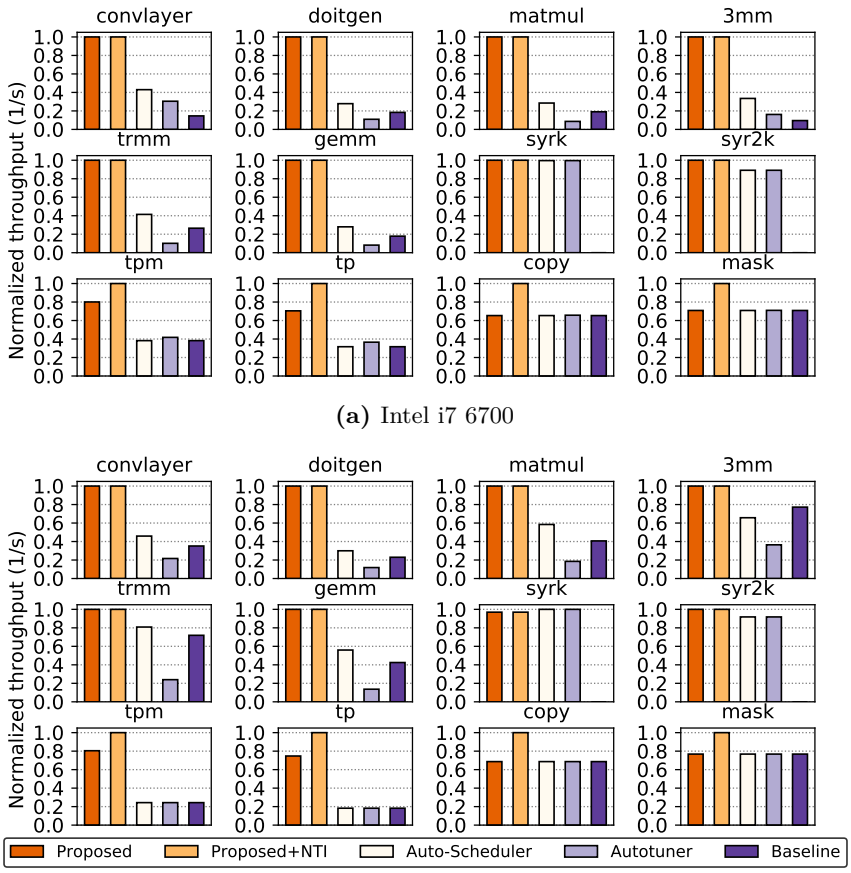


Figure 4.4: Throughput (1/s) relative to fastest implementation; autotuner ran for 1 day on Intel 5930K , NTI refer to implementations with non-temporal instructions.

The autotuning framework generates relatively poor schedules for most benchmarks either because it excludes schedules with tiling in all dimensions, or because it needs even more time to converge to a better solution. Due to this reason, we performed an extra experiment for the matrix multiplication, doitgen, convolution layer and transposition and masking benchmarks, where we compare our solution to the schedule generated by the autotuner after one day of runtime. It should also be noted that most of the applications had to be rewritten in a Halide-specific way that uses helper functions (e.g. sum) and bypasses the initial definition of the algorithms (e.g. initialize sum to zero) in order to be optimized by the autotuner. These functions rely on the compiler to perform some optimizations instead of actual loop transformations specified by the developer. Without this alternate definition, the autotuner would only attempt to optimize the initialization step and not the actual computation. The syrk and syr2k benchmarks could not be rewritten in such a way and thus the autotuned implementations are excluded.

Figure 4.4 demonstrates the performance of the solutions generated after one day of autotuning along with the results of our framework. We chose algorithms with different loop dimensions (2, 3, 4, 5 dimensions for



(a) Intel i7 6700

(b) Intel i7 5930K

Figure 4.5: Intel platforms - Throughput (1/s) relative to fastest implementation (see Table 4.4)

the transpose and masking, matrix multiplication, doitgen and convolution layer respectively) in order to compare our analysis for transforming N-dimensional loops with stochastic autotuned methods. These results are similar to the ones presented in Figure 4.5 and therefore strengthen our decision to tile each dimension of the input loop nest, as opposed to the autotuner schedules that only attempt tiling in the dimensions of the output array.

The schedules provided by the Auto-Scheduler offer a significant speed-up compared to both the baseline schedules and the autotuned ones. However, our schedules still perform significantly better for most benchmarks. The syrk and syr2k benchmarks are the only exceptions where our approach performs similar to the baseline schedule due to the fact that the algorithms contain memory references along cache lines, and therefore do not significantly benefit from tiling. However, as expected, after repeating the experiments for larger problem sizes, the tiled version performed around 25% better than the baseline schedule.

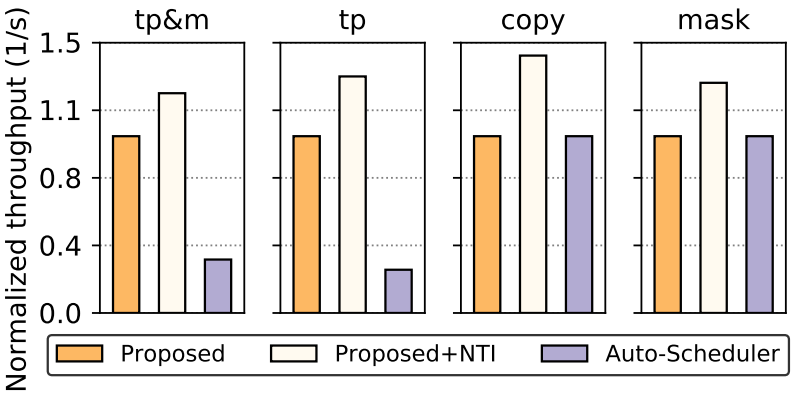


Figure 4.6: Throughput (1/s) relative to Proposed Non-NTI implementation, NTI refer to implementations with non-temporal instructions. (Intel 5930K)

Table 4.5: Optimization runtime

Benchmark	`convlayer`	`doitgen`	`matmul`	`3mm`	`gemm`	`trmm`
	7.604s	0.153s	0.006s	0.006s	0.006s	0.005s
Runtime	`syrk`	`syr2k`	`tpm`	`tp`	`copy`	`mask`
	0.009s	0.012s	0.002s	0.002s	0.002s	0.002s

Figure 4.6 shows the effect of non-temporal store instructions in the applications where the classifier does not detect output data reuse for

the Intel i7-5930K platform. As seen in the graph, this optimization can significantly improve the performance in applications with no temporal reuse on the output data due to a reduction of the total number of cache misses.

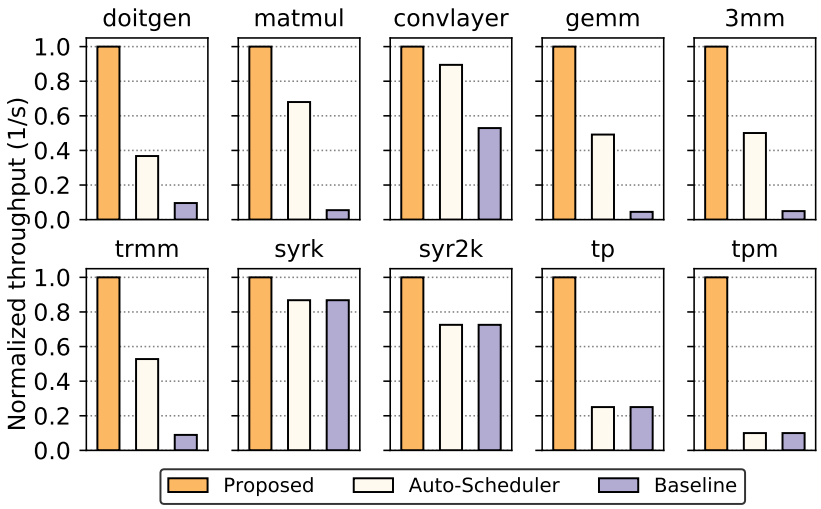


Figure 4.7: ARM Cortex A15 platform - Throughput (1/s) relative to fastest implementation.

Figure 4.7 demonstrates the results for the ARM Cortex A15 architecture. This architecture does not have an L3 cache and the L2 one is shared among the four cores of the platform. Due to this reason, a minor change to the model was required before conducting the experiments: The calculation of $L2_{way}$ in Algorithm 2 should be updated to $\frac{L2_{way}}{N_{cores}}$ instead of $\frac{L2_{way}}{N_{threads}}$ to account for this fact. Furthermore, since the ARM architecture does not support vector stores with non-temporal hints the mask and copy algorithms are not included in this graph (their performance is identical in all three of the proposed, baseline and Auto-Scheduler implementations). In general our proposed algorithm outperforms the Auto-Scheduler and baseline on this architecture as well.

Comparison to other tiling approaches

In this subsection, we compare our approach to previous state of the art analytical models for automatic tile size selection. Namely we pick the TSS method proposed in [49] as well as the TTS method which was

introduced in [50]. We pick these techniques as they have similarities with our approach: The TSS method considers reuse in the L1 and L2 cache without taking prefetching into account, while the TTS technique optimizes for L2 and L3 cache while taking advantage of hardware prefetching. However, prefetching is not considered in the analytical model and prefetched references are not taken into account while estimating the number of cold misses in every iteration. As a result, the proposed tile sizes for both TTS and TSS are different than the ones picked by our approach.

Table 4.6: Average execution time (ms) - Intel 5930K

Problem Size		**400**			**800**	
Benchmark	**TTS**	**TSS**	**Proposed**	**TTS**	**TSS**	**Proposed**
matmul	1.76	**1.54**	1.65	12.08	15.06	**9.35**
tmm	1.01	1.19	**0.93**	5.93	22.42	**5.02**
syrk	14.80	7.20	**5.80**	96.24	115.07	**69.04**
syr2k	30.57	13.22	**11.29**	58.88	86.99	**51.21**
Problem Size		**1024**			**1600**	
matmul	23.56	71.97	**20.46**	98.65	104.18	**71.62**
tmm	**10.01**	35.46	10.57	58.47	137.83	**36.21**
syrk	224.88	228.83	**159.47**	242.11	294.84	**213.32**
syr2k	228.33	248.60	**97.14**	451.45	536.22	**314.38**

Table 4.6 shows the average execution time (ms) on the Intel i7-5930K platform for the three methods. Since both TTS and TSS use a different framework and back-end compiler, we are not able to reproduce their optimization flow. For this reason, we choose this specific platform for our experiments, since it has similar cache hierarchy ($L_{CLS}, L1_{way}, L1_{CS}, L2_{way}$, $L2_{CS}$) as the one used in [50] in order to use the tile dimensions picked by TTS and TSS as listed in [50], with a difference on the size of the L3 cache and the number of cores. However, since the size of the L3 cache in both platforms is large enough, and the effective size per core is the same, we do not expect a big impact on the final tile dimesions. Furthermore, since nor TTS, nor TSS consider loop interchange, we try every possible loop permutation for each benchmark and pick the one that results in the best performance to include in our experiments. We compare the three techniques for the four benchmarks that are common between the ones used in [50] and the ones we used in subsection 4.5 and four different problem sizes.

The results presented in Table 4.6 indicate that our method outperforms the other two techniques by up to two times on the syr2k benchmark. Furthermore, the experiments show that our proposed approach generates results which are on average 26% and 41% faster than the solutions provided by TTS and TSS respectively.

4.6 Summary

In this chapter we proposed an optimization framework for memory bound applications that considers architecture and application specific parameters while taking advantage of the hardware prefetching mechanisms in modern platforms. We implement it as a tool to be used with the Halide DSL and compiler and compare it to both other analytical as well as empirical methods. Experimental results indicate a significant improvement in performance compared to previous models, while providing solutions usually within milliseconds. These results show that interaction between loop transformations and the sophisticated hardware prefetching mechanisms in modern architectures is of utmost importance when optimizing memory intensive applications.

5

Reuse analysis for multi-stage pipelines

5.1 Introduction

High-tech systems such as wide format printers, radars, and health-care monitoring applications execute complex image processing algorithms on a target platform which is typically a multi-core CPU (e.g. from ARM or Intel) with SIMD extensions. In order to meet the real-time constraints, the final implementation needs to be highly optimized for the target platform. Traditional optimizations usually include a series of loop transformations, such as tiling and loop fusion as well as vectorization and parallelization and aim to exploit locality (spatial and temporal), data level parallelism and task level parallelism respectively. However, manually applying these transformations siginificantly reduces code readability and portability and discourages high level design space exploration.

Recently, domain-specific-languages (DSLs) such as Halide [73], which was introduced in detail in Chapter 3 and PolyMage [53] were introduced in order to facilitate the optimization process in high-performance image processing applications. These DSLs allow developers to express applications in a more abstract format while maintaining the ability to apply low level optimizations and transformations on the final code. The benefits of approaches can be invaluable in the case of image processing pipelines where a combination of optimizations including stage interleaving or stage

fusion, tiling, vectorization and parallelization are necessary in order to achieve high performance.

As explained in Chapter 2, an image processing pipeline can be defined as a series of functional stages, where each stage contains an arbitrary number of nested loops and depends on data produced during an earlier stage. As a result, interleaving the computation of these stages can offer significant performance improvement by exploiting producer/consumer locality and ensuring that intermediate buffers are kept inside the local caches or registers. Both Halide and PolyMage employ techniques that allow for the automatic optimization of such imaging pipelines. The Halide Auto-Scheduler [52] attempts to group stages together and evaluates an effective tiling in each group. PolyMage can use both auto-tuning to search parts of the design space as well as a recently introduced model-driven approach [32]. This new approach quickly attempts to fuse stages and extends the search space in order to cover more solutions than the previous auto-tuning method. However, all three techniques [32, 52, 53] focus on the interleaving of the computation of each stage using overlapping tiles and therefore lead to solutions with limited reuse possibilities and often miss sliding window opportunities.

In this chapter we present a novel optimization strategy for image processing pipelines that considers stage fusion for maximum producer/consumer locality in conjunction with tile size selection while evaluating reuse possibilities not considered in previous state-of-the-art approaches. Our technique is driven by an analytical model that takes relevant application and architecture specific parameters (such as the number of cores/threads, cache size, interaction with hardware prefetching) into account and is capable of producing optimized schedules within seconds, even for complex pipelines with a large number of stages. We implement it as a tool to be used with the Halide DSL, as an alternative cost model and analysis to the Halide Auto-Scheduler and evaluate it across a variety of benchmarks and target platforms. We compare our solutions to the ones produced by the Halide Auto-Scheduler, the manual solutions given for the Halide DSL on the same benchmarks when applicable, as well as the ones produced by PolyMage (using both the original auto-tuned method, as well as the DP-fusion technique implemented in [32]) on the same target architectures. We observe a substantial performance improvement across all platforms and architectures.

It is important to remark that our technique is not restricted to Halide. It can be used with other DSLs and general purpose compilers that target image processing, tensor or linear algebra applications and offer control over the production and consumption of pipeline stages, as well as the allocation of intermediate buffers.

The rest of this chapter is organized as follows: subsection 5.2 discusses related work. Subsection 5.3 gives a motivational example, while

subsection 5.4 presents our proposed optimization technique in detail. Subection 5.5 showcases the experimental results that were obtained. Conclusive remarks are finally discussed in subsection 5.6. This work was published in [82].

5.2 Related work

In this section we discuss prior related work. We identify the limitations of traditional loop fusion and tiling techniques used in general purpose languages when optimizing image processing pipelines and investigate some of the benefits of recent image processing domain-specific-languages.

General purpose languages

Loop fusion in conjunction with tiling has been extensively studied in the past, especially in the case of general purpose compilers. Most of these approaches focus on exploiting data locality while maintaining parallelism in applications dominated by linear algebra or stencil computations [35,61, 95, 99, 103]. More specifically, the authors in [103] propose a hierarchical tiling technique for iterative solver applications in order to reduce communication overhead without introducing severe redundant computation. In [61], the effects of various inter-loop optimization strategies on PDE solvers are investigated.

In [17] the authors propose an optimization strategy for compute-intensive multi-dimensional summations that involve products of several arrays. They investigate the effects of loop fusion and tiling in such applications while also reducing the memory footprint of intermediate temporary buffer requirements.

Other approaches have focused on enabling loop fusion in applications with complex data dependencies between loop iterations [47, 98]. The authors in [98] propose a technique that eliminates fusion-preventing dependencies by means of loop tiling and array copying. After iteratively applying the aforementioned method to multiple loop nests, a single equivalent nested loop can be formed that can be tiled for cache locality. In a similar fashion, [47] proposes a way of mitigating the presence of fusion-preventing dependencies, while maintaining parallelism and eliminating cache conflicts in the subsequent fused loops.

However, all aforementioned methods involve traditional loop fusion techniques that target time iterated stencils, the scope of which differs from the complex multi-dimensional problems defined in the context of image processing pipelines, a term that covers all applications within the scope of this work. Stages in these pipelines perform various data-parallel

computations before having their output consumed by the next stage, which in turn executes a different computation or stencil.

Domain specific languages

Recently, domain specific languages (DSLs) have emerged that enable quick design space exploration in the image processing domain. These DSLs provide high level abstractions in the definitions of the functional steps inside the pipeline, as well as the ability to apply optimizations on the generated code in order to ensure high performance on the final implementation.

Tensor Comprehensions [87] is an example of a recent DSL that targets deep learning applications such as convolutional and recurrent neural networks. It consists of a high level language with syntax that resembles the mathematics of deep learning and a Just-In-Time polyhedral compiler for CUDA-based GPU architectures. It employs an autotuner in order to automatically generate efficient polyhedral schedules.

PolyMage [53] is another DSL for image processing applications that uses a dataflow-like language to describe pipelines. It employs polyhedral transformations [29,39,66] to optimize the computations performed by the functional stages of the pipeline with a grouping-then-tiling approach. More specifically, it relies on auto-tuning over various tile dimensions, which are all powers of two, in order to decide which stages of the pipeline will be grouped together. It then applies polyhedral optimizations on each group to generate the final nested loops. An alternative optimization strategy for pipelines implemented in PolyMage was introduced in [32]. This method introduced a dynamic fusion and tiling model that extends the search space to tile sizes that are not powers of two and resolves the need for auto-tuning. However, due to the nature of the analysis that is used in the PolyMage compiler, its application scope is limited to stencil computations and up/down sampling.

In Halide [73], image processing pipelines are defined as directed acyclic graphs, where each node of the graph represents a functional stage. Each stage is equivalent to a Halide function, which specifies all producer/-consumer relations at the specific stage. Furthermore, the functional description of the pipeline is independent of its optimization schedule. In other words, the Halide functions define the relations and dependencies between the stage of the pipeline, but do not influence the way the stages will get executed. As a result, the optimization schedule can control both the order of execution within a single stage, as well as the way the computation of stages gets interleaved during the execution of the pipeline. Figure 5.1b shows a simple two-stage blur filter implemented in Halide, along with its optimization schedule. Given this schedule, the compiler will tile the loop of the *blury* filter using a tile size of 256x32, vectorize

(a) Pipeline graph

```
1  Func blurx, blury
2  Var x, y, xi, yi;

4  // The algorithm
5  blurx(x,y) = (input(x-1,y) + input(x,y) + input(x+1,y))/3;
6  blury(x,y) = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1))/3;

8  // The schedule
9  blury.tile(x,y,xo,yo,xi,yi,256,32)
10      .vectorize(xi,8).parallel(yo);
11 blurx.compute_at(blury,xo).vectorize(x,8);

13 blury.realize(1024,1024);
```

(b) Halide algorithm definition & schedule

```
1  parallel yo in [0,1024/32):
2    for xo in [0,1024/256):
3    allocate blurx[34][256]
4      for y in [-1,33):
5        for x.o in [0,32):
6          vectorized x.vector in [0,8)
7            blurx(...) = ...
8      for yi in [0, 32):
9        for xi.o in [0,32):
10         vectorized xi.vector in [0,8)
11           blury(...) = ...
```

(c) Equivalent C-like loop-nest

Figure 5.1: 3x3 Blur pipeline

the innermost intra-tile loop (xi) using vectors of size 8 and parallelize its outermost inter-tile loop (yo). Furthermore, the computation of the blurx stage will be interleaved on a per-tile basis and its innermost loop will also be vectorized using vectors of size 8. In other words, before each intra-tile loop iteration, Halide will first allocate buffer space and compute all pixels of blurx that will get consumed during this iteration. The equivalent loop-nest in pseudo-C can be seen in Figure 5.1c.

Halide initially employed an auto-tuning framework to automatically generate optimized schedules for pipelines [73] which required an extensive amount of time in order to derive an adequate schedule. A more generic auto-tuning approach which is driven by genetic algorithms was proposed in the auto-tuning framework Opentuner [4]. This framework was able to generate efficient schedules in less time for small pipelines (e.g. bilateral grid), but fails to converge to a good solution for larger, more complex problems.

Currently, Halide uses a heuristic based Auto-Scheduler which was initially proposed in [52] but then received an updated cost model by the Halide community [26]. This method uses a greedy grouping algorithm to group stages of the pipeline together in order to maximize producer/consumer locality and applies tiling to the output stage of each group independently. However, the grouping strategy excludes parts of the design space, considers only a limited number of tile sizes and its analysis does not cover buffer allocation and storage scheduling. As a result, while it can quickly produce schedules within seconds, it misses interesting solutions of the design space which may benefit from sliding window opportunities. Those missed solutions may however allow for better SIMD vector unit utilization and better exploitation of the hardware prefetchers.

The analytical model we introduced in Chapter 4 may also automatically schedules kernels in Halide. However, while it can take hardware prefetching into account and involves a hierarchical tiling approach, it is limited to single stage pipeline and stage fusion falls outside its analysis.

Our method considers both the compute as well as the storage levels of a stage while determining its final optimization schedule. We show that by taking both compute and store level into account, we can reduce the amount of intermediate temporary buffer space required, which in return allows for different grouping and tiling options as well as increased producer/consumer locality. Furthermore, our analytical model takes hardware prefetching inherently into account and investigates tile sizes in a larger scope.

Other, more recent schedulers were proposed (after publication of this work) which also attempt to schedule Halide applications [2]. Due to this reason, the evaluation section of this chapter only compares to solutions available at the time of publication.

5.3 Motivational example and problem formulation

In this section we use the blur pipeline seen in Figure 5.1 as a motivational example in order to demonstrate the limitations of current state-of-the-art approaches, as well as the idea behind our work.

As already mentioned, optimizing an image processing pipeline usually involves dealing with a complex trade-off between parallelism, locality and recomputation. The transformations that are often considered include a combination of loop interchange, splitting, fusion, parallelization and vectorization. Choosing a proper fusion strategy for each stage in a pipeline has a significant effect on the performance of the final implementation. Figure 5.2 shows three example schedules for the blur pipeline in pseudo-C syntax. The amount of reuse or recomputation, as well as the size of the intermediate buffer that is required can be controlled through the combination of various loop permutations, tile sizes, and levels at which we compute and store each stage of the pipeline. For example, the solution shown in Figure 5.2a computes all necessary pixels in `blurx` before consuming them in order to compute `blury`. Such a schedule avoids all recomputation but suffers from poor locality and a large intermediate buffer (depending on the problem size). On the other hand, fully inlining the producer (`blurx`) into its consumer (`blury`) increases locality but at the cost of the highest recomputation.

Current state-of-the-art approaches (e.g. the current Halide Auto-Scheduler), only consider scheduling options where compute and store are set to the same level of a loop nest. As an example, consider the schedule seen in Figure 5.2b. In this case, tiling the iteration space of `blury` and fusing its producer into the innermost inter-tile loop (`xo`), allows for an intermediate solution that offers increased locality compared to the fully stored implementation and less recomputation than the fully inlined one. Furthermore, it does not hinder parallelization of the outermost inter-tile loop, since the computation of `blurx` is interleaved at a lower level than the parallel loop. We can quantify the amount of intermediate storage (B_{blurx}) needed as well as the amount of recomputation (R_{blurx}) in such a schedule for arbitrary tile dimensions:

$$B_{blurx} = T_x(T_y + v_y) \cdot sizeof(DataType) \tag{5.1}$$

where T_x, T_y are the tile sizes in the x and y dimensions and v_y is the amount of overlap between `blurx` and its consumer `blury` in the y dimension (in this example $v_y = 2$).

$$R_{blurx} = Cp_{blurx}^{blury_{xo}^{xo}} - C_{blurx}^{root} \tag{5.2}$$

```
allocate blurx[By+2][Bx]
for y in [0,By+2):
  for x in [0,Bx):
    blurx(...) = ...

for y in [0,By):
  for x in [0,Bx):
    blury(...) = ...
```

(a) store/compute root

```
for yo in [0,By/Ty):
  for xo in [0,Bx/Tx):
    allocate blurx[Ty+2][
    Tx]
    for y in [-1,Ty+1):
      for x in [0,Tx):
        blurx(...) = ...
    for yi in [0, Ty):
      for xi in [0, Tx):
        blury(...) = ...
```

(b) overlapping tiles

```
for yo in [0,By/Ty):
  for xo in [0,Bx/Tx):
    allocate blurx[3][Tx]
    for yi in [-2, Ty):
      for x in [0,Tx):
        blurx(...) = ...
      if (yi<0): continue
      for xi in [0, Tx):
        blury(...) = ...
```

(c) sliding windows inside tiles

Figure 5.2: 3x3 Blur pipeline - scheduling options

where $Cp_{blurx}^{blury_{xo}^{xo}}$ is the total computation cost of *blurx* in this fusion scenario and C_{blurx}^{root} is the cost when all of its pixels are computed and stored before being consumed (as in Schedule (a)).

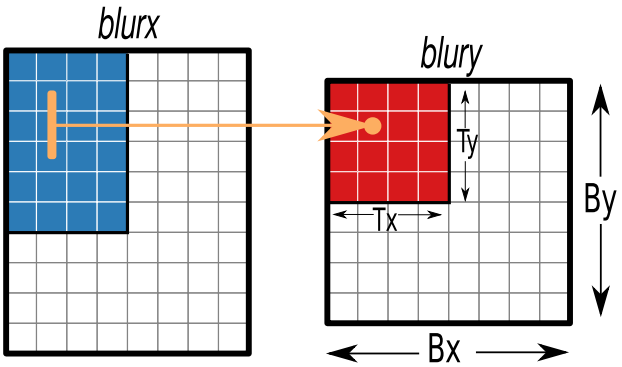
More specifically, $Cp_{blurx}^{blury_{xo}^{xo}}$ is the cost of `blurx` when fused into `blury` with its computation (subscript `xo`) and allocation (superscript `xo`) set to the `xo` level/index of the loop-nest of `blury`. Similarly, C_{blurx}^{root} is the cost of computing and storing `blurx` outside the loop nest of the consuming stage `blury`.

$$Cp_{blurx}^{blury_{xo}^{xo}} = T_x(T_y + 2)\frac{B_x}{T_x}\frac{B_y}{T_y} \tag{5.3}$$

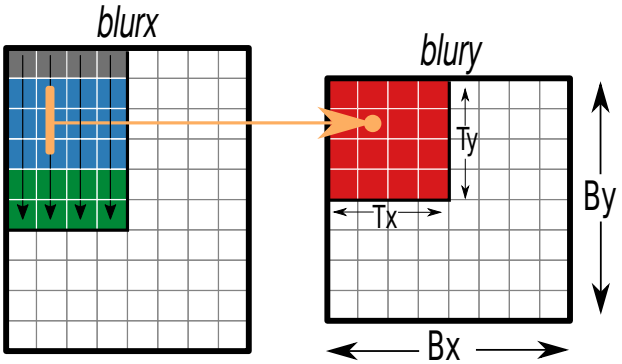
$$C_{blurx}^{root} = B_x(B_y + 2) \tag{5.4}$$

where B_x, B_y are the problem sizes (loop bounds) in the `x` and `y` dimensions respectively [1].

[1] In order to keep the equations and the example clear, we assume that the loop bounds in each dimension are a multiple of the tile size.



(a) Overlapping tiles: all blue pixels of `blurx` are evaluated before being consumed to produce an area equal to the output tile in `blury` (red pixels). The memory allocation for `blurx` should be as large as the blue area.



(b) Sliding windows inside tiles: values of `blurx` are computed per line of the output tile as needed. Pixels that are no longer needed are discarded (gray area). The intermediate buffer requirement is equal to the blue area. Green pixels will get evaluated in future intra-tile iterations

Figure 5.3: Overlapping tiles and sliding window implementations.

Similar equations can be used in order to calculate the load cost (C_l) for `blurx` which is equivalent to the load cost for the input data (C^{blurx}_{input}). In detail:

$$Cl^{blury^{xo}_{xo}}_{blurx} = C^{blurx}_{input} = (T_x + 2)(T_y + 2)\frac{B_x}{T_x}\frac{B_y}{T_y} \tag{5.5}$$

In the presence of a streaming hardware prefetcher [2], the previous equa-

[2]We assume that the problem size in the column dimension is larger than the size

tion becomes:

$$Cl_{blurx}^{blury_{xo}^{xo}} = (T_y + 2)\frac{B_x}{T_x}\frac{B_y}{T_y} \tag{5.6}$$

where we eliminate the sequential accesses across cache lines. Finally, the amount of data that needs to stay in the cache in order to benefit from input data reuse is:

$$B_{input} = (T_x + 2)(T_y + 2) \cdot sizeof(DataType) \tag{5.7}$$

Such a schedule benefits from increased locality compared to the one with root storage. Furthermore, as seen in the above equations, the trade-off between redundant computation and locality can be controlled by tuning the applied tile dimensions. It should also be noted that a different inter or intra-tile loop permutation leads to different buffer requirements and cost-functions. Figure 5.3a shows a visual representation of the schedule for an 8x8 output image with 4x4 tiling ($B_x = 8$, $B_y = 8$, $T_x = 4$ and $T_y = 4$). As seen in the figure, all blue pixels of `blurx` are evaluated and stored before being consumed to produce one red tile of `blury`.

The third schedule (Figure 5.2c) shows an implementation where computation and storage are set to different levels. Such schedules benefit from sliding window opportunities that usually enable the folding of intermediate buffers without reducing the amount of data reuse. As an example, consider the buffer requirements for this schedule (all of the other costs will be the same as in schedule 5.2b). Starting from Equation 5.1 (since the store level remains the same) we can calculate the memory footprint of the producer `blurx` stage as follows:

$$B_{blurx} = T_x(T_y + 2) \cdot sizeof(DataType) \tag{5.8}$$

Since pixels of `blurx` are now computed per line of the output tile (`yi`), we do not need to keep all of them in the intermediate buffer, but only those that can be reused across intra-tile iterations (or across one `xo` iteration). Therefore, B_{blurx} can be folded down to a circular buffer of size:

$$B_{blurx} = T_x(1 + 2) \cdot sizeof(DataType) = 3T_x \cdot sizeof(DataType) \tag{5.9}$$

The same holds for the input data buffer which will now be:

$$B_{input} = (T_x + 2)(T_y + 2) \cdot sizeof(DataType) \tag{5.10}$$

and will be folded down to:

$$B_{input} = (T_x{+}2)(1{+}2){\cdot}sizeof(DataType) = 3(T_x{+}2){\cdot}sizeof(DataType) \tag{5.11}$$

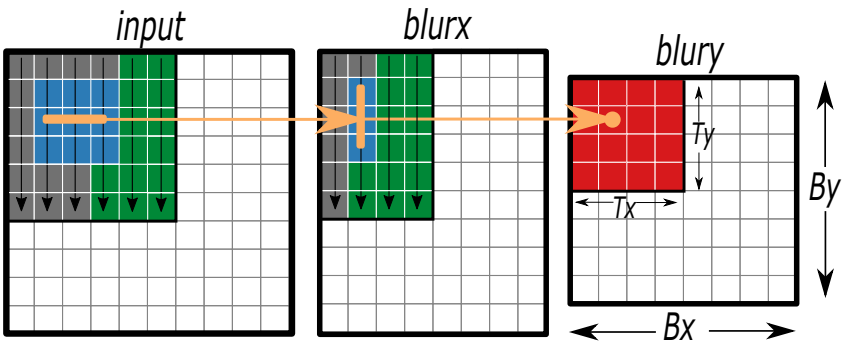
of a physical page and therefore the constant stride prefetchers cannot follow the stride of the non-consecutive load operations

Figure 5.3b shows a visual representation for a small 8x8 output image with an applied tile size of 4x4 ($B_x = 8$, $B_y = 8$, $T_x = 4$ and $T_y = 4$). Unlike Figure 5.3a, pixels of the producing stage `blurx` are produced per line (`yi`) of the consuming stage `blury` as needed. For example, three lines of width equal to Tx will be computed during the first yi iteration in order to produce one tile row but only one line of `blurx` will need to be computed for $yi > 0$ since two lines may be reused. Pixels that are no longer needed (cannot be reused across one iteration of `xo`) are discarded.

Note that the above schedule does not ensure maximum folding of the intermediate buffer allocated for `blurx`. For example, consider the schedule seen in Figure 5.4a, interchanging the loop such that the ordering (from innermost to outermost) is (yi, xi, yo, xo), setting the compute level of `blurx` to `yi` and its storage to `y` would allow the buffer to get folded down to just the amount of overlap across `y` without any extra recomputation compared to the previous schedule:

```
1 for xo in [0,Bx/Tx):
2   for yo in [0,By/Ty):
3     for xi in [0, Tx):
4     allocate blurx[3]
5       for yi in [-2, Ty):
6         blurx(...) = ...
7       if (yi<0): continue
8         blury(...) = ...
```

(a) maximum folding of the intermediate buffer



(b) Sliding windows inside tiles: values of `blurx` are computed per pixel of the output tile as needed. Pixels that are no longer needed are discarded (gray area). The intermediate buffer requirement is equal to the blue area. Green pixels will get evaluated in future intra-tile iterations

Figure 5.4: Maximum folding of the intermediate buffer `blurx`

$$B_{blurx} = (1 + v_y) \cdot sizeof(DataType) = 3 \cdot sizeof(DataType) \quad (5.12)$$

However, as it can also be seen from Figure 5.4b computing `blurx` at the innermost level of its consumer causes the loading of the input buffer to be much less efficient. In detail, since `input` is accessed in a column major order (three horizontal pixels at a time are needed to produce one pixel of `blurx`), prefetched (consecutive) cache lines will only be used after `Ty` iterations, or will not even be used at all if `Ty` is too large and they get evicted from the cache. As a result, the input load cost is now equal to:

$$Cl_{blurx}^{blury_{xi}^{yi}} = C_{input}^{blurx} = 3(T_y + 2)T_x \frac{B_x}{T_x} \frac{B_y}{T_y} = 3(T_y + 2)B_x \frac{B_y}{T_y} \quad (5.13)$$

where the T_x factor can no longer be simplified since accesses to *input* are not consecutive and the schedule does not benefit from hardware prefetching (as much as the previous one). For reference, the previous schedule (Figure 5.2c) performs twice as fast compared to this one, even though it does not maximize folding.

Based on the above (Equations 5.9 and 5.11, Figure 5.3), we can conclude that folding the intermediate buffers leads to much smaller local memory requirements without sacrificing data reuse or increasing the amount of redundant computations. As a result, solutions that were previously not considered, e.g. tile sizes that led to large memory footprints can now easily be captured by separating the computation and storage of a stage. However, as seen by comparing Equations 5.13 and 5.6, maximum folding does not always ensure exploitation of the spatial locality or the hardware prefetching mechanisms of the platform. Due to this reason a trade-off analysis between reuse, recomputation, input loading cost and memory requirements has to be conducted.

Figure 5.5 shows an abstract representation of the design space when considering stage fusion. As already mentioned, previous state-of-the-art techniques only consider solutions which reside within a small area of this space. The Halide Auto-Scheduler only produces solutions where the compute granularity of a stage is the same as its storage granularity. As a result, the generated schedules are limited to fully inlined, fully stored and tiled implementations with redundant computations where the computation and storage are set to the innermost inter-tile level (overlapping tiles). Figure 5.6 shows the distinct solutions for the above loop permutation of the blur example. The root and inlined solutions have been excluded due to limited reuse, parallelism or locality as explained in the above example. We can notice that most solutions of the design

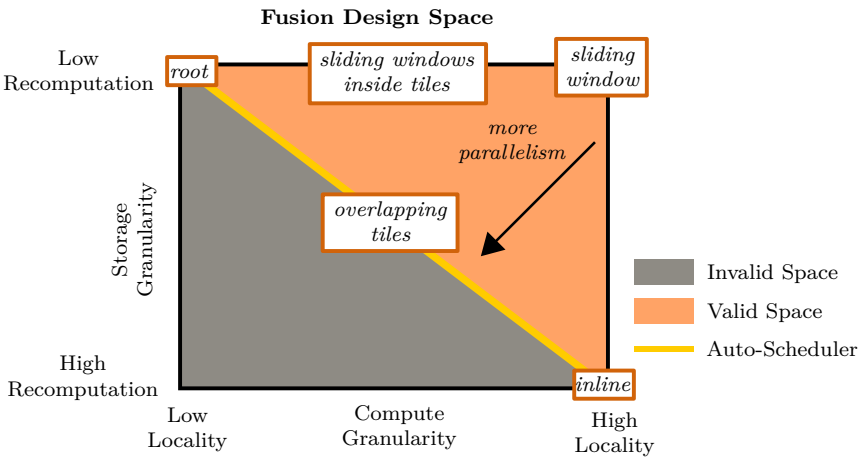


Figure 5.5: Fusion solution space, adapted from [73]

space are currently not considered and all sliding window opportunities are missed.

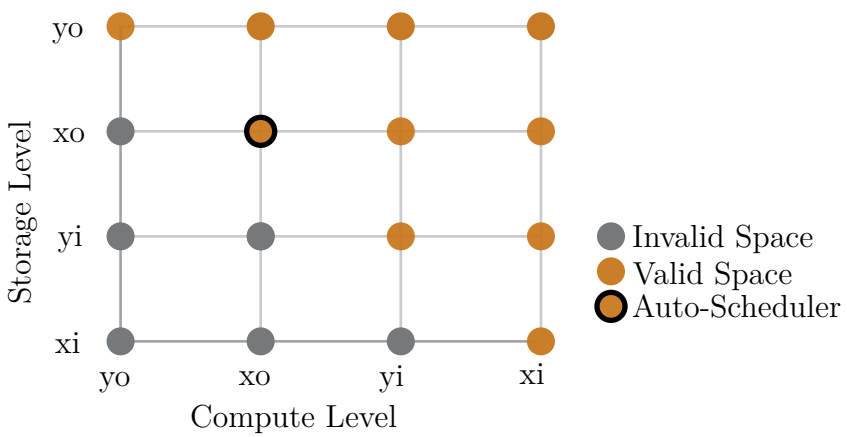


Figure 5.6: Fusion solution space for the blur example

Our method enables fast exploration of this new design space. We will show that through the use of heuristics, we can quickly prune the space down to a single solution (e.g. out of the 10 valid schedules in Figure 5.6, we only need to evaluate one). This is achieved by automatically eliminating most uninteresting schedules which are pareto dominated by other more efficient solutions. Dominant schedules are considered the following:

1. Schedules that offer more reuse with the same buffer requirements. e.g. consider the schedules (`yi,yi`) and (`yi,xo`). The second schedule provides more reuse while the sliding window optimization allows for the same memory requirements.

2. Schedules that offer the same amount of reuse with the smaller buffer requirements. e.g. schedules (`xo,xo`) and (`yi,xo`) as explained in the previous example.

The final solution is then evaluated through a cost function in order to pick the tile sizes. The analysis has to be repeated for different loop permutations, since that leads to a different design space with different solutions.

5.4 Proposed method

In this section we present each major step in our optimization flow. We follow a grouping-then-tiling technique that only attempts to split the pipeline into smaller segments if the initial solution does not fit within the memory constraints. More specifically, Section 5.4 discusses the algorithms responsible for choosing the compute and store level of a stage inside a pipeline (or a segment of a pipeline). Section 14 presents the tiling analysis that determines the proper tile sizes for a pipeline/segment. Section 11 demonstrates the procedure that is followed in order to split a pipeline into smaller segments. Some final optimizations are discussed in Section 11and an overview of the optimization flow is given in Section 11

Fusion strategy

This subsection introduces the analysis and heuristics that are used to determine which single point of the fusion space should be chosen for further evaluation. More specifically, this section addresses the problem of choosing the computation and allocation level of each stage inside a pipeline (or a segment of it).

As already mentioned, our goal is to eliminate inefficient schedules without evaluating their costs. Algorithms 4 and 5 show the procedure that is followed in order to accomplish that. In detail, Algorithm 4 takes a pipeline (P) as an input which can be either the whole DAG of the initial pipeline or a sub-graph of it, and identifies the compute and store level for each stage (K) in P. On the other hand, Algorithm 5 attempts to inline stages with trivial computational costs.

The pipeline can be described as a DAG of m connected nodes such that $P = \{K_0, K_1, ..., K_m\}$, where K_m is the output/final stage of P. Furthermore, in order to be able to describe all necessary dependencies

between the nodes of the DAG, as well as the schedule of each stage, we perform the following definitions for all $i \leq m$:

- A linearly ordered set $D_i = \{xi_0, xi_1, ..., xi_{n1}, xo_0, xo_1, ...xo_{n2}\}$ which represents the tiled loop nest of K_i where the xi and xo are the intra and inter-tile loop indices respectively.
- A list of tuples $W_i = \{Y_0, Y_1, ..., Y_l, ..., Y_t\}, 0 \leq t < m$, with $Y_l = \{K_l, I_l\}$, K_l the consuming stage and $I_l = \{E_0, E_1, ..., E_z\}, z$ the number of unique indices in the loop nest of K_i, $E = (x, v), v \epsilon \mathbb{N}$, while $x \epsilon D_{K_l}$ is the dimension where the dependency exists and v the amount of overlap.
- A list of producers $L_i = \{K_0, K_1, ..., K_p\}, 0 \leq p < m$
- A tuple $S_i = (x_{compute}, x_{store}), x \epsilon D_m$, which will partially define the final schedule and where $x_{compute}$ and x_{store} are indices of the output domain.

Algorithms 4 and 5 determine the compute and store level of a stage inside a pipeline. More specifically, Algorithm 4 first checks whether a stage has overlap with any of its consumers (whether any of its values can be reused across iterations). If that is true, then the algorithm searches for the dependency index with the highest intra-tile order. If that index is also present in the loop nest of the output stage and is not the innermost one, then it is set as the compute level of the stage. Its store level is set to one level higher in order to benefit from sliding-window opportunities and ensure that all possible reuse is captured as explained in Section 5.3. While there might be cases where maximum folding and therefore even smaller buffer requirements can only be obtained by either moving the store level higher or the compute level lower than what Algorithm 4 considers, the above heuristics allow us to quickly choose a single point in the design space while ensuring maximum reuse. Furthermore, if the chosen index corresponds to the innermost intra-tile loop of the loop nest or is a reduction dimension that offers full reuse, then the compute level is also set to one level higher. This decision is made in order to better exploit spatial locality and hardware prefetching in cases where the compute level is set to the column (as also explained in the motivational example) or vector index (which often corresponds to the innermost intra-tile loop) of a loop-nest and avoid redundant computation in reductions that have full overlap with their consumers. If on the other hand, the chosen index is not found in the output loop nest, then it means that there is no direct overlap between the output domain and the stage that is being scheduled, but dependencies exist across intermediate stages. Its compute and store levels are therefore set to the innermost intra-tile loop of the output stage of that segment. The above method allows us to quickly choose

Algorithm 4: Stage Fusion Analysis

Input: $P, D_0, D_1, .., D_m, W_0, W_1, ..., W_m$
Output: $S_0, S_1, ..., S_m$

$i \leftarrow m$
repeat
  if $W_i \neq \emptyset$ **then**
    $c_i \leftarrow max(\bigcup_{Y \epsilon W_i} select(\bigcup_{E \epsilon I} select(x, E), Y))$
    if $c_i \epsilon D_m$ **then**
      $s_i = next(D_m, c_i)$
      if $c_i = min(D_m)$ ***or*** $is\_reduction(c_i)$ **then**
        $c_i = next(D_m, c_i)$
      $S_i \leftarrow (c_i, s_i)$
    end
    else $S_i \leftarrow (min(D_m), min(D_m))$
  end
  else $S_i \leftarrow (inline, -)$
  $i \leftarrow i - 1$
until $i = 0$

Algorithm 5: Inline Trivial Stages

Input: $P, L_0, L_1, .., L_m, S_0, S_1, ..., S_m$
Output: $S_0, S_1, ..., S_m$

$i \leftarrow m$
repeat
  if $S_i \neq (inline, -)$ **then**
    if $trivial(K_i) = True$ **then**
      for j *in* $0 \leq j \leq p_i$ **do**
        if $!trivial(Kj)$ && $S_j = (inline, -)$ **then**
          $S_j \leftarrow S_i$
        end
      end
      $S_i \leftarrow (inline, -)$
    end
  end
  $i \leftarrow i - 1$
until i=0

a compute/store level for each stage of the group/segment. Furthermore, as explained in Section 5.3, moving the compute level even higher (to be the same as the store level) would lead to dominated solutions that require larger buffers only to achieve the same reuse. Finally, if a stage has zero overlap with its consumers, then its computation is inlined. We should note that all stages are scheduled with respect to the output stage of the pipeline. This eliminates any possibilities of nested loop fusion, which would add extra recomputation between the loops. We also introduce notation for two helper functions (*select* and *next*) where:

- *select* returns the first subset (or element) denoted by the first argument that belongs to the tuple (or pair) denoted by the second argument.
- *next* also takes two arguments and returns the element which belongs to the ordered set specified by the first argument and the position equal to the second argument plus one.

Algorithm 5 uses the partially defined output schedules of Algorithm 1 and attempts to inline the trivial stages of the pipeline. A stage is considered trivial only if its computational cost is equivalent to its load cost (similar to the analysis followed by the Halide Auto-Scheduler) and only if all of its producers are non-inlined. After finding that a stage is trivial, the algorithm checks if any of its direct producers that were previously inlined (due to zero overlap), may now have to be scheduled. In such a case, the compute and store levels of the newly found non-trivial stage is set to be the same as the ones of the now inlined stage.

Group tiling

This section presents the analysis that chooses a proper tiling for a given Pipeline/group. Algorithm 6 shows the procedure in detail.

The algorithm requires a pipeline (or segment) P as well as the linearly ordered set D_m as inputs. The latter represents the ordering of the tiled loop nest of the output stage and can initially be any (arbitrary) permutation of the loop nest as long as the intra-tile loops (xi) do not mix with the inter-tile ones (xo). The cost of evaluating each stage without any recomputation ($C_{Ki}^{root}, S_{Ki} = (root, root)$) is computed in order to be able to calculate the amount of recomputation for a given schedule. While this factor will be constant and will not alter the analysis within a group, it can affect the total cost of the pipeline when a different grouping is considered (and different stages have zero recomputation). As explained in the previous sections, the various discrete points in the fusion space depend on the loop permutation of the pipeline. As a result, the algorithm needs to try all possible intra and inter-tile loop permutations. Since the

Algorithm 6: Tiling Analysis

Input: P, D_m
Output: $Tm_0, Tm_1, ..., Tm_{n1}, P_{fused}$
for *all i in* $0 \leq i < m$ **do** Evaluate C_{Ki}^{root}
repeat
  Perform Stage Fusion Analysis
  Inline trivial Stages
  repeat
    $C_{total} \leftarrow 0$
    for *all i in* $0 \leq i < m$ **do**
      $C_{total} += w \cdot Cl_{Ki}^{Km_{ci}^{si}} + Cp_{Ki}^{Km_{ci}^{si}} - C_{Ki}^{root}$
  until *all valid tile sizes evaluated*
until *all valid loop permutations evaluated*

Tm_n	Tile Size in nth dimension
$Cl_{Ki}^{Km_{ci}^{si}}$	Load Cost of K_i
$Cp_{Ki}^{Km_{ci}^{si}}$	Compute Cost of K_i
C_{Ki}^{root}	Root Compute Cost of K_i
C_{total}	Total Cost of all stages
Bm_n	Problem Size in nth dimension
w	Relative cost of load operation

Table 5.1: Notation

number of possible schedules explodes for large pipelines with multiple nested loops (such as convolutional neural networks) we do not attempt to interchange the kernels or other loops that only perform a few iterations. This decision allows us to easily eliminate the loop overhead in many cases by unrolling those loops. For each possible loop permutation we perform the fusion analysis described in Algorithm 4 and then attempt to inline any trivial stages (Algorithm 5). At this point we can evaluate all relevant costs presented in Section 5.3 for each stage of the pipeline individually, for an arbitrary tiling dimension. We iterate over all possible tile sizes that fit into specific constraints:

- The tile size of the innermost intra-tile dimension (which is not part of the kernel) has to be a multiple of the cache-line size, as well as a multiple of the native vector width.
- The tile size of the outermost inter-tile dimension has to fulfill:

$$\frac{Bm_{no}}{Tm_{no}} \geq N_{threads} \tag{5.14}$$

- The tile size in a dimension where a dependency exists has to be at least as large as the amount of maximum overlap in that dimension such that, if x is the dimension of interest then:

$$Tm_x \geq max(\bigcup_{Y \epsilon W_i} select(\bigcup_{E \epsilon I} select(v, E), Y)) \tag{5.15}$$

In detail the first constraint is imposed in order to maintain vectorization in conjunction with spatial locality across cache lines. The second constraint ensures that the final schedule will have enough parallelism to utilize the multi-threaded aspects of the target architecture. The third constraint avoids invalid tile sizes that would lead to redundant computations without extra buffer benefits (since due to the inter-stage dependencies the memory allocation would be at least equal to the amount of overlap anyway). Finally, we do not consider tiles where the total footprint of stages without folded storage (compute and store level are the same) does not fit into the L2 cache. This constraint ensures that values that will be immediately consumed and cannot be reused in future iterations stay local. We calculate the costs defined in Section 5.3 for each stage individually using a weighted cost function and sum them together to compute the total cost of the pipeline. Our cost function uses the load cost of a stage ($Cl_{Ki}^{Km_{ci}^{si}}$) multiplied by the relevant overhead of a load operation compared to a computation (w) plus the amount of recomputation of that stage ($Cp_{Ki}^{Km_{ci}^{si}}$ - C_{Ki}^{root}). The combination of loop permutation, fusion choice and tile size ($Tm_0, Tm_1, ..., Tm_{n1}$), that minimizes the total cost (C_{total}) of the pipeline is chosen as the final schedule.

Stage grouping

If the memory footprint of the final schedule is larger than the size of the last-level-cache, the pipeline is split into segments and each segment is scheduled independently of the others. Given the fact that current multi-core architectures contain caches of many MBs in size that will likely fit many stages, our strategy attempts to reduce design time by only attempting to split the pipeline if the initial solution (where all stages are either fused or inlined into the output stage) does not fit into the cache. The memory footprint of the pipeline is equal to the amount of memory required/allocated for all intermediate stages of the pipeline (or segment). Data from intermediate stages will either be stored in order to be reused in future intra-tile iterations (in circular/folded buffers) or will immediately be consumed in the current intra-tile iteration and are not needed afterwards. Buffers in the former category are folded down to the maximum amount of overlap (only in the dimension specified by the compute level of the stage) and their total size needs to fit into

Algorithm 7: Group Stages

Input: P, D_m
Output: $H_0, H_1, ..., H_w$

$n_P \leftarrow 0$
for *all* i *in* $0 \leq i < m$ **do**
  if $size(W_i) > 1$ **then**
    $S_i = \{root, root\}$
    $H_{n_P} \leftarrow \{K_{0i}, ..., K_{ti}, K_i\},\ n_P++$
    $P \leftarrow erase(\{K_{0i}, ..., K_{ti}, K_i\})$
  end
end
$H_n \leftarrow P$
for j *in* $0 \leq j \leq n_P$ **do** $SplitSegment(H_j)$

Algorithm 8: Split Segments

Input: H
Output: $H_0, H_1, ..., H_w$

$TilingAnalysis(H)$
if $wset_H > csize$ **then**
  $n_H \leftarrow max\_split$
  repeat
    $S_{n_H} = \{root, root\}$
    $H_n \leftarrow \{K_0, ..., K_n\},\ n_H++$
    $H \leftarrow erase(\{K_0, ..., K_n\})$
    $TilingAnalysis(H_n)$
    if $wset_{H_n} < csize$ **then** $SplitSegment(H)$
  until $wset_{H_n} > csize$
end

the last-level-cache for future use, while buffers that will not get folded need to fit inside the L2 cache (such that their data stays local between production/consumption). All buffers are calculated based on the areas required (allocated) by the compiler for a given schedule.

Algorithms 7 and 8 show the steps that are followed in order to split the pipeline P into non overlapping segments $(H_0, H_1, .., H_w)$. Algorithm 7 takes the initial pipeline as an input and first checks if any stages have more than one consumers. In that case, these stages form a new pipeline, along with their producers and are erased from the initial DAG. This is done in order to limit the design space and enable faster optimization runtime. While as a result we may end up end up with multiple smaller segments in some pipelines, we did not notice any significant performance degradation due to this fact. Further investigation of performance benefits that may be captured by merging those smaller segments is left as future work. During the next step, we attempt to schedule all new pipelines,

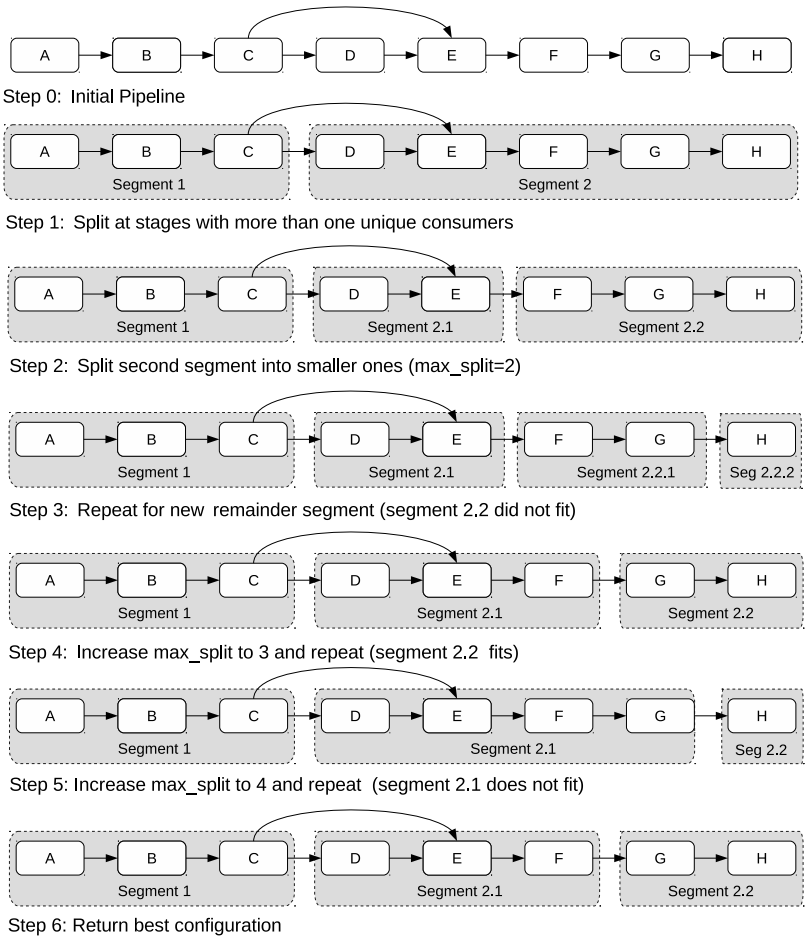


Figure 5.7: Pipeline segmentation strategy

along with the remainder of the previous step (remaining stages of the original pipeline). Algorithm 8 checks whether the footprint of the new segments is still larger than the available cache size, and then recursively splits those into smaller segments using the following process. Starting from the nth stage of the pipeline, where n is set to *max_split* (an integer value which controls the minimum size of a segment) we schedule all stages up to the nth. If the segment fits, then we attempt to schedule the remaining stages by recursively repeating the same algorithm. After having evaluated all possible configurations for a specific n, we increase it by one and the process is repeated until the working set of the segment

does not fit any more. This ensures that we skip configurations with invalid segment sizes. Each (unique) valid solution generated by Algorithm 8 where all stages of the original pipeline (P) have been successfully scheduled is cached and the sum of all independent sub-pipelines' cost is evaluated. The configuration that results in the minimum (summed) cost is chosen as the final solution. We should also note that if the initial value of *max_split* is set to one, then the algorithm will evaluate all valid grouping configurations for the pipeline.

Evaluating all possible configurations may require an extensive amount of time for larger pipelines (such as deep neural networks). Our method reduces the runtime of the grouping process by eliminating non-interesting segmentations. This is achieved in two ways:

- Upon identifying that the memory footprint of a segment is larger than the available cache size, we do not attempt to fuse more stages into the same segment. This choice can be explained as follows: A segment with a memory requirement larger than the available cache size, will only grow larger if more stages are included into it, especially if the newly included stage has extra dependencies.

- We do not attempt to split the final segment of a pipeline into smaller ones, since that would only add external load costs from the previous root stages to the subsequent consuming ones. This choice allows us to significantly reduce the time needed to find the final configuration especially in the context of modern multiprocessor architectures with large cache sizes.

The steps followed for an example pipeline can be seen in Figure 5.7.

Final optimizations

Upon finding the final configuration of a pipeline, we have groups of stages with a specified tiling and loop permutation per group. We vectorize the innermost intra-tile loop of a group that is not part of a reduction (or a kernel) and parallelize its outermost inter-tile loop among the platform's threads/cores as explained in Section 14 (Eq.5.14). However, we have not yet considered any changes in the permutation of individual stages within a stage. We optimize the loop nest of each producing stage within a segment through loop interchange that improves reuse distance by reordering loop indices with minimum strides to be innermost. Moreover, the loop that corresponds to the compute level of a stage is always set as outermost, since that loop will always iterate once (or once plus the equivalent overlap with its consumer) and would add extra loop overhead in any other position.

Optimization flow overview

Figure 5.9 shows the optimization flow for an input pipeline along with all iteration steps involved, while Figure 5.8 shows the steps followed in order to schedule each segment (or the initial whole pipeline if it fits in the cache).

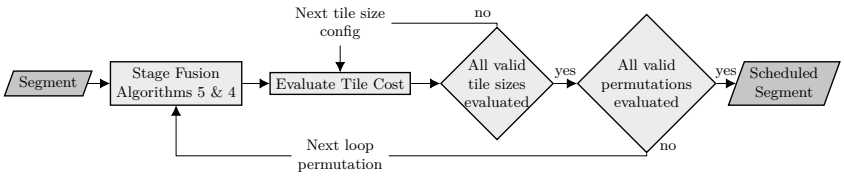


Figure 5.8: Optimization flow for a pipeline segment (Algorithm 6)

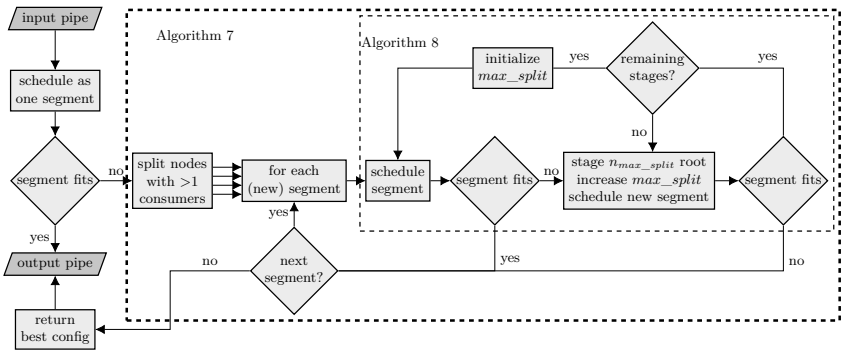


Figure 5.9: Optimization flow for an arbitrary pipeline

In detail, for all valid permutations of the tiled loop nest, Algorithm 6 calls Algorithms 4 and 5 in order to determine the compute/store levels of each stage. It then evaluates the total cost of the pipeline for all valid tile sizes and the combination of D_m, T_m (loop permutation and tile sizes respectively) that minimizes C_{total} is chosen. If the memory footprint of the final schedule is larger than the constraints imposed by the last level cache of the target system, then Algorithms 7 and 8 are used to split the pipeline into smaller segments, with Algorithm 6 (and subsequently Algorithms 4 and 5) used again in order to schedule each new segment. Every valid configuration (where all stages of the original pipeline have been successfully scheduled) is cached in order to be evaluated at the end of the process. The segmentation/configuration that minimizes the total cost of the original pipeline is chosen as the final, now scheduled pipeline.

5.5 Experimental results

This section demonstrates the results obtained across a wide variety of image processing applications on three different architectures.

Experimental setup

The architectural details of each platform used in the experiments are listed in Table 5.2. Table 5.3 provides a description of each benchmark along with the problem size considered. The optimization/compile time of all benchmarks is performed within seconds. Most of the descriptions were found in [52]. The chosen benchmarks include image processing pipelines used in [52], the pyramid blending algorithm used in [32], as well as a popular recent Deep-Neural-Network used for single image super-resolution (VDSR) that was introduced in [37]. All problem sizes are chosen to be the same as the ones found either on the official Halide repository on GitHub [26] or as the ones used in [32].

Platform	**LLC size (MB)**	**L2 cache size (KB)**	$N_{threads}$
Intel i7-6700	8	256	8
Intel i7-5930K	12	256	12
ARM Cortex A15	2	512	4

Table 5.2: Platform Features

In the following graphs, the manual implementations refer to the manual schedules found in the Halide repository (the only exceptions being the pyramid benchmark, the manual schedule of which was found in [32] as well as the VDSR network which we implemented in Halide). The PolyMage-A and PolyMage-DP implementations refer to the results replicated using the artifacts and instructions provided by the authors in [32] and [65]. However, implementations were provided for only six benchmarks, which are also the ones considered in [32].

We compare our results to the equivalent ones produced by the other methods: since all of our applications are implemented in Halide, we can use the Halide Auto-Scheduler [52] to produce schedules for all benchmarks. Each benchmark is executed 100 times and the average execution time per run is measured. This process is repeated multiple times per benchmark and the minimum average among those is used as the final average execution time. Furthermore, we properly adjust the optimizations parameters of both the Auto-Scheduler and PolyMage before our experiments for the solutions to be tuned to the target platforms. Since PolyMage cannot explicitly vectorize loops (unlike Halide) the performance of the PolyMage implementations is highly influenced by the efficiency of the auto-vectorizer of the back-end compiler [32]. Finally, the problem

Benchmark	Description
blur 2 stages 6400x4800	Simple two-pass 3x3 blur filter
bilateral 5 stages 2560x1536x3	Fast bilateral filter using the bilateral grid [14]. Constructs the grid using a histogram reduction, followed by stencil and sampling operations.
unsharp 6 stages 2560x1536x3	Enhances local contrast by smoothing an image with a small support gaussian and subtracting it from the original to isolate the high-frequency content, which is then combined with the original image.
harris 13 stages 1920x1024x3	Implementation of the popular harris corner detection algorithm [28] which combines multiple stencils and point-wise operations.
camera 30 stages 2560x1936x3	The Frankencamera pipeline for processing raw data from an image sensor into a color image [3]. The pipeline performs hot-pixel suppression, demosaicing, color correction, gamma correction, and contrast.
interpolate 52 stages 1536x2560x3	Interpolation of image pixel values using an image pyramid for seamless compositing, based on the newest healing brush in Photoshop. Pyramid construction deals with image data at multiple resolutions and creates chains of stages with complex dependencies
laplacian 99 stages 1536x2560x3	A local Laplacian filter: an edge-aware, multi-scale approach for enhancing local contrast [62]. The pipeline builds multiple image pyramids with complex dependencies and performs data-dependent sampling.
lensblur 74 stages 1536x2560x3	Given a rectified stereo pair of images, produces a synthetic shallow-depth-of-field image. It first solves for depth by constructing and filtering a cost volume [77] using a convolution pyramid [22], then renders the synthetically defocused image by randomly sampling the source image over a virtual aperture.
nlmeans 13 stages 614x1024x3	Fast non-local means image denoising using the method of [20]. Computes a 7x7 image blur with weights determined by 7x7 patch similarity
maxfilter 9 stages 1920x1024x3	Computes the maximum-brightness pixel within a circular region around each target pixel. Uses a precomputed table of differently-sized vertical max filters to reduce complexity from O(radius2) per output pixel to O(radius).
pyramid 52 stages 1920x1024x3	Pyramid blending that blends two input images into one using a mask and a Laplacian pyramid of 4 levels.
VDSR 24 stages 256x256x64	VDSR (Very Deep network for Super-Resolution) [37] is an end-to-end network with 20 convolutional layers for single image super-resolution.

Table 5.3: List of benchmarks. Table extended from [52]

size used in [32] for the harris and unsharp benchmarks differs from the one in the Halide repository. We therefore repeat the experiments for this problem size as well and the results of this comparison can be seen in Table 5.4. Halide was built using llvm 4.0.0, while the PolyMage implementations were compiled using icpc on the i7-6700 platform and gcc on the i7-5930K and ARM platforms.

At this point it is important to emphasize that, as seen in Figures 6.5 and 5.9, all of the proposed algorithms are tightly coupled. As explained in the motivational example of Section 5.3, sliding windows and circular buffers allow for tile sizes that would otherwise be impossible to consider (e.g. large tile strips that otherwise would never fit into the local buffer constraints imposed by the cache size). As a result, evaluating each algorithm independently is not possible; they should all be considered together

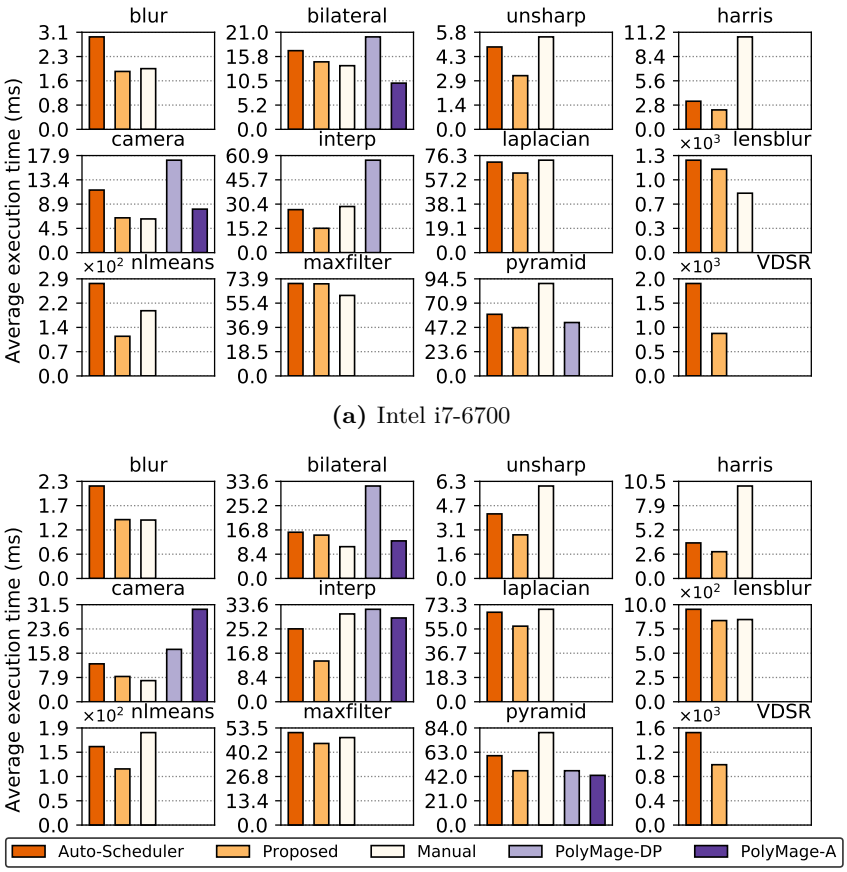
Performance results

Figure 5.10 shows the average execution time (in ms) for each benchmark on the two Intel platforms listed in Table 2. The results for the harris and unsharp benchmarks on the problem size of the PolyMage implementations can be seen in Table 5.4.

Our schedules outperform the Auto-Scheduler solutions in almost all cases, with the Laplacian benchmark being the only exception on the Intel i7-5930K, where the difference in execution time is still within $\approx 2\%$. We noticed that while the initial Auto-Scheduler paper optimizes for L2 cache size, the currently used and updated one is targeting the shared last level cache. We conducted multiple experiments for both choices and noticed that while some benchmarks experience a slight performance improvement when the memory footprint constraint is set to the size of L2, other ones suffer a dramatic performance degradation. As a result, we choose to use the currently advised method of optimizing for last level cache in the results. Finally, our schedules are also comparable or even better than the manual ones in many cases.

PolyMage-DP performance is similar to the Auto-Scheduler in almost all cases. The constant updates and focus on the Auto-Scheduler by the Halide community may explain the difference in the results presented here and the ones in [32] between the two methods. The efficiency of auto-tuned PolyMage-A solutions vary per benchmark and platform: the raw camera and bilateral grid implementations of the autotuned solutions on the intel i7-6700 are close to (or slightly better than) the manual Halide schedules. On the other hand, they are much less efficient compared to the other implementations of the harris filter on both Intel platforms.

The results for the ARM Cortex platform can be seen in Figure 5.11 and Table 5.4, where a similar pattern can be discerned. Our schedules



(a) Intel i7-6700

(b) Intel i7-5830K

Figure 5.10: Performance Results on the two Intel Platforms

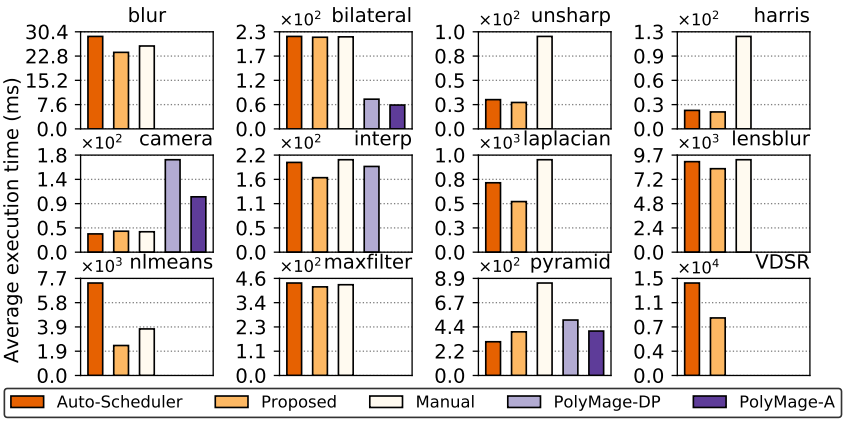


Figure 5.11: Performance results on the ARM platform

Method	Intel i7-6700		Intel i7-5930K		ARM A15	
	harris	unsharp	harris	unsharp	harris	unsharp
PolyMage-A	45	27	27	27	377	254
PolyMage-DP	15	21	21	21	304	388
Auto-Scheduler	16	22	14	22	164	206
Proposed	11	20	10	20	171	189

Table 5.4: Average execution time (ms) for harris & unsharp benchmarks - problem size 4256×2832

outperform both the manual and autoscheduled ones with the largest differences observed in the interp, laplacian and VDSR benchmarks. The performance of the PolyMage solutions varies per benchmark. For example, it performs significantly worse than the Halide solutions on the camera pipeline, slightly better than both the Auto-Scheduler and the manual Halide schedule on the interp benchmark, and much faster than all Halide solutions in the bilateral pipeline. The main reasons behind this result are the differences in the functional description of the pipeline between the Halide and PolyMage implementations: Halide uses a built-in linear interpolation function that performs more complex computations than the PolyMage implementation of it. Upon forcing Halide to use a simpler approach, performance was improved by up to 40% in all three cases (Auto-Scheduler, Manual and Proposed). Furthermore, Halide requires all expressions used as indices in functions/stages to be bounded, and therefore performs extra clamping in two stages for the compiler to be able and derive the bounds of the equivalent producers. These extra computations are the main bottlenecks in the performance of the Halide

bilateral pipeline on the ARM platform. However, finding an efficient description is outside the scope of this work.

Finally, in order to test the efficiency of our grouping strategy (Algorithms 7 and 8), we repeat the experiments for the VDSR network and investigate the performance for various problem sizes. We choose VDSR since it consists of sequential stages where each subsequent stage consumes the output of the previous one (except for the input image which is consumed twice). This benchmark is therefore a good candidate for such an experiment since various problem sizes will lead to different tiling choices and therefore different memory footprints, which, due to a constant memory constraint will require new segmentations. The results of this experiment on the Intel i7-6700 platform for 5 different dimensions of the output image are presented in Figure 5.12. Our schedules perform more than 2x better than the equivalent Auto-Scheduled solutions for large problem sizes.

In order to demonstrate the robustness of our method, the same experiment was conducted on another platform (with an Intel i7-6560U processor) once with the hardware prefetcher enabled and once with the hardware prefetcher disabled. The experiments followed a similar trend as in Figure 5.12 when comparing the two implementations. Furthermore, the performance degradation when the hardware prefetcher was disabled in our solutions was close to 20% while for the Auto-Scheduler solutions it was more than 2x. Upon further investigation, we noticed that the loop permutation chosen by the Auto-Scheduler (which attempts to reorder loops based on their stride, i.e. placing the loop with the smallest stride innermost) interleaves the column, row and kernel dimensions, limiting the amount of spatial reuse that can be captured in the process. This incurs a high penalty when the hardware prefetchers are disabled. On the other hand our proposed method does not reorder loops with low iterations (similar to the 3x3 convolution kernels) and only attempts to exploit prefetching when determining the tile size dimensions (Algorithm 6). Setting the kernel inner to the column and row dimensions allows data to stay in the local caches (or even registers) before they are reused. As a result, self-spatial reuse can still be exploited across kernel iterations and this explains why our schedules do not suffer as much when hardware prefetchers are disabled.

Finally, based on the above results (Figures 5.10-5.12), we can observe that larger pipelines with multiple stages such as the interp, laplacian and VDSR benchmarks benefit the most from our schedules, where sliding window opportunities are easily captured, buffers are folded down to smaller memory footprints, and new tiling opportunities are considered. Moreover, even in cases where the pipeline does not offer such opportunities (e.g. bilateral, nlmeans), our solutions remain similar to (or in many cases even better than) both the Auto-Scheduler, and the manual

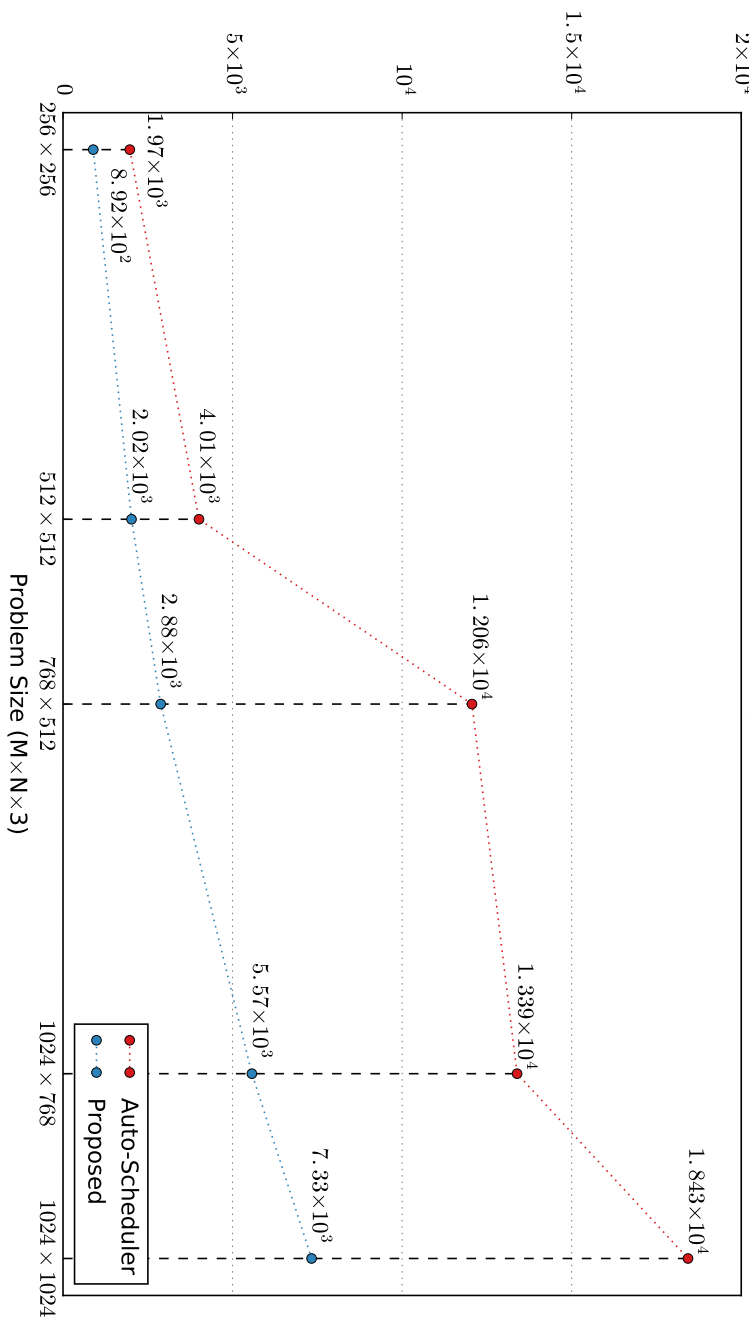


Figure 5.12: VDSR average execution time (ms), Intel i7-6700

solutions.

5.6 Summary

In this chapter we presented a novel platform-aware algorithm for the optimization of image processing pipelines running on multi-core CPU based architectures. We show that our method captures solutions of the design space which were not covered in previous state-of-the-art techniques by effectively considering combinations of loop tiling, interchange and stage fusion with independent computation and allocation per stage. Our model takes into account multiple architecture specific parameters such as multithreading, vectorization and hardware prefetching. We evaluate our proposed method across a variety of image processing applications implemented in the Halide DSL and compiler and compare it to both previous state-of-the-art techniques that target the Halide and PolyMage DSLs, as well as manually optimized Halide solutions. Experimental results show significant average performance improvements compared to previous related work as well as the manually optimized implementations of Halide pipelines.

6

Efficient scheduling for GPGPUs

6.1 Introduction

Code generation for image processing pipelines remains a challenging problem due to the increasing need for high performance as well as the complexity of modern hardware platforms. Image processing applications usually require developers to have expert knowledge of both the algorithm that needs to be implemented, as well as the behavior of the underlying platform that will be used. These platforms are usually of heterogeneous nature, with a multi-core CPU with SIMD extensions acting as a host and a dedicated or onboard GPU unit acting as an accelerator. In the context of image processing pipelines, GPUs can often be more than an order of magnitude faster than a traditional CPU architecture [6]. As a result, developers have to spend a lot of manual effort in order to provide efficient implementations of each pipeline and manage host and accelerator communication. This effort usually has to be repeated each time an algorithm gets designed or modified or a new target platform has to be used.

Modern compilers and languages attempt to alleviate this issue by using libraries with predefined manual optimized implementations of the most popular image processing algorithms [69], or by allowing developers to specify their applications in a general-purpose, high-level language. Such an example is the Julia language which uses the LLVM CUDA backend [10] to generate code for NVIDIA GPU architectures, enabling

easier offloading for applications through a general purpose language.

Domain specific languages (DSLs) have also proven to be invaluable for efficient GPU code generation. These languages often incorporate a syntax that allows for both quicker exploration of the optimization space, as well as offloading parts of the application to the GPU extensions of the platform.

However, unlike the CPU platforms that we studied in the previous chapters, GPU based architectures impose strict constraints on the schedule that make many schedules invalid. Such constraints are the maximum number of threads per block as well as the maximum shared memory per Streaming Multiprocessor (SM) which may vary per architecture or Compute Capability (although usually some parameters remain constant). Developers have to keep such constraints in mind when determining the proper tile sizes for their implementations, as they can have a severe impact on performance.

In this chapter we make the following contributions:

1. We extend the current autoscheduler of Halide master [27] with a new analytical cost model that considers GPU specific parameters when generating optimization schedules.

2. We perform fast design space exploration by eliminating uninteresting and invalid configurations without evaluating the equivalent schedules while ensuring that the final schedules meet all constraints imposed by the platform.

3. We introduce a set of heuristics that enable nested fusion, extending to possible solutions outside the traditional optimization space where computation of each group's intermediate stages is always placed relative to the group's output stage and always set to the block level of the consuming loop nest. Nested fusion reduces the shared memory requirements of the schedule configuration, allowing previously computed values to stay in local registers.

4. We evaluate our approach across various applications and test it on two different CUDA based platforms. Experimental results show a significant performance improvement over previous attempts (over 2x) at automatic GPU scheduling while our solutions remain competitive or are even better than the manual schedules written by Halide experts (around 10% faster).

5. We implement our method as an extension over the previous CPU autoscheduler reusing parts of its analysis in an effort to ensure compatibility with the current Halide versions.

The rest of this chapter is organised as follows: Section 2 discusses related work. Section 3 establishes the search space and scope of our approach. Section 4 presents the proposed method which we name Auto-GPU. Section 5 demonstrates the experimental results that were obtained. Possible future work and conclusive remarks are discussed in Section 6. This work was published in [79].

6.2 Related work

This section discusses related work on optimization strategies for image processing applications and GPU code generation. We divide this section into three parts: *a)* Common loop transformations used to optimize loop nests of image processing pipeline stages, combinations of which are often used in automatic scheduling attempts, *b)* prior automatic scheduling for Halide pipelines and their limitations for GPU schedule generation, *c)* other optimization strategies for efficient GPGPU code generation in the image processing domain as well as general purpose compilers.

Loop transformations

Most scheduling approaches for image processing pipelines focus on a combination of loop transformations and optimizations to exploit parallelism and avoid costly memory accesses. The most common of these transformations are loop fusion and tiling. Loop fusion can enable other optimizations by increasing locality between production and consumption of intermediate values [48]. In the context of GPU code generation, fusion can help avoid global memory accesses by merging multiple kernels, increasing performance in memory bound applications by introducing redundant computations and ensuring that data used across consecutive, merged stages of the pipeline remain in the shared memory or local caches [71, 90, 91].

Loop tiling is often used alongside kernel fusion to exploit parallelism and enable both spatial and temporal reuse across stages. Tiling has been extensively used to optimize applications in the image processing domain targeting either CPU or GPU based architectures. Most such approaches focus on the optimization of affine programs, using what is commonly called an overlapping tiles analysis that executes one thread block per tile, interleaving the computation of producing stages at the block level of the consuming loop nest and storing all pixels computed into the shared memory [30, 75]. Tile sizes are often chosen through a cost function that attempts to model the performance of the underlying architecture while taking into account key CPU parameters (hardware prefetching, SIMD vector units, number of cores) or GPU specific parameters (register

and shared memory usage, achieved occupancy) [67, 82]. Our model considers even more architecture specific parameters, such as the thread block size, the total number of global memory accesses, active streaming multiprocessors and threads per stage while extending the kernel fusion space by allowing computation of producing stages to be placed at depths lower than the block level, reducing the shared memory requirements of the schedule and allowing values to be placed in the constant memory and registers instead.

Halide autoscheduling

Automatic scheduling for Halide pipelines has been investigated a number of times in the past. Halide originally used an autotuner [73] that was later replaced with a more optimized one that uses genetic algorithms in order to find an efficient schedule [4]. However, this approach was unable to converge to optimal solutions especially for complex large pipelines. An analytical heuristic based model was later introduced by Mullapudi et al [52] which uses an overlapping tile analysis along with a greedy grouping/merge algorithm, which enables fast exploration of the design space and generation of optimization schedules. Its search space is limited to tile sizes that are powers of two (8 to 256), stages can either be fully inlined (completely concatenating the statements of producers and consumers), computed in a breadth-first manner, or interleaved at the innermost inter-tile level of the group output (overlapping tiles). This method was extended by the Halide community and after having its cost model updated it is one of the supported autoschedulers in the Halide master [27]. While the original publication shows promising results on GPU architectures as well, that part of the scheduler was never integrated into the Halide master. We extend the Halide master scheduler with a new analytical model and analysis passes that enable (i) GPU schedule generation, (ii) a larger tiling and kernel fusion solution space than prior approaches, as well as (iii) schedule requirements that ensure that the final solution adheres to the constraints of the underlying hardware, all without sacrificing design/compile time.

Recent analytical models [81, 82] tried to extend the search space considered while attempting to model cache and hardware prefetching behaviors. The analytical model proposed by Sioutas et al [82] attempts to quickly generate efficient schedules through the use of heuristics while maintaining a larger search space (sliding window optimizations) compared to the one explored by both the Mullapudi et al [52] and Halide master [27] autoschedulers. However, both above models [81, 82] along with the associated heuristics were tuned to CPU behavior with large caches, and due to favoring sliding window optimizations, they are inca-

pable of exploiting the massive parallelism available on GPU architectures without sacrificing performance to thread synchronization overhead.

Finally, Adams et al [2] investigated a learned model that used random pipelines as training data in order to train a hybrid model for x86 multi-core CPUs. Its search space is much larger than prior non-autotuning attempts, but retraining and changes to the search space are needed for more efficient GPU-valid schedules. The authors report preliminary results compared to the Li et al scheduler [43] (29 to 33% faster) in CUDA based platforms but without yet retraining for GPUs. The latter [43] is the only functional autoscheduler for GPUs where tiling is applied to stages independently while stages themselves are either set to root (breadth-first implementations) or inline. While this can serve as a good baseline for an optimization schedule, stage/kernel fusion is not considered at all and solutions are often far from optimal or inferior to the manually tuned ones. This chapter extends the current CPU scheduler present in Halide master [27] with new heuristics and an updated analytical model that considers a broader space along with GPU parameters when generating optimization schedules for Halide pipelines.

Other DSLs and approaches

Besides Halide, there have been several other DSLs with GPU offloading support. Hipacc [51] is similar to Halide, as it can generate code for both multi-core CPUs as well as GPUs, while employing an autoscheduling framework in order to optimize the final code. This framework was recently extended in [72] with a novel kernel fusion model that tries to interleave computation of stages within the pipeline, but unlike our approach, loop tiling and interchange is not considered in the model.

Forma [74] is another DSL that behaves similar to Halide and offers an integrated autoscheduler as well. It supports CUDA (PTX) code generation and can cover a large set of image processing applications. However, its primary optimization strategy is to generate code in such a form that the back-end compiler (nvcc) will be able to efficiently optimize.

PolyMage [54] is a DSL comparable to Halide that relies on the polyhedral framework and also targets image processing pipelines. It combines autotuning with heuristics in order to automatically generate schedules. However unlike our approach, tile sizes are limited to powers of two and stages are always fused at the innermost inter-tile level of their consumers (overlapping tiles analysis).

Many other DSLs focus on optimizing tensor operations and only a subset of the algorithms found in traditional image processing applications. Such are for example TVM [15] and Tensor Comprehensions [87]. TVM focuses on local optimizations for single operators in the context of deep learning. Developers define an optimization space and

the compiler can automatically determine which optimizations should be applied. Tensor Comprehensions uses a front-end that is similar to the one used by Halide and its intermediate representation, but it replaces Halide's interval analysis with a polyhedral representation. Automatic optimization is enabled through autotuning across various possible schedules. Our method uses an analytical model and heuristics and does not require autotuning to generate efficient schedules, thus enabling faster design time and cross-compilation.

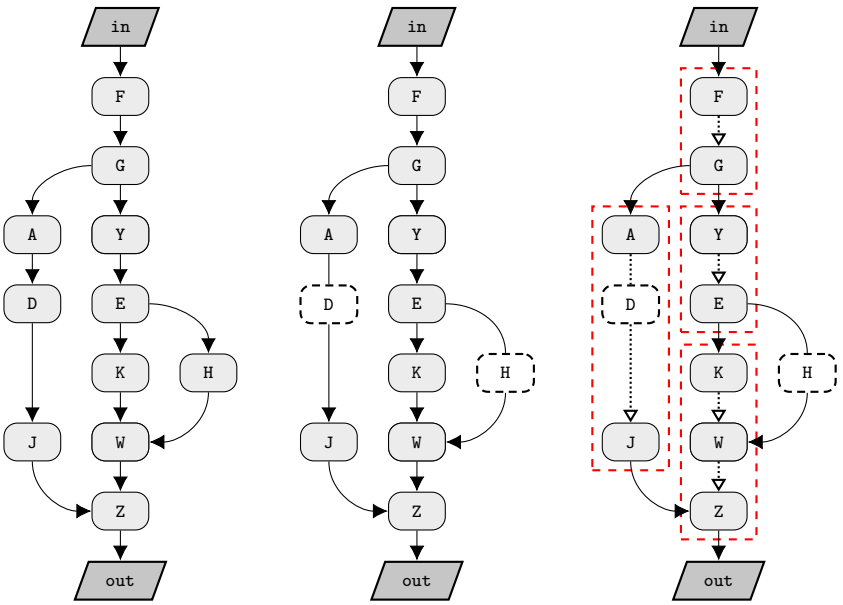
Outside the scope of DSLs, polyhedral compilers are often used to optimize image processing and tensor or stencil operations in GPUs [7, 67, 88]. These compilers employ polyhedral transformations in order to optimize affine programs. They aim to maximize parallelism through proper tile size selection but their application is limited to small-scale algorithms (i.e. GEMM based kernels) and they are unable to express many of the trade-offs explored in the above non-polyhedral DSLs like introducing redundant computations in an attempt to further increase locality.

6.3 Problem statement

Halide pipelines can be described as DAGs where each node of the graph represents a Halide function (`Func`), or stage of the pipeline. Each stage can be defined as a rectangular n-dimensional array, the allocation and size of which is determined/inferred by the compiler based on the dependencies with its consuming stages and the schedule. Each stage can have multiple dependencies on input images/buffers or other preceding stages.

As an example consider the graph shown in Figure 6.1a which represents an arbitrary pipeline consisting of 11 nodes or functional stages. In a naive implementation where the granularity of all stages is set to `root` each producer would be evaluated once and stored into a buffer to be consumed later. A naive implementation of this pipeline would require a separate CUDA kernel to be launched for each stage, storing all computed pixels necessary for the following stages in large buffers/arrays. In GPU terms that would result in multiple accesses to the global memory and the local caches (depending on the size of the buffers, as well as the dependencies between the stages). In other words, each edge would represent a number of global memory accesses equal to the allocation of the preceding node (buffer).

An example of such an implementation for part of the pipeline can be seen in Figure 6.2. The definitions of stages `K, H, W` and `Z` along with an example schedule that launches a separate CUDA kernel for each of them is seen in Figure 6.2a. The `compute_root` scheduling directive tells the compiler to fully compute a stage before moving to the next one. When



(a) Initial DAG: Each node represents a functional stage of the pipeline, each edge represents a number of global memory accesses. A different CUDA kernel is launched for each stage.

(b) Transformed DAG: Trivial stages are inlined into their consumers by concatenating their definitions. The number of global memory accesses can be reduced at the cost of redundant computation.

(c) Optimized DAG: The pipeline is split into segments in order to increase locality. Each group/segment corresponds to a different CUDA kernel.

Figure 6.1: Generic Pipeline Example: Trivial stages are inlined into their consumers before splitting the pipeline into smaller groups of stages which are assigned an optimization schedule.

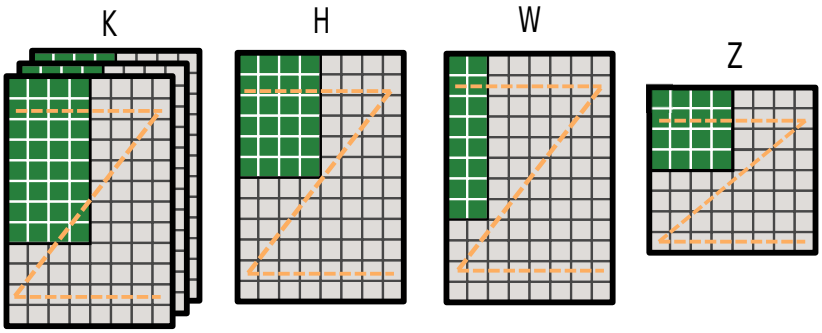
```
//function definitions for stages K,W,H,Z
K(x,y,c) = E(x,y) + E(x+1,y) + E(x+2,y)
H(x,y) = E(x,y) * 4
W(x,y) = K(x,y,0) + K(x,y,1) + K(x,y,2) + 2 * H(x,y)
Z(x,y) = W(x, y-2) + W(x,y-1) + W(x,y) +  W(x,y+1) +  W(x,y+2)

//GPU schedule
//tile the loops
Z.compute_root().gpu_tile(x, y, x_o, y_o, x_i, y_i, 4, 4);
W.compute_root().gpu_tile(x, y, x_o, y_o, x_i, y_i, 6, 8)
H.compute_root().gpu_tile(x, y, x_o, y_o, x_i, y_i, 8, 6);
K.compute_root().gpu_tile(x, y, x_o, y_o, x_i, y_i, 8, 4);
```

(a) Definitions and Example GPU Schedule of `KHWZ` stages: Compute granularity of all stages is set to `root`. Each stage is fully computed and stored in the global memory before moving to the next one. All memory transactions occur through the global memory and/or the local caches. A single kernel is generated for each stage.

```
//produce each stage in a separate
    kernel
allocate __global__ K[3*12*8]
<CUDA>gpu_block K.y_o
 <CUDA>gpu_block K.x_o
  <CUDA>gpu_thread K.y_i
   <CUDA>gpu_thread K.x_i
    for K.c
     K(..) = ...
allocate __global__ H[12*8]
<CUDA>gpu_block H.y_o
 <CUDA>gpu_block H.x_o
  <CUDA>gpu_thread H.y_i
   <CUDA>gpu_thread H.x_i
    H(..) = ...
allocate __global__ W[12*8]
for <CUDA>gpu_block W.y_o
 for <CUDA>gpu_block W.x_o
  for <CUDA>gpu_thread W.y_i
   for <CUDA>gpu_thread W.x_i
    W(..) = ...
allocate __global__ Z[8*8]
<CUDA>gpu_block Z.y_o
 <CUDA>gpu_block Z.x_o
  <CUDA>gpu_thread Z.y_i
   <CUDA>gpu_thread Z.x_i
    Z(..) = ...
```

(b) Equivalent pseudo-CUDA loop nest for stages `K,H,W,Z`: Allocation for each stage is moved to the global memory. A different kernel with variable grid dimensions is launched for each stage. The grid dimensions are controlled through the scheduling directives.



(c) Visual representation of the previous (`compute_root`) schedule: Each stage will launch a different CUDA kernel. The green pixels correspond to thread block dimensions of the CUDA grid that will be launched for each kernel controlled by the `gpu_tile` directive. Each stage is fully computed and all pixels are stored into the global memory before moving to the next one.

Figure 6.2: A naive implementation fully computes each stage in a different CUDA kernel and stores all data into the global memory.

paired with the `gpu_tile` command, the loop nest that corresponds to the surrounding stage will be tiled and the inner intra-tile loops will be mapped to CUDA threads, while the outer inter-tile dimensions will be mapped to CUDA blocks. As a consequence, the loop nest of stage `H` gets tiled such that the intra-tile loops `x_i` and `y_i` have sizes 8 and 6 iterations respectively or a threadblock of size 8x6. The equivalent CUDA pseudo-code can be found in Figure 6.2b. A separate CUDA kernel is launched for each stage and all pixels computed are stored in the global memory. Finally, Figure 6.2c shows a visual representation of the schedule, where the green pixels correspond to the tile applied to each loop nest, which is equal to the dimensions of the CUDA thread block. All pixels need to be loaded back from the global memory before they can be used in the consuming stages.

However, global memory accesses are often costly compared to ones in the cache or shared memory since DRAM bandwidth is often much lower than the one achieved by shared memory. A more efficient implementation would then require splitting the pipeline into groups of stages where each group corresponds to a different CUDA kernel and therefore global accesses only happen between groups, while all intra-group communication happens either through registers or the shared memory. However, such communication introduces extra synchronization between threads and therefore may limit the amount of parallelism that can be exploited.

As a consequence, optimizing a pipeline as a whole involves generating schedules that affect both the intra-group as well as inter-group granu-

larity [2, 82]. Inter-group scheduling focuses on the segmentation of the pipeline into groups of stages as well as inlining stages into their consumers such that maximum producer/consumer locality can be achieved. Scheduling stages within a group (intra-group) includes optimizations such as tiling, unrolling, selecting the variables that should be assigned as threads/blocks as well as determining the level of the consuming loop nest at which the computation of each producer should be placed. Figure 6.1b shows the new DAG after stages `D` and `H` have been inlined into their consumers (`J` and `W` respectively). Stage inlining is equivalent to replacing all occurrences of a producing stage inside the functional definition of the consuming stage with all necessary computations of said producer. The same pipeline after being partitioned into 4 groups (red dashed line) with stages `G, E, J` and `Z` as the output functions of each group is seen in Figure 6.1c. As seen in the new graph, the number of edges that correspond to global memory accesses has reduced in an effort to maximize producer/consumer locality.

An example of what is usually called an "overlapping tiles" schedule can be seen in Figure 6.3, where all non-inlined stages are computed as needed per intra-tile iteration (or per thread block) of the output stage. Inlining stage `H` is equivalent to replacing its occurrence in the definition of `W` with `2 * E(x, y) * 4`. Kernel fusion is achieved through the `compute_at, level` scheduling directive which tells the compiler to compute all pixels of a stage necessary for one iteration of `level` by the consumer. As a consequence, computation of stages `K` and `W` gets interleaved on a per-tile basis of the consumer `Z` and all pixels are stored in the shared memory. Contrary to the previous implementation, this one requires a single kernel to be launched for the whole group, and global memory accesses are limited to writes for the output, and reads for stages outside the group.

The equivalent CUDA pseudo-code can be found in Figure 6.3b. As already mentioned a single CUDA kernel is launched for the whole group and all pixels computed in a single intra-tile iteration are stored in the shared memory. Finally, Figure 6.3c shows a visual representation of the schedule, The blue and green pixels of the producing stages correspond to the pixels that will be computed before each intra-tile iteration and stored in the shared memory in order to produce the red pixels in the output. The green pixels indicate how the dependencies propagate in order to generate the pixels for one `x_i` iteration, while the orange arrows show the single pixel dependencies between stages. It is important to note that while the tile applied to the output would cause a 4x4 thread block on the generated CUDA kernel, assigning dimensions `x` and `y` of the producing stages `K` and `W` causes the actual thread block to grow into 4x8 due to inter-stage dependencies.

An even more optimized implementation is shown in Figure 6.4 where computation of stage `K` has been moved inside the inner thread dimension

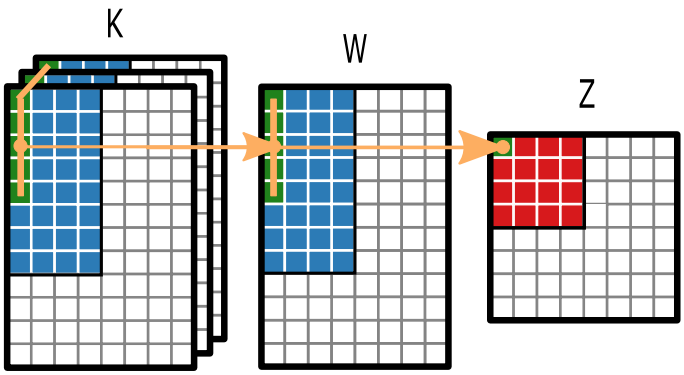
```
//function definitions for stages K,W,H,Z
K(x,y,c) = E(x,y) + E(x+1,y) + E(x+2,y)
H(x,y) = E(x,y) * 4
W(x,y) = K(x,y,0) + K(x,y,1) + K(x,y,2) + 2 * H(x,y)
Z(x,y) = W(x, y-2) + W(x,y-1) + W(x,y) +  W(x,y+1) +  W(x,y+2)

//group schedule
//start with the output of the group
Z.compute_root().gpu_tile(x, y, x_o, y_o, x_i, y_i, 4, 4);
W.compute_at(Z,x_o).gpu_threads(x, y);
K.compute_at(Z,x_o).gpu_threads(x, y);
```

(a) Definitions and Example GPU Schedule of Group `KWZ`: Computation of `K` and `W` has been moved at the block (innermost inter-tile level) of the output `Z`. A single kernel is launched for the whole group.

```
//produce Z
<CUDA>gpu_block Z.y_o
 <CUDA>gpu_block Z.x_o
  allocate __shared__ K[3*4*8]
  //produce K
  <CUDA>gpu_thread K.y_i
   <CUDA>gpu_thread K.x_i
    for K.c
     K(...) = ...
  //produce W
  //consume K
  allocate __shared__ W[4*8]
  <CUDA>gpu_thread W.y_i
   <CUDA>gpu_thread W.x_i
    W(..) = ...
  //consume W
  <CUDA>gpu_thread Z.y_i
   <CUDA>gpu_thread Z.x_i
    Z(..) = ...
```

(b) Equivalent pseudo-CUDA loop nest for segment `KWZ`: Allocation for stages `K` and `W` is moved to the shared memory. Grid dimensions are controlled by the tiling of the output `Z` loop and its dependencies with the producing stages.



(c) Visual representation of the dependencies and the previous schedule: The blue and green pixels of the producing stages correspond to the pixels that will be computed before each intra-tile iteration and stored in the shared memory in order to produce the red pixels in the output. The green pixels indicate how the dependencies propagate in order to generate the pixels for one `x_i` iteration, while the orange arrows show the single pixel dependencies between stages.

Figure 6.3: An overlapping tiles schedule computes all pixels needed for one intra-tile iteration (or thread block) and stores them in the shared memory.

of its consumer `W` to achieve what we call nested fusion in this work. Since one pixel of `W` requires three pixels of `K` (across the third dimension) but none across `x` or `y`, computing `K` per pixel of `W` does not cause redundant computation to increase. The equivalent loop nest is shown in Figure 6.4b, where we can see that computation of `K` is nested inside `W` and shared memory allocation is limited to the one required by `W`. This is better explained through the visual representation of the schedule in Figure 6.4c where none of the light gray pixels of Stage `K` need to be stored in the shared memory and are computed on-the-fly as needed by `W`. The third dimension of `K` is also unrolled to minimize loop overhead inside `W.x_i`. Nested fusion increases the work computed by the `W.x_i` threads sacrificing parallelism in the process but it can boost performance in applications with severe memory requirements by replacing large shared memory allocations with smaller ones in the constant memory and registers.

A larger tile size could further reduce the communication to the global memory (less pixels needed per tile by stage `E`), but may reduce the occupancy of the GPU and even cause the schedule to exceed the constraints imposed by the architecture. As an example, assume that the output stage `Z` is tiled with a 32x12 tile. Since dimensions `x` and `y` of stage `W` are also assigned as threads and due to the dependencies with their consumer `Z` (four extra pixels along `y`) the dimensions of the thread block will be

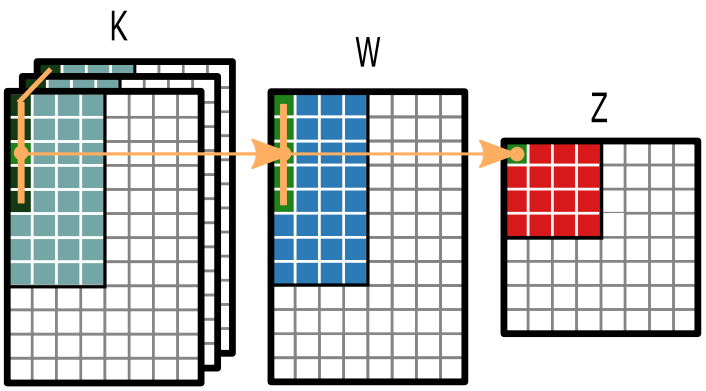
```
//function definitions for stages K,W,H,Z
K(x,y,c) = E(x,y) + E(x+1,y) + E(x+2,y)
H(x,y) = E(x,y) * 4
W(x,y) = K(x,y,0) + K(x,y,1) + K(x,y,2) + 2 * H(x,y)
Z(x,y) = W(x, y-2) + W(x,y-1) + W(x,y) +  W(x,y+1) +  W(x,y+2)

//group schedule
//start with the output of the group
Z.compute_root()
 //tile the loop
 .split(x, x_o, x_i, 4).split(y, y_o, y_i, 4)
 .reorder(x_i,y_i,x_o,y_o);
 //assign Vars to threads
 .gpu_threads(x_i,y_i).gpu_blocks(y_o,y_o);
//optimize the member stages
W.compute_at(Z,x_o)
 .reorder(x, y).gpu_threads(x, y);
//nested fusion should be allowed
K.compute_at(W, x).unroll(c);
```

(a) Definitions and Example GPU Schedule of Group `KWZ`: Computation of `K` has been moved at the block (innermost inter-tile level) of the output `Z` and `W` has been interleaved inside the thread level that computes `W`.

```
//produce Z
<CUDA>gpu_block y_o
 <CUDA>gpu_block x_o
  allocate __shared__ W[4*8]
  //produce W
  <CUDA>gpu_thread W.y_i
   <CUDA>gpu_thread W.x_i
    //produce K
    unrolled K.c
     K(..) = ...
    //consume K
    W(..) = ...
  //consume W
  <CUDA>gpu_thread y_i
   <CUDA>gpu_thread x_i
    Z(..) = ...
```

(b) Equivalent pseudo-CUDA loop nest for segment `KWZ`: Allocation of stage `W` is moved to the shared memory. A single kernel is launched for the whole segment. Values of `K` are computed as needed per pixel of `W` and stored in registers until consumption.



(c) Visual representation of the dependencies and the previous schedule: The blue and green pixels of the producing stages correspond to the pixels that will be computed before each intra-tile iteration and stored in the shared memory in order to produce the red pixels in the output. The green pixels indicate how the dependencies propagate in order to generate the pixels for one `xi` iteration, while the orange arrows show the single pixel dependencies between each stage. The light gray pixels correspond to values of `K` that will be produced (once) for one inter-tile operation without being stored into the shared memory.

Figure 6.4: Nested fusion can significantly lower shared memory usage, without increasing redundant computation

32x16 causing 512 threads per block in total and 2048 bytes allocated in the shared memory (assuming 4 bytes per pixel). Such dependencies can easily be derived by the compiler but are difficult to deduce by developers for more complex cases.

In GPUs, multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on an SM. Maximizing the occupancy can help hide latency during global memory loads which are followed by a thread synchronization command. The occupancy is determined by the amount of shared memory and registers used by each thread block. Achieved occupancy can be calculated using a set of equations that vary per Compute Capability (CC) of the GPU. These equations can be found in [59]. In this specific schedule, on a GPU of 7.5 CC and a configuration of 64Kbytes of shared memory per block, if we assume that our kernel requires 64 registers per block, 2048 bytes of shared memory usage and 512 threads per block we would get a 100% occupancy of each SM.

As seen from the above, proper kernel fusion alongside tile size selection has a direct impact on the amount of parallelism that will be exploited in the implementation, the occupancy of the GPU's SMs as well as the number of external (global) memory accesses. The problem we aim to

solve then lies *in introducing a model that can quickly generate an efficient schedule for a whole pipeline while ensuring that all constraints imposed by the target GPU architecture are satisfied.* Such a model needs to be able to find a balance between parallelism and redundant computations and should focus on minimizing the number of global memory accesses while maximizing the occupancy of the GPU's SMs.

6.4 GPU autoscheduler

This section presents the new optimization passes implemented in the Halide master [27] autoscheduler in order to generate optimization schedules that target CUDA-based GPU architectures. We follow a process similar to the current optimization flow where trivial (pointwise consumed) stages are first inlined into their consumers and then partitioned into groups using the greedy algorithm implemented in the Halide master. We adapt its model with new heuristics and steps which are presented in the following algorithms. Figure 6.5 shows an overview of the optimization flow used by the autoscheduler. Our method can generate schedules using the traditional overlapping tiles analysis, as well as a new nested fusion method. Throughout the next sections, our proposed GPU scheduler is named AutoGPU.

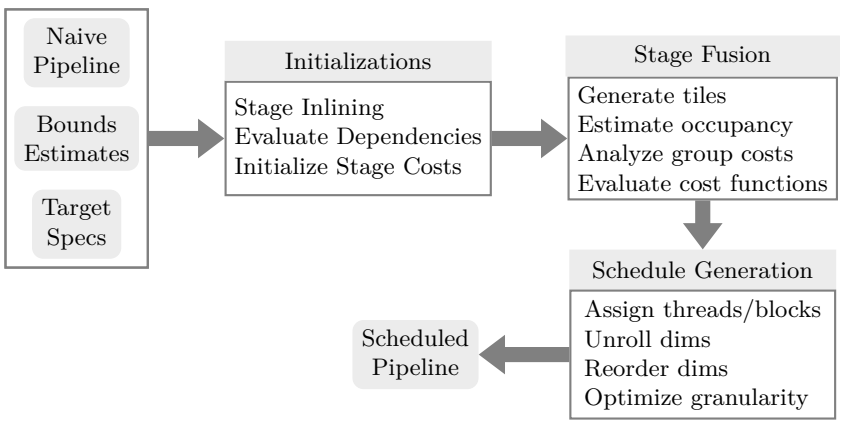


Figure 6.5: Basic Scheduling Flow: The scheduler requires the loop bounds estimates along with a target specification description given by the user in order to produce an optimization schedule for a given pipeline. Most of the steps in the compilation flow have been extended in order to support automatic GPU scheduling.

Initialization and overview

Most of the steps in the initialization process are identical with the ones performed by the CPU autoscheduler. The user needs to give an estimate of the problem size (loop bounds for the input buffers and outputs) as well as the specifications for the target architecture (compute capability). During the initialization step, the scheduler evaluates the amount of reuse/overlap between stages and inlines trivial functions. Trivial are considered the functions that are either consumed in a pointwise function or have a low arithmetic cost. After having initialized the cost of each stage, the scheduler uses the greedy algorithm of the Halide master in order to inline stages into their consumers when that is deemed beneficial by the model. The next step involves tiling and splitting the pipeline into segments, while the last step generates the final optimization schedule of the pipeline and further tightens the compute granularity of each stage when applicable. These two steps are discussed in more detail in the following subsections.

Stage fusion and tiling

This section discusses the new algorithms developed for automatic schedule generation targeting CUDA-based GPU architectures. These algorithms focus on efficient tiling and fusion of stages of the pipeline while exploiting both parallelism and producer/consumer locality. As already mentioned in Section 2, our scheduler is driven by an analytical model that expands upon a number of architecture specific parameters considered in prior related work by incorporating features such as the active Streaming Multiprocessors and threads per stage when evaluating the cost of a grouping configuration while also ensuring that the final schedule meets the constraints imposed by the target hardware platform.

The scheduler begins by determining which dimensions of each stage should be tiled. To this end, the bounds across each dimension are analyzed and in an attempt to limit the search space, loops with low iteration count are not tiled (e.g. channel dimensions in RGB images, small filter kernels). If all dimensions of a stage are found to have a low iteration count (i.e. less than 64 iterations), then we pick the largest one to be tiled, ensuring that at least one dimension can be tiled and an adequate number of blocks will be generated by the equivalent PTX kernel. The loop bounds for the outputs of the pipeline are derived by the estimates given by the user. Loop bounds for all producing stages are instead determined through the bounds inference analysis pass of the compiler. After the dimensions to be tiled have been determined, a list of all possible tile sizes for these dimensions gets generated. Since evaluating all possible combinations would require an enormous amount of

time for deeply nested loops, we impose an upper bound on the generated tiles. These upper bounds vary per dimension and depend on both its extent as well as the number of dimensions that will be tiled (N_{Tdims}). The bounds (upper T_{max} and lower T_{min}) for the generated tile sizes (T_{dim}) across each dimension (dim) based on the corresponding extents B_{dim} are selected such that the scheduler does not spend extra time evaluating options that are known to be inefficient or invalid. Invalid are considered the configurations that exceed the constraints imposed by the platform (number of threads or shared memory allocation higher than the maximum permitted), while inefficient are deemed those that do not exploit enough parallelism (e.g. number of blocks less than 2-4 times the number of SMs). These upper and lower bounds are defined based on the following equations:

$$T_{min} \leq T_{dim} \leq T_{max}$$

$$T_{min} = \begin{cases} 8, & \text{if } B_{dim} \geq 64 \\ 2, & \text{otherwise} \end{cases}$$

$$T_{max} = \begin{cases} \dfrac{B_{dim}}{128}, & \text{if } B_{dim} \geq 1024,\ N_{Tdims} = 1 \\ \dfrac{B_{dim}}{32}, & \text{if } B_{dim} \geq 1024,\ N_{Tdims} > 1 \\ \dfrac{B_{dim}}{2}, & \text{otherwise} \end{cases}$$

The numbers 2, 32 and 128, used as upper bounds for the tiles in the equations above, have been chosen such that at least one block is active per SM, but they can easily be changed in the model for architectures with a low SM count. In a similar fashion, the lower bounds ensure more than 8 threads per block in loops with extents larger than 64, and at least one block active per SM in loops with low iteration count. The step size used for the final tile size configurations is set to two.

After generating the tile size configurations that will be evaluated, we proceed to the fusion analysis of the pipeline's stages by recursively attempting to merge groups of stages until no more beneficial merges can be found. This process is performed using the greedy algorithm of the CPU autoscheduler in the Halide master. A merge is deemed beneficial only when the total cost of the new merged group is less than the sum of the costs of each individual group. Each independent group corresponds to a single CUDA kernel as seen in the examples of Section 3 (Figures 6.3 and 6.4). The cost of a group as well as the benefit of a configuration are determined through the algorithm and analytical model presented in listing 6.1. Specifically, A brief description of some of the terms used in the pseudocode is found in Table 6.1. The terms denoted with capital letters refer to architecture parameters used as constant constraints in the following algorithm.

Listing 6.1 shows the analysis that evaluates the costs for a given tiling configuration of a `group` (`memory` and `arithmetic`). Similar to the CPU

```
Function evaluate_group_costs(group):
 //evaluate load costs
 for each stage in group.inputs:
   memory_cost += stage.memory_cost / consecutive_loads
 for each stage in group.members:
   memory_cost += stage.memory_cost
   footprint += stage.footprint
 thread_count=1 //evaluate GPU terms
 for each dim in (group.output and loop_threads):
   thread_block[dim] = tiles[dim]
   thread_count *= thread_block[dim]
 //get extents required for each stage across each dimension
 local_bounds = dependence_analysis(group.tiles);
 occupancy = 1.0 //initialize metrics
 active_threads = MAX_THREADS_PER_BLOCK
 active_SMs = SM_COUNT
 for each stage in group.members:
   stage.thread_count=1
   for each dim in (stage.dims and loop_threads):
     stage.thread_block[dim] = local_bounds.stage[dim]
     stage.thread_count *= stage.thread_block[dim]
     //track maximum on each dimension for the thread block size
     thread_block[dim] = max(thread_block[dim],stage.threads[dim])
   {stage.occupancy, stage.active_threads, stage.active_SMs} =
    estimate_occupancy(stage.thread_count, footprint)
   //ensure occupancy above threshold
   if(stage.occupancy < OCCUPANCY_THRESHOLD)    return
    invalid_configuration
   //evaluate arithmetic costs
   arithmetic_cost += stage.arithmetic_cost /
                      (stage.occupancy * stage.active_threads)
   occupancy = min(occupancy, stage.occupancy)
   active_threads = min(active_threads, stage.active_threads)
   active_SMs = min(active_SMs, stage.active_SMs)
   thread_count = max(thread_count, stage.thread_count)
 //ensure resource requirements within target constraints
 if(thread_count % WARP_SIZE != 0 || active_SMs < SM_COUNT ||
    thread_count > MAX_THREADS_PER_BLOCK ||
    footprint > MAX_SHARED_MEM_PER_BLOCK)
   return invalid_configuration
 group_cost = //evaluate total cost
   arithmetic_cost + memory_cost / (occupancy * active_threads)
 return group_cost
```

Listing 6.1: Group Costs Analysis: Calculates the total cost of a group (kernel fusion and tiling) as well as various GPU specific metrics in order to ensure that the equivalent optimization schedule adheres to the target's resource constraints.

	Description
`group`	Stages merged together into a single kernel.
`tiles`	Tiling configuration applied on the group's output stage loop nest
`inputs`	Stages computed outside the group, or stages input to the pipeline.
`members`	Non-inlined stages of the group
`footprint`	Total size of the shared memory allocation required for this group
`thread_block[dim]`	Thread block size across each `dim` dimension.
`thread_count`	Sum of threads required for a computation.
`active_SMs`	Number of active SMs during a computation.
`active_threads`	Number of active threads during a computation (subset of the total threads).
`occupancy`	SM occupancy during the computation of a member stage.
`SM_COUNT`	Number of Streaming Multiprocessors in the GPU.
`MAX_THREADS_PER_BLOCK`	Maximum threads per block constraint.
`MAX_SHARED_MEM_PER_BLOCK`	Maximum shared memory per block constraint.
`WARP_SIZE`	Number of threads that make up a single warp.

Table 6.1: Notation of terms used in Listing 6.1

scheduler, the memory cost of a stage is calculated as the number of loads from a buffer, multiplied by a factor equal to the cost of accessing the global memory compared to a computation. Specifically, the algorithm first calculates the costs of loading data from stages outside the group (global memory accesses), which may be either stages from other groups or input buffers. Stages accessing input buffers (which typically reside in the global memory) have their memory cost divided by the number of consecutive loads from that buffer (in bytes) in order to take memory coalescing (lines 3-4) into account. The benefit from such loads is capped by the maximum transfers that can be issued per global transaction. The memory cost for stages within the group (`member` stages) is then calculated in a similar fashion. Allocations for buffers of such stages are allocated in the shared memory (line 6) and the sum of all such allocations will be equal to the shared memory requirements for a given tiling/grouping configuration (line 7).

For each specific tiling configuration for a `group` we need to estimate the dimensions of the thread block required for the corresponding schedule in order to ensure that the generated kernel does not exceed the target's constraints. If the `group` is a singleton (only one non inlined stage which

is the `output`), then the thread block dimensions are equal to the intra-tile extents (`tiles`) of the loop levels that were chosen to be assigned as threads by the equations described above, found in `loop_threads` (lines 10 to 12). This behavior can be seen in Figure 6.2 where each of the `K, H, W, Z` stages correspond to a different group and therefore an independent CUDA kernel whose thread block dimensions are equal to the tile sizes on each dimension. On the other hand, for groups with multiple (non-inlined) stages, the thread block dimensions have to be calculated based on the regions of each stage required to produce one tile of the output. These regions (`local_bounds`)s are inferred by the dependence analysis of the compiler (line 14). The actual final thread block size in each dimension will be equal to the maximum extent across all stages of the group (line 25) and the total number of threads will be equal to the product across each thread block dimension (lines 23). As an example consider the definitions and schedule of Figure 6.3a. The schedule of the group output Stage Z will require a thread block of 32x12 dimensions (due to tiling in x and y dimensions respectively), but due to its dependencies with the producing stage W, the actual grid dimensions will be 32x16 with a total of 512 threads per block.

Based on the total shared memory allocation, as well as the number of threads required per stage, we can calculate the occupancy of each stage, the number of active threads, as well as the number of SMs that will be active during the computation of said stage (line 26). This calculation takes place in a new pass (`estimate_occupancy`) that is implemented in our scheduler and is made based on the NVIDIA Occupancy Calculator [59] which can determine all of these metrics as a function of the number of threads (`estimated_threads`), shared memory per block (`footprint`), the number of registers per thread as well as the compute capability of the platform. Since it is not possible to accurately predict the number of registers that will be used at compile time, we estimate them such that:

$$N_{regs} \leq min(MAX\_REGS\_PER\_THREAD, \frac{TOTAL\_REGS\_PER\_SM}{N_{threads}})$$

If the occupancy of a stage is less than `OCCUPANCY_THRESHOLD` (usually set to 0.1) then the number of active warps per SM may severely limit parallelism and the configuration can already be considered inefficient (line 28). Unlike a CPU-only architecture where the number of tiles `n_tiles` is enough to obtain an estimate to the amount of parallelism that can be exploited, a GPU architecture requires all of the above metrics for a given configuration. In order to calculate the total arithmetic cost of each group, the arithmetic cost of each stage is scaled by the product of active threads, and occupancy (line 30). Finally, the total (sum) arithmetic and memory costs are multiplied by the number of tiles, and the number of

active threads and occupancy of the group is set to the minimum across all stages (lines 30-31).

The final cost of a grouping/tiling configuration can be evaluated given all of the metrics calculated above. We first check whether the new schedule will be valid for the target platform (lines 36-37). To this end, we ensure that the number of threads (`thread_count`) and shared memory per block (`footprint`) does not exceed the maximum values allowed for the architecture (`MAX_THREADS_PER_BLOCK` and `MAX_SHARED_MEM_PER_BLOCK` respectively). Unlike CPU scheduling, where such checks are not necessary, GPU schedules that exceed these platform specific constraints will cause the generated kernel to fail at run-time, and should therefore be invalidated by the scheduler's analysis as quickly as possible. We ensure that the minimum active SMs per group (`active_SMs`) are at least equal to the number of SMs in the platform (`SM_COUNT`, line 36). During the fusion analysis, the final, total cost is simply equal to the arithmetic and memory cost of the analysis, as determined by the algorithm in Listing 6.1. During the fusion/grouping analysis, we only generate tile sizes which are powers of two in order to further reduce optimization runtime (not shown in the listing for simplicity). However, the final tile sizes should ensure that the total number of threads is a multiple of the `WARP_SIZE` (usually 32 in most architectures). As a result, a final tiling pass, where tiling configurations use a step size of 2), is performed after all groupings have been concluded. During this step, the memory cost of a group configuration (tiling/fusion) is scaled by the product of occupancy and active threads in order to avoid situations where the tile sizes grow too large while the occupancy and active threads remain the same (line 39).

Similar to the CPU scheduler in the Halide master, after the group's cost has been calculated, a new grouping choice is picked for evaluation until no more beneficial group merges can be found. The pipeline whose groups result in the minimum overall cost is picked for the final optimizations and schedule generation.

Other optimizations and schedule generation

After tile sizes have been selected and the pipeline has been split into segments, we finalize the optimization schedule of the pipeline. For each group, we first tile the loop based on the sizes selected during the previous steps and then assign the outer (up to three) inter-tile variables as blocks and the outer intra-tile variables as threads (Halide `gpu_blocks` and `gpu_threads` respectively). The loop nests of each stage are reordered such that dimensions not assigned as threads are innermost (ordered based on their stride), followed by the thread dimensions and finally the block dimensions. Inner intra-tile loops (such as the kernels of convolution layers and the channel dimension of RGB images) are then unrolled.

```
Function max_order_reuse(consumer, producer):
 //find overlap dimensions with largest ordering in the consumer's
    loop nest
 overlap_dims = reuse_per_stage[consumer].find(producer);
 set = false
 for dim in overlap_dims:
   if (!set) max_order = dim
   else if(consumer.loop_order[dim] > consumer.loop_order[max_order
    ] :
     max_order = dim
 return max_order

Function optimize_granularity(group):
 for each stage in group.members:
   set = false
   if(stage == group.output_stage) continue
   //find its consumers
   for each consumer in group.members:
     //if the compute level is not set initialize it here
     if(!set):
       member.compute_level = max_order_reuse (consumer, stage)
       if(consumer == stage) stage.compute_stage = group.
    output_stage //consumes itself
       else stage.compute_stage = consumer
     else:
       if(topological_order(consumer) > topological_order(stage.
    compute_stage)):
       stage.compute_level = max_order_reuse (consumer, stage)
       if(consumer == stage) stage.compute_stage = group.
    output_stage //consumes itself
       else stage.compute_stage = consumer
 return
```

Listing 6.2: Nested Fusion Optimization Pass: A quick post-tiling pass that attempts to tighten the interleaving of stages by lowering the compute level of producing stages without affecting the amount of redundant computation.

Contrary to the traditional overlapping tiles analysis where computation of stages is always placed/interleaved at the innermost inter-tile (or GPU block) level of the output (consuming) loop nest, our scheduler can also generate schedules where nested fusion is enabled. Nested fusion allows scheduling the computation of stages at different levels, including at intermediate stages of the group similar to the schedule presented in Figure 6.4, where computation of stage `K` has moved from the block dimension (`x_o`) of the group's output stage (`Z`) to the inner thread dimension `x` of stage `W`. This optimization pass is applied on groups where member stages have severe resource requirements (i.e. high number of active threads, high shared memory usage).

The above code (Listing 6.2) demonstrates how nested fusion is implemented in our scheduler. The algorithm attempts to tighten the compute and storage granularity of a stage by lowering its `compute_at` level both in terms of consumers (`compute_stage`) as well as dimension (`compute_level`). After a loop ordering (`loop_order`) has been chosen, we schedule producing stages (each stage in `group.members`) at their last consumer (in topological order) and one level above the overlap dimension with the highest order in the consuming loop's ordering (lines 5-8), using the `max_order_reuse` function. For stages that only consume themselves (e.g. matrix multiplications, convolutions) the `compute_stage` is set to the group's output stage (lines 20 and 25). The amount and dimension of reuse/overlap per stage (`reuse_per_stage`) is determined during the initialization step of the autoscheduler as seen in Figure 6.5. On the example seen in Figure 6.4, computation of stage `K` has been moved to the `x` level of the loop nest of stage `W` since there is no reuse/overlap between `K` and `W` across iterations of `x` or `y`. On the other hand, stage `W` will not be moved below the innermost inter-tile loop level (block level `x_o`) since reuse possibilities exist across iterations of `y` (which is the outermost intra-tile loop of the consuming stage `Z`). This extra optimization step can further increase locality in applications with severe memory requirements by reducing shared memory allocations and allowing temporary values to stay in the constant memory or registers, at the cost of extra synchronization and therefore reduced parallelism.

6.5 Evaluation and experimental results

This section presents the results that were obtained using our proposed method on a test suite of 14 applications. We test our algorithms using two state-of-the-art CUDA-based architectures, the key parameters of which are shown in Table 6.2. The RTX 2080Ti platform is chosen to represent targets in the High Performance Computing domain, while the AGX Xavier represents the embedded domain. The list of benchmarks

along with the corresponding number of channels or dimensions of the output loop nest, number of stages and compile time (on an AMD Ryzen 2920X processor) using our scheduler can be found in Table 6.3. All benchmarks share a problem size of 1536x2560 (width, height) and differ in the number of output channels. Exceptions are the matmul and convlayer benchmarks that compute a 1536x1536 and 128x128x64x4 (width, height, output feature maps, batch size) output image respectively. A description of each of the benchmarks used can be found in [2, 52].

	RTX 2080 Ti	**AGX Xavier**
Compute Capability	7.5	7.2
L1 cache	64KB	128KB
Max Shared memory per Block	64KB	48KB
SM count	68	8
Max threads per Block	1024	
Max regs per Block	255	
Max regs per SM	65536	

Table 6.2: Architectural parameters for the two platforms

Benchmark	**[c,s,t]**
bilateral	[2,8,24s]
camera	[2,30,47s]
harris	[3,13,3s]
histogram	[3,7,4s]
IIR	[3,8,3s]
interpolate	[3,52,10s]
laplacian	[3,103,21s]
maxfilter	[3,9,15s]
unsharp	[3,9,2s]
nlmeans	[3,13,28s]
stencil	[3,34,28s]
lensblur	[3,74,51s]
matmul	[2,2,1s]
convlayer	[4,4,2s]

Table 6.3: Benchmarks, corresponding number of channels, functional stage and compile time using AutoGPU respectively.

Halide GPU scheduling

We compare our solutions to the manual schedules obtained from the Halide official repository [27] as well as the ones generated by the Li et al scheduler [43]. Some manual schedules were further optimized before benchmarking since the existing ones were either targeting GPUs with limited amount of available memory (interpolate) or older architectures (matmul) and the results would not be representative of actual expert-tuned schedules. To investigate the impact of each optimization pass/step

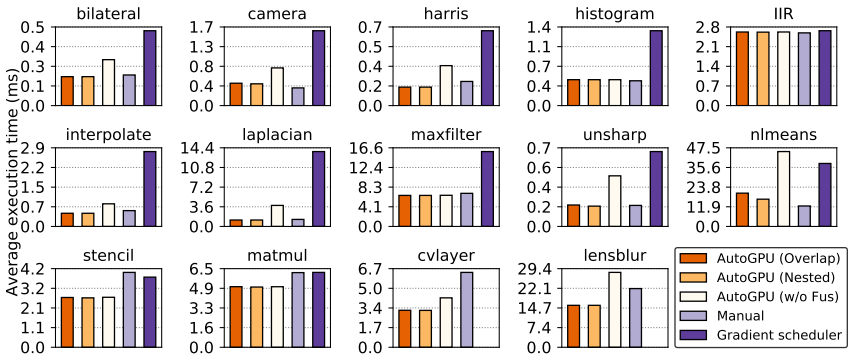


Figure 6.6: Average Execution time (ms) NVIDIA RTX 2080 Ti
Comparison of the average runtime of our proposed method (without stage fusion, with overlapped tiling and with nested fusion applied on all groups) with the manual tuned Halide schedules and the Li et al autoscheduler [43]

in our model, we generate three kinds of implementations: schedules where fusion is entirely disabled and stages are tiled and computed either inline or at root level (AutoGPU w/o Fus), schedules where fusion strategies are limited to the the traditional overlapping tiles technique (AutoGPU Overlap) and finally schedules where all optimization passes are enabled and nested fusion may also be applied on a group (AutoGPU Nested) depending on the heuristics described in the previous section. The performance of our proposed AutoGPU autoscheduler corresponds to the AutoGPU Nested bar.

The average execution time of each implementation is measured as follows: each application is executed 100 times and afterwards host and GPU device are synchronized. We measure the average time elapsed and repeat this process 100 times. The minimum average execution time across all samples is finally used in the following graphs.

Figure 6.6 shows the results obtained with the NVIDIA RTX 2080Ti platform. Our solutions outperform the Li et al scheduler [43] in all benchmarks with a significant speedup (over 5x) in large pipelines where fusion is beneficial. Moreover, our schedules result in an average of 10% performance improvement over the manual implementations. Three applications also have a moderate performance improvement when the scheduler operates under the nested fusion mode compared to only overlapping tiles. Forcing the scheduler to apply the nested fusion optimization on all groups would cause two benchmarks to suffer a slowdown (bilateral, lensblur) due to reduced parallelism. The effect of our tiling analysis can be determined by comparing the Li et al scheduler with the results that correspond to the no fusion schedules. Our solutions (AutoGPU w/o

Fusion) outperform the latter [43] in most cases due to a more extensive tiling analysis. We notice that four out of 14 benchmarks experience zero slowdown when fusion is disabled, since all implementations would converge to breadth-first schedules anyway (where all non-inlined stages are set to `compute_root`). The Li et al autoscheduler was not able to generate valid solutions for the last two applications (cvlayer and lensblur).

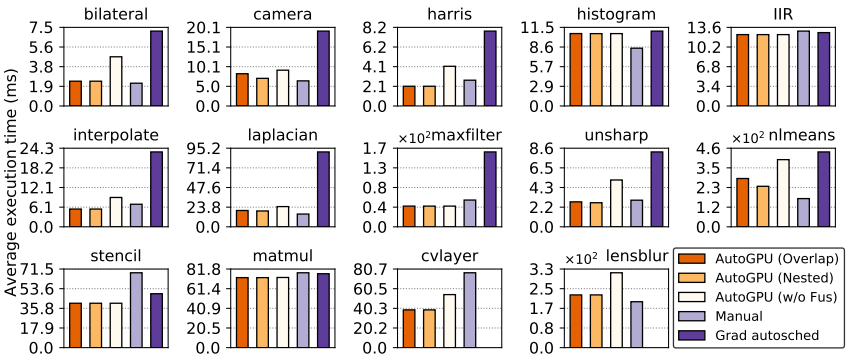


Figure 6.7: Average Execution time (ms) NVIDIA AGX Xavier
Comparison of the average runtime of our proposed method (without stage fusion, with overlapped tiling and with nested fusion applied on all groups) with the manual tuned Halide schedules and the Li et al autoscheduler [43]

Results for the same benchmarks when run on the NVIDIA AGX Xavier architecture while running at max clock on the default power mode are shown in Figure 6.7. The results follow a similar trend with our scheduler outperforming both the manual and the Li et al solutions, with the latter being slower in all cases even when fusion is disabled in our model. The only application where we can notice a deviation compared to the RTX platform is the histogram, where the Li et al autoscheduler performs similar to our methods due to limited parallelism offered by the platform (low SM count compared to the RTX 2080 Ti). Overall, we notice that two non-local means (nlmeans) and camera pipeline are the only applications with a significant benefit when nested fusion is enabled (around 40% and 33% respectively in the AGX platform). Pipelines with a small number of stages (histogram, IIR, matmul) do not offer large fusion opportunities and all three methods result in similar performance. Similar results (lower runtimes but similar ratios) were obtained on the maximum power mode.

All experiments were repeated on four more platforms with different GPUs of various generations. Figure 6.8 shows the average speedup achieved using our proposed AutoGPU method over the manual and Li

et al schedules for all six considered architectures. The performance of AutoGPU is equal to the AutoGPU-Nested bar of the above graphs and corresponds to the situation where fusion is enabled and the nested optimization pass is performed only when it is deemed profitable by the heuristics presented in the previous section. As seen in the graph, our schedules on average perform similar to the manually tuned ones. In detail, they achieve around 10% higher performance on the RTX 2080Ti and RTX 2070 platforms, 3% to 5% on the embedded Tegra boards (K1 and Xavier) but are 7% slower on the older GTX TITAN GPU. On the other hand, and as expected since the Li et al scheduler does not consider stage fusion, our solutions are 70% to 127% faster than the ones generated by [43].

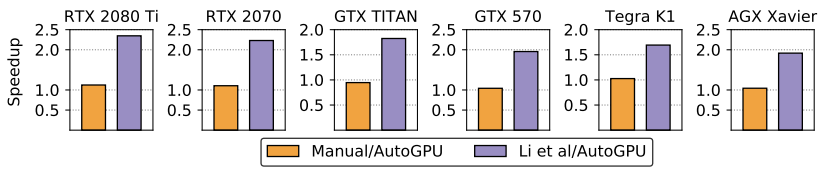
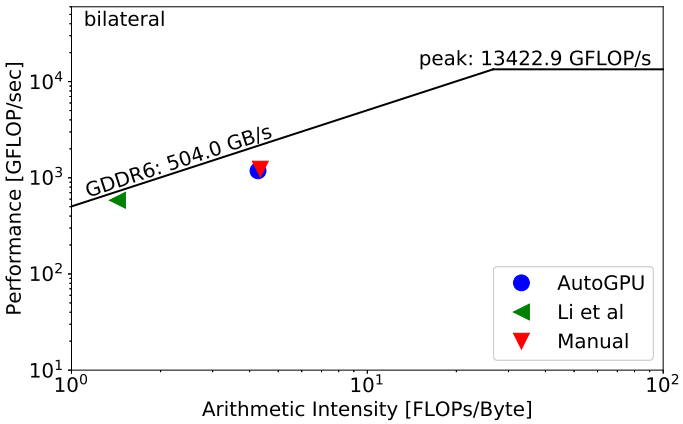


Figure 6.8: Speedup of AutoGPU compared to manual and Li et al scheduling: AutoGPU refers to our scheduler when all optimization passes are enabled.

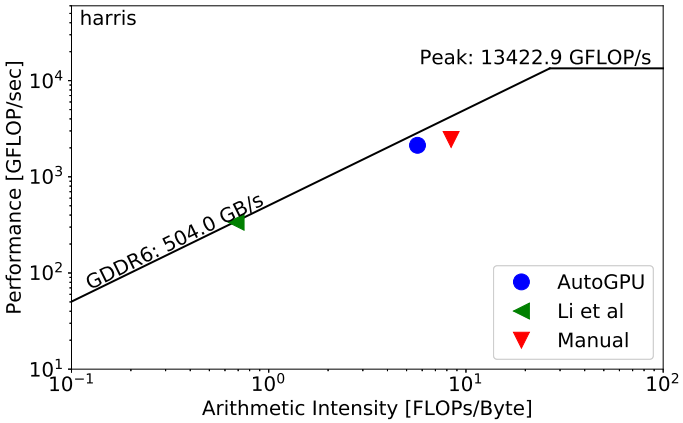
We should also note that even though our framework itself has not been optimized for compile-time, all schedules are generated within the order of seconds as can be seen in Table 6.3.

In order to further investigate our results, the roofline model for the RTX 2080 Ti platform [94] was derived for six of the benchmarks as shown in Figure 6.9. The roofline model can show how close an implementation is to the maximum performance achieved by the target platform. Memory bound applications are bound by the memory bandwidth of the hardware (GDDR6 on RTX2080 ti), while compute bound applications are bound by the maximum achieved performance, or Floating point Operations per second (FLOP/s). Arithmetic intensity was calculated after profiling each application using the NVIDIA Nsight profiler [60] in order to count the number of DRAM (and other memories for the hierarchical roofline) transactions, and Floating point Operations (FLOPs). Peak performance and bandwidth was measured using the Empirical Roofline Toolkit (ERT) [85].

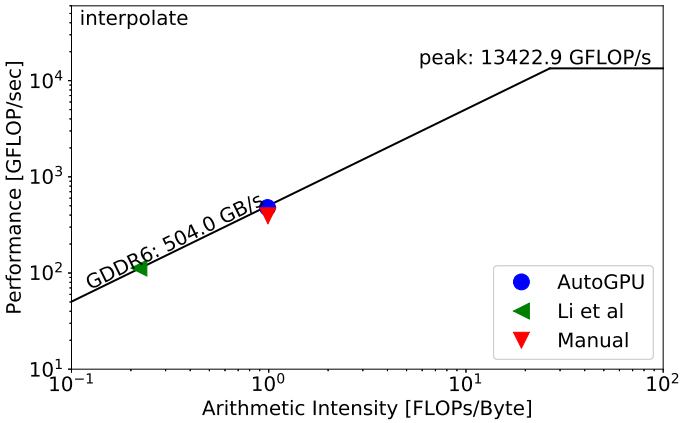
As seen from the above figures, all applications are mostly memory bound which is common in image processing. It is also interesting to note that since different optimization schedules can heavily influence the number of memory accesses as well as floating point operations (e.g. inlining) implementations do not share the same arithmetic intensity (AI). We can



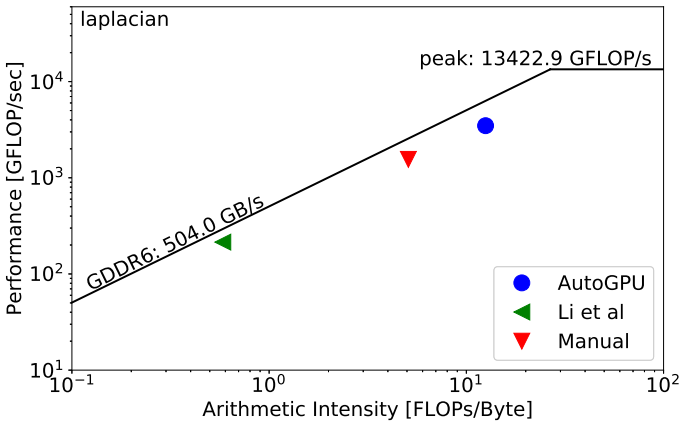
(a) bilateral



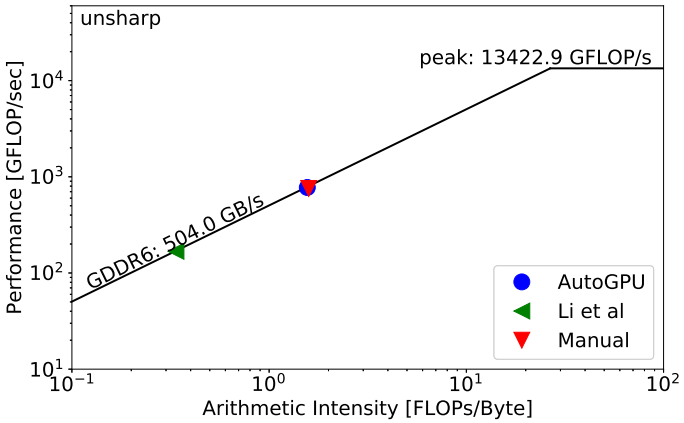
(b) harris



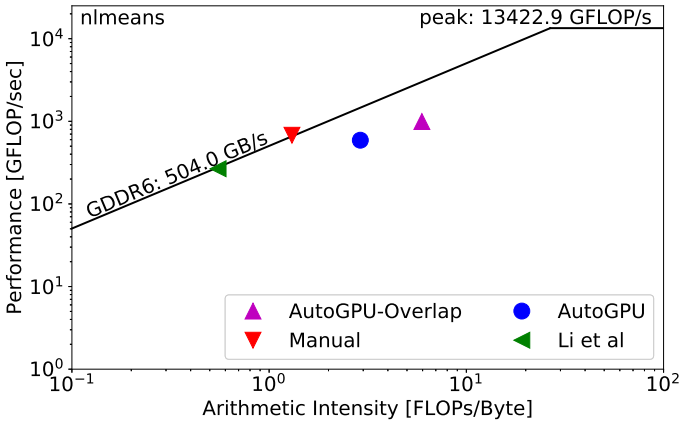
(c) interpolate



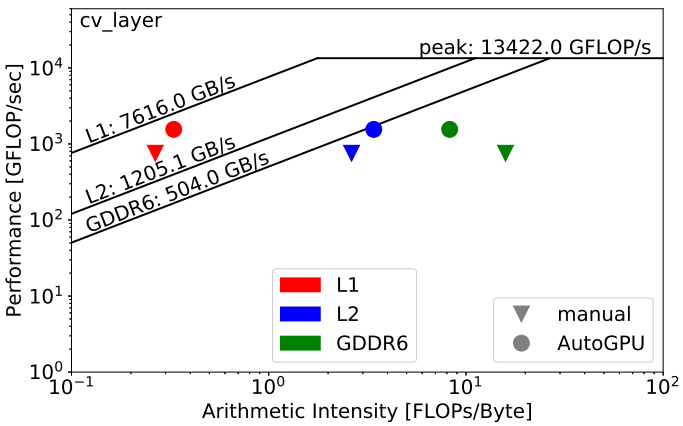
(d) laplacian



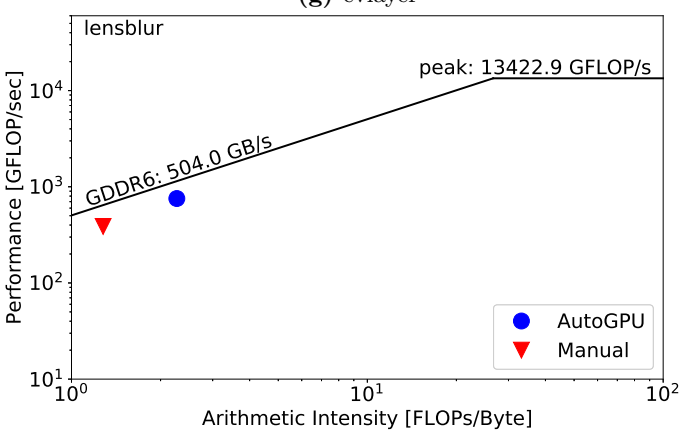
(e) unsharp



(f) nlmeans



(g) cvlayer



(h) lensblur

Figure 6.9: Roofline models for a subset of the applications used as benchmarks. The ceiling values correspond to the maximum achieved memory bandwidth of the RTX 2080 Ti architecture and the maximum achievable performance.

notice that for three benchmarks (bilateral, interpolate and unsharp) the AutoGPU implementations are equivalent to the manual ones and close to the ceiling imposed by the DRAM memory bandwidth. In two cases (laplacian and lensblur) AutoGPU schedules cause a higher AI allowing for higher performance. In cvlayer, AutoGPU achieves higher FLOP/s with more dram accesses (lower AI) but higher L1 and L2 AI which also explains the lower execution time. It is important to note however that all figures should be considered alongside the runtimes shown in Figure 6.6, since higher performance (in FLOP/s) does not necessarily mean lower execution time. As an example, consider the nlmeans benchmark where the AutoGPU overlap implementation achieves a higher performance than AutoGPU with nested fusion enabled even though the latter is 20% faster. Through nested fusion, AutoGPU requires less than half of the dram accesses of AutoGPU-Overlap (half bytes) for the same number of floating point operations. A similar situation happens for harris, where the manual schedule achieves a higher rate of floating point operations per second (FLOP/s) but at reduced performance compared to AutoGPU since it requires nearly 2x FLOPS for the same bytes (and has therefore higher AI). Finally, we can see that without loop fusion and a limited tiling model, Li et al is constrained to a much lower AI than the other implementations due to excessive memory accesses and no shared memory usage, which explains why it is heavily bound by a platform's memory bandwidth ceiling. This coincides with the fact that loop/kernel fusion and inlining can make applications less memory bound, enabling higher performance through other optimizations.

All experiments were repeated using a much smaller problem size (192x320 for most benchmarks and 512x512 for matmul) as well as a larger one (3840x2160 and 4096x4096 for matmul). For smaller problem sizes our scheduler performed on average similar to the manual (within 1%) while in the larger cases, our solutions outperformed the manual ones by 15% and the results were similar to the ones presented in Figure 6.6. In both cases the solutions generated by our scheduler were around 2 or more times faster than the ones given by the Li et al scheduler.

Finally, in order to showcase the portability of our approach to non-CUDA architectures, the whole test suite was repeated on an Intel GE onboard graphics card using the openCL target of Halide. The main changes that had to be made to account for the differences in the memory hierarchy and target specifications was to set the maximum threads per block to 512 (instead of 1024 in CUDA) and set the maximum tile size to half of that in CUDA architectures. The results obtained were similar to the ones presented above with the difference that the Li et al scheduler [43] was unable to generate valid schedules in a few benchmarks due to the reduced maximum threads/memory per block constraints.

Comparisons with other frameworks

As already mentioned in Section 2, HiPacc is a DSL similar to Halide, which was recently extended with a kernel fusion model for CUDA. We compared the performance of Halide using our proposed scheduler with the performance of HiPacc using the instructions provided in [72] for unsharp, harris and bilateral (which are the common benchmarks in the two suites). HiPacc was in all cases faster than Li et al but more than 2x slower than both the manual and our schedules. Unsharp was the only application where HiPacc was only 20% slower than our method and on par with the manual implementation. (However the two definitions of the algorithms were different, i.e. the Gaussian kernels in Halide get generated at runtime, while in HiPacc they are hardcoded.)

CuDNN is another widely used framework that provides hand optimized implementations of popular deep learning applications. We tested our autoscheduler on ResNet-50, a popular deep learning application used for image classification. Our solutions were on average 25% to 30% slower compared to the pytorch implementation with CuDNN enabled on the RTX 2080 TI platform. (Noted: No manual Halide, or other implementations were provided for this network). However, Halide is not yet capable of utilizing the tensor cores on the Turing architectures.

6.6 Summary

In this chapter we introduced a new analytical model along with novel optimization passes and heuristics for the Halide DSL and compiler in order to enable automatic generation of schedules targeting CUDA-based GPU architectures. We integrated our model into the Halide autoscheduler and tested it on a variety of image processing pipelines. Experimental results show that the generated schedules can achieve performance comparable to, or even better than that of manual, expert-tuned solutions.

Future work directions can either improve the current model with new techniques (i.e. multi-level tiling, unrolling of outer loops) or even use the heuristics we developed here as features in a learned autoscheduler similar to [2]. The occupancy of the target platform and the arithmetic cost per thread can for example be features that could be beneficial during the training process. Moreover, a scheduler more dedicated to deep learning could also be enabled as an extension to our framework with parametric based schedules for layers. Furthermore, an extended scheduler should integrate the existing CPU and GPU models in order to be able to independently decide whether pipelines/stages should be scheduled on the host CPU or offloaded into the GPU accelerator when present.

7

Programming Tensor Cores from an image processing DSL

Tensor Cores (TCUs) are specialized units first introduced by NVIDIA in the Volta microarchitecture in order to accelerate matrix multiplications for deep learning and linear algebra workloads. While these units have proved to be capable of providing significant speedups for specific applications, their programmability remains difficult for the average user. In this chapter, we extend the Halide DSL and compiler with the ability to utilize these units when generating code for a CUDA based NVIDIA GPGPU. To this end, we introduce a new scheduling directive along with custom lowering passes that automatically transform a Halide AST in order to be able to generate code for the TCUs. We evaluate the generated code and show that it can achieve over 5x speedup compared to Halide manual schedules without TCU support, while it remains within 20% of the NVIDIA cuBLAS implementations for mixed precision GEMM and within 10% of manual CUDA implementations with WMMA intrinsics.

7.1 Introduction

Matrix multiplication (GEMM) has proven to be an integral part of many applications in the image processing domain [24]. With the rise of CNNs and other Deep Learning applications, NVIDIA designed the Tensor Core Unit (TCU). TCUs are specialized units capable of performing 64 (4x4x4)

multiply - accumulate operations per cycle. When first introduced alongside the Volta microarchitecture, these TCUs aimed to improve the performance of mixed precision multiply-accumulates (MACs) where input arrays contain half precision data and accumulation is done on a single precision output array. With the newer Turing architecture, TCUs also support fixed precision MACs as well as more data types compared to the previous generation.

Although TCUs can significantly increase the performance of applications such as DNNs and other tensor contractions whose main workloads can be formulated as matrix multiplications, direct programmability of these units remains either inaccessible to non CUDA experts, or completely hidden behind libraries such as cuBLAS and CUTLASS.

Halide [73] is a Domain Specific Language (DSL) for image processing applications that aims to increase code portability and readability by separating the functional description of an application from its optimization schedule. Using LLVM [41] as a backend compiler, Halide can target various architectures including multi-core CPUs as well as CUDA based GPGPUs. These multi-core CPUs can often act as a host while parts of the code are offloaded into a GPU which acts as an accelerator.

In this chapter we extend the Halide DSL with the `tensor_core` scheduling directive, along with all necessary lowering and backend compiler passes in order to allow the compiler to automatically utilize the TCUs when asked by the user. To this end, we implement custom lowering passes that replace the parts of the AST that correspond to the traditional matrix multiplication and inject calls to new compiler intrinsics that correspond to tensor operations. Furthermore, using NVVM as a backend, we extend the PTX (Parallel Thread Execution) code generator for each of the new intrinsics. Finally, we demonstrate that through our extensions, the compiler can automatically generate a highly optimized implementation of GEMM without the user having to worry about data types, loop bounds or other scheduling choices. Experimental results show that the performance of the generated code is over 5x faster than manually tuned Halide schedules without tensor core support, and close to or even faster than NVIDIA cuBLAS implementations.

The rest of this chapter is organized as follows: Section 7.2 presents background information on the NVIDIA tensor core architecture, focusing on its programmability as well as matrix multiplication in the Halide DSL. Section 7.3 discusses related work on compiler support for similar architectures across various DSLs. Section 7.4 introduces the new scheduling directive along with all necessary compiler passes that enable code generation for TCUs in Halide, along with an example optimization schedule that was used for benchmarking. Section 7.5 evaluates the performance of generated code based on the aforementioned schedule compared to equivalent cuBLAS implementations as well as manually scheduled Halide

implementations without tensor core support. Finally, concluding remarks are made in Section 7.6. This work was published in [80].

7.2 Background information

This section presents key background information on the NVIDIA Tensor Core architecture and its programmability, as well as on the Halide DSL and compilation flow.

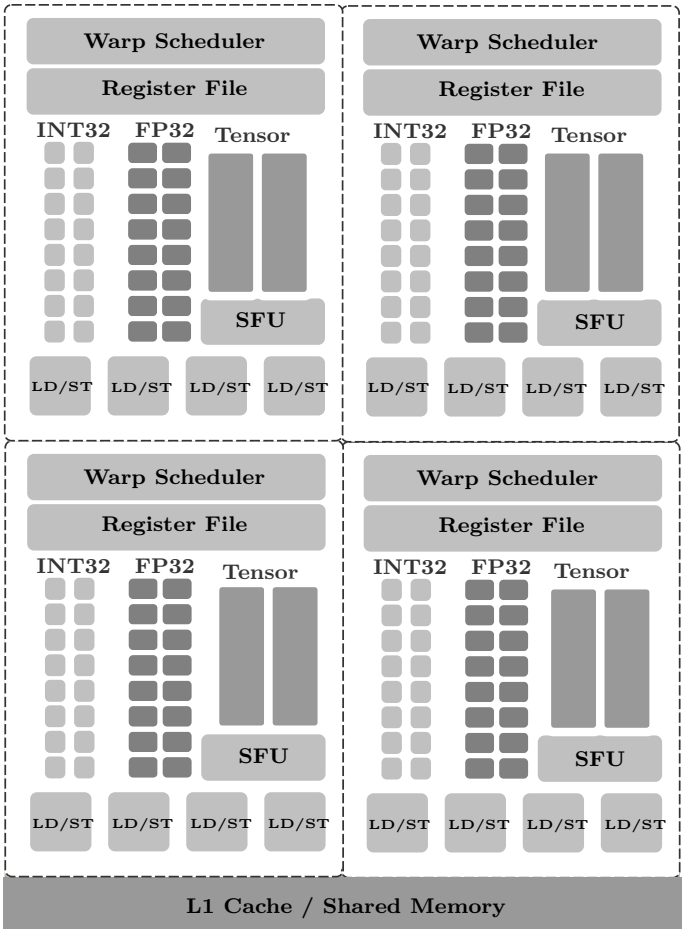


Figure 7.1: Simplified view of the Turing SM microarchitecture. Each SM sub-core contains 2 Tensor Cores capable of executing 64 multiply/accumulate operations per cycle. Warps in each subcore can utilize these units and can communicate through the shared memory.

The NVIDIA Tensor Core architecture

The NVIDIA Tensor Core Unit [58] (TCU) was first introduced alongside the Volta architecture and is also present in the Turing microarchitecture. A simplified model of the Turing Streaming Multiprocessor (SM) microarchitecture can be seen in Figure 7.1. Each SM contains four sub-cores (processing units). Sub-cores contain two processing units, each capable of executing $4x4x4$ multiply/accumulate per cycle. This translates to 128 operations per cycle for every TCU. On a Turing RTX 2080Ti (TU102, which was used in our experiments) that operates on a 1.635Ghz clock and contains 68 SMs (or 544 TCUs), theoretical tensor core performance reaches 113TOPS. Similar architectures have been introduced by Google [33] (TPU) and Intel [76] (NNP).

NVIDIA provides two dinstinct ways of programming these units: a) Widely used libraries such as cuBLAS [55] and cuDNN [16] have been extended with new kernels that utilize the TCUs to accelerate GEMM performance. CUTLASS [56] (CUDA templates for Linear Algebra Subroutines), another NVIDIA library that built upon C++ in order to enable high-performance in BLAS-like kernels supports code generation for the TCUs as well. b) the CUDA WMMA (Warp level Matrix Multiply and Accumulate) API provides a more direct way for CUDA developers to program these units using specific intrinsics. These intrinsics operate on a new data type called `fragment` which represents the part of the array that will be used in the following TCU instructions and can vary per data type, memory layout of the corresponding array and/or size. Fragments can either be used in load, store or multiply/accumulate instructions (`wmma.load`, `wmma.store` and `wmma.mma` intrinsics respectively). Load and store intrinsics can read/store into the shared or the global memory. In this work, we instead use the NVVM IR intrinsics that correspond to the above instructions, and extend the Halide compiler passes accordingly in order to generate high-performance GEMM kernels.

Matrix multiplication in Halide

As already mentioned in Chapter 3, Halide separates the algorithmic description of an application from its optimization schedule. As an example, consider the code seen in Listing 7.1, which implements a simple matrix multiplication kernel in Halide.

In detail, lines 5 and 6 are responsible for the functional behavior of the application and define the relationship between output and input data. Line 5 initializes all elements of array C to zero and then line 6 which is called an update definition (in Halide terms) over the initialization describes the matrix multiplication of arrays A and B (and accumulation in output array C). Lines 9 and 10 dictate the optimization schedule of the

```
Var x("x"), y("y"),xi("xi"),yi("yi");
int matrix_size=1024;
// Algorithm
RDom k(0, matrix_size);
C(x, y)  = 0.0f ;
C(x, y) += A(k,y) * B(x,k);

// Schedule
C.compute_root().gpu_tile(x,y,xi,yi,32,16);
C.update().gpu_tile(x,y,xi,yi,32,16);
```

Listing 7.1: Example Matrix Multiplication in Halide

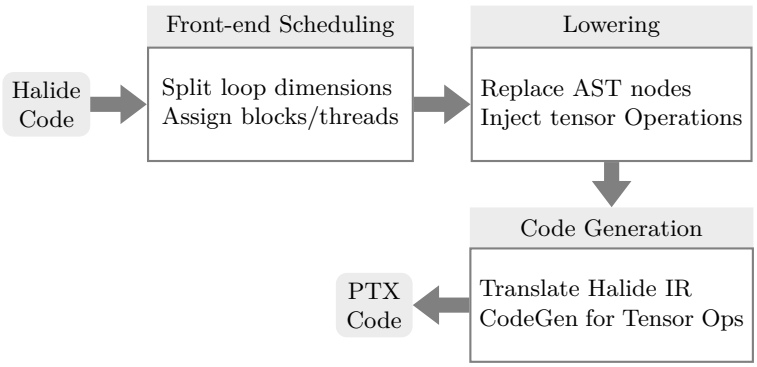


Figure 7.2: Basic Compilation Flow: The `tensor_core` scheduling directive dictates that the corresponding Halide function should be ported on the TCUs. Calls to custom tensor core intrinsics are injected into the AST during lowering and finally translated into LLVM/NVVM IR before generating the final PTX code.

implementation and control details such as the loop transformations that will be applied on the generated code, the memory storage and layout for intermediate buffers, the order in which input data is loaded and other architecture specific information, such as the loop dimensions that correspond to the x, y and z dimensions of a threadblock on a GPU. It is important to note that the optimization schedule does not affect the functional output of the code. In this specific example, the `gpu_tile` scheduling directive will cause the compiler to tile the `x` and `y` dimensions with tile sizes of 32 and 16 respectively, such that `xi` and `yi` are the inner intra-tile loops and `x`, `y` the outer inter-tile loops. At the same time, the inner (intra-tile) loops will be assigned as CUDA threads while the outer (inter-tile) loops as block dimensions. The same will be done for

the update definition of `C`.

As seen in the above example, Halide allows for increased code portability and readability, since: a) the optimizations applied through the scheduling directives cannot change the functional output of the code and b) changing the target platform only requires rewriting the schedule. All scheduling transformations are applied after the initial description has been lowered into an intermediate representation (IR). Halide IR remains platform independent until later stages of the compilation flow where it is translated into LLVM IR for the final target code generation. Specifically, for CUDA based architectures, Halide IR gets mapped into LLVM's NVVM backend to generate PTX code. NVVM IR is a compiler IR (internal representation) based on the LLVM IR which was designed to represent GPU compute kernels (for example, CUDA kernels). For each Halide function/stage a different CUDA kernel is generated through LLVM. In the above example, one kernel would be launched to initialize the output array to zero and another one would be responsible for the actual matrix multiplication.

We will show that by injecting new tensor specific intrinsics as well as properly modifying Halide's PTX code generator (where the translation to LLVM IR occurs for the PTX backend) we can generate high performance code that utilizes the NVIDIA TCUs.

7.3 Related work

Imaging DSLs provide an efficient high-level way for developers to generate high-performance implementations without sacrificing code readability and maintainability. Due to this reason, custom backends and extensions for architectures that operate as accelerators have been a popular subject in the context of such languages [51, 73, 74, 87]. Most DSLs use LLVM as a backend to generate code for accelerators. Halide [73] and Tensor Comprehensions [87] are two example languages that translate their IR into LLVM IR in order to generate PTX code for GPU offloading. In a similar fashion, Halide can generate code for the Qualcomm Hexagon DSP with HVX extensions. Other approaches make use of C code generators in these DSLs to support FPGA code generation through HLS (High level synthesis) [68], or DSPs [89]. TVM [15] is a deep learning compiler stack, heavily influenced by Halide IR. It supports various architectures and was recently extended with Tensor Core code generation [86]. However, unlike our approach, TVM instead injects CUDA WMMA intrinsics into C/C++ output code and requires nvcc to compile the final implementation. Our method does not need the code to be linked with any CUDA libraries during design time and thus enables cross compilation. Moreover, TVM requires the user to make scheduling choices for the mapping, while we

predefine an efficient schedule that as seen in the benchmarks of Section 7.5 can achieve high throughput in nearly all cases while remaining generic.

7.4 Tensor Cores in Halide

This section presents our extensions to the Halide compiler in order to enable code generation for the NVIDIA TCUs. To this end, we introduce a new `tensor_core` scheduling directive. After our extensions the optimization schedule presented in Listing 7.1 becomes:

```
1 // Schedule
2 C.compute_root().gpu_tile(x,y,xi,yi,32,16);
3 C.update().tensor_core(A,row,B,col);
```

Listing 7.2: The new tensor_core directive is enough to generate high performance code for GEMM kernels in supported architectures

NVVM Intrinsic	Layout			Data Type		
nvvm.wmma.m16n16k16	A	B	C	A/B	C Type	Halide IR
load.a.row.f16	row	-	-	float16	-	mma_load
load.b.col.u8	-	col	-	uint8	-	mma_load
load.c.row.f32	-	-	row	-	float32	mma_load
store.d.col.i32	-	-	col	-	int32	mma_store
mma.col.col.f32.f32	col	col	-	float16	float32	mma_operation
mma.row.col.u8	row	col	-	uint8	int32	mma_operation

Table 7.1: NVVM WMMA Intrinsic Selection Examples

As seen in the above listing, the new directive only requires the input arrays and their storage layout. The size of the fragments, as well as the data types and the necessary NVVM intrinsics to generate code are all automatically derived based on the type of the input buffers. In a similar way, the dimensions of the grid to be launched as well as the amount of shared memory to be requested for the implementation are calculated. A simplified view of the compilation steps required to generate TCU code can be seen in Figure 7.2. In the language front-end, we introduce the `tensor_core` directive which marks the computation of a Halide function to be ported on the TCU. At the same time we split the dimensions of the loop such that we create a 256x256 tile (similar to the CUDA WMMA implementation in [57]) and assign the corresponding dimensions to the CUDA grid dimensions. Finally we propagate the types of the input and

output arrays, as well as their layout to the upcoming IR passes. Since this is a new scheduling directive, we need to provide new information during the lowering phase in order for the compiler to know what kind of new transformations should be applied on the code. We first attempt to fix the bounds of the loops that need to be modified to account for the required grid dimensions. To this end, we introduce a new lowering pass during which we transform the AST by injecting IR nodes with calls to tensor operation intrinsics. For the rest of this chapter, A, B and C refer to the arrays/buffers of Listing 7.1, while `a`, `b`, `c` and `d` refer to the tensor fragments used in WMMA operations. We introduce an intrinsic for each type of tensor instruction:

1. `mma_load(a/b/c,pointer,layout,stride,type)` : Loads data from the memory location of `pointer` into the `a/b/c` fragment of a TCU. The memory location may be in either the shared or global memory. The size, type and stride of the fragment is inferred by `type` and `stride`.

2. `mma_store(pointer,layout,stride,type)` : Stores the `d` fragment data from the TCU registers into the memory location specified by `pointer`, which can point to either the shared or global memory. The size, type and stride of the fragment is inferred by `type` and `stride`.

3. `mma_operation` : Multiplies the `a` and `b` fragments and accumulates the result (plus `c`) in the `d` fragment.

All of the above arguments that are not present in the `tensor_core` scheduling directive are automatically inferred by the compiler.

We use a modified version of the schedules provided by NVIDIA [57] as an example schedule to showcase the effectiveness of the tensor core architecture in Halide. We focus on the optimized version which automatically makes use of the maximum shared memory available on the platform to store chunks of A, B and C arrays. When this is not possible, a naive implementation that does not make use of the shared memory is provided at the cost of performance. Efficient streaming of A/B/C data to and from the global memory is realized through calls to memcpy. Listing 7.3 shows an example lowered AST of the schedule generated by the lowering pass for arrays of size 1024×1024, row, column layout for A and B respectively and mixed precision (float16 for A, B and float32 for array C). For simplicity, the pointer and indexing calculation is not shown, but all pointers are typically functions of the surrounding loop iterators. The number of blocks of the launched kernel is equal to the SM count of the GPU. A While node was also implemented in the Halide IR as seen in line 4 of Listing 7.3 to account for the step size in the original CUDA WMMA code.

```
gpu_block<CUDA> (output.s1.x.__block_id_x, 0, 68) {
 allocate __shared[65536] in GPUShared
 gpu_thread<CUDA> (.__thread_id_x, 0, 256) {
  while (s0, output.s1.x.__block_id_x, ((s0*8)/64)*(8) < 64, 68){
   unrolled (s1, 0, 16) {//Stream C into shared memory
    (float32)mma_memcpy_C_to_shared(ptr_C, ptr_shared)
   }
   gpu_thread_barrier()//Sync threads
   unrolled (s2, 0, 8) {//Load multiple fragments of C
    (float32)mma_load(c, ptr_shared, row, stride)
   }
   gpu_thread_barrier()
   unrolled (s3, 0, 16) {//loop over global K dimension
    unrolled (s4, 0, 8) {//Stream A/B chunks into shared memory
     (float16)mma_memcpy_AB(ptr_A, ptr_B, ptr_shared)
    }
    gpu_thread_barrier()
    unrolled (s5, 0, 4) {
     unrolled (s6, 0, 2) {//Load fragment A
      (float16)mma_load(a, ptr_shared, row, stride)
      unrolled (s7, 0, 4) {//Load fragment B
       (float16)mma_load(b, ptr_shared, col, stride)
       (float32)mma_operation()//Multiply/accumulate
      }
     }
    }
    gpu_thread_barrier()//Sync threads
   }
   unrolled (s8, 0, 8) {//Store accumulator into shared memory
    (float32)mma_store(ptr_shared, row, stride)
   }
   gpu_thread_barrier()//Sync threads
   unrolled (s9, 0, 16) { //stream C back to global memory
    (float32)mma_memcpy_C_to_global(ptr_C, ptr_shared)
   }
   gpu_thread_barrier()//Sync threads
  }
 }
}
```

Listing 7.3: Example lowered AST of the computational loop after tensor operations have been injected. The schedule has been adapted from NVIDIA. All loop bounds and types are automatically derived by the compiler.

Elements of C are first streamed into the shared memory (from the global memory). In case C is zero as in the example of Listing 7.1 then the call to the initial `memcpy` can be replaced by a `memset`. The same data is then loaded into fragments to be reused in the following multiply/accumulates through a `mma.load.c` instruction. A similar process is repeated for chunks of A and B arrays. Iterating across the global K dimension, fragments a and b are multiplied and accumulated into fragment d, reusing fragments of b against a in the process. Finally, d fragments are stored into the shared memory before being streamed back into the global memory. All of the previous steps are repeated until there are no more tiles to compute for each block. More specific details for the schedule can be found in the NVIDIA CUDA Samples [57].

At this point the Halide IR along with the newly injected tensor nodes/calls needs to be translated into LLVM/NVVM IR before the final code generation. Proper NVVM intrinsic selection depends on the data type and storage layout of the input/output arrays. All the necessary information was either already propagated by the previous compilation stages or was inferred by the compiler. An example of Halide IR to NVVM intrinsic mapping can be seen in Table 7.1. All fragments of C/D are 8 elements wide, while fragments of A/B arrays are 4 elements wide when they contain `uint8` elements and 8 elements wide when the data type is `float16`.

Since load and multiply/accumulate operations return a fragment data type (an array of 8 floats or integers depending on the data type of the output), all elements in the accumulator fragment C/D need to be extracted and stored into the local memory after each `wmma.mma.load.c` operation in order to ensure that the accumulator fragment is actually updated after each cycle. For `wmma.mma.store.d` operations, the same elements need to be loaded into registers before they can be used in the instruction. Finally, `wmma.mma` operations are handled by first loading the previous values of the accumulator into registers and then storing them back after each multiply/accumulate. This process is automatically handled during the translation of the Halide IR tensor intrinsics along with all necessary memory allocations.

7.5 Evaluation

This section presents the experimental results obtained using the `tensor_core` scheduling directive along with the other extensions presented above.

All experiments were performed on a RTX 2080Ti NVIDIA GPU with 68 SMs. By default CUDA limits the maximum allowed shared memory per block to 48Kb. Bypassing this limit is possible by set-

ting the maximum dynamic shared memory size to 64Kb through the `cuFuncSetAttribute` PTX runtime function which is added to the Halide CUDA runtime before calling the kernel. The average execution time of each benchmark was measured through Halide's `benchmark` subroutine across 100 iterations and 10 samples followed by a device sync.

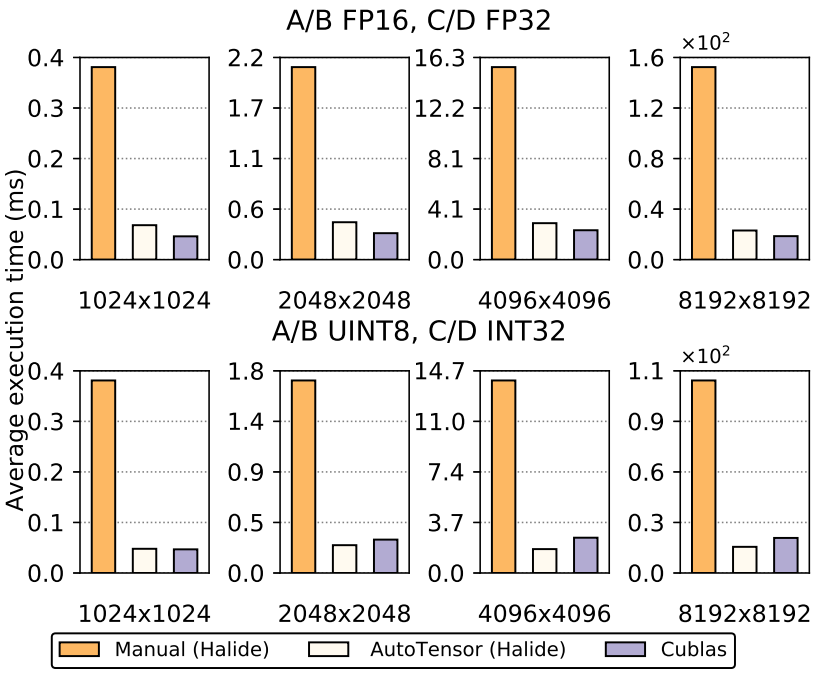


Figure 7.3: Matrix multiplication average execution time on an NVIDIA RTX 2080Ti GPU, for mixed precision and fixed-point implementations on various problem sizes.

The experimental results obtained on the NVIDIA RTX 2080Ti GPU are shown in Figure 7.3. All implementations refer to the Halide function of Listing 7.1. The top rows show the results across four different problem sizes when the input A/B arrays contain elements of half precision and the output is calculated in full precision, while the bottom row shows the same results on fixed-point implementations. The "manual" bar refers to the fastest Halide optimization schedule for matrix multiplication (that does not use tensor cores), while cuBLAS refers to the results obtained using cuBLAS on CUDA version 10.0. The optimized implementation we provide assumes that the storage layout for array B is column major while the Halide code of Listing 7.1 requires all arrays to be row major.

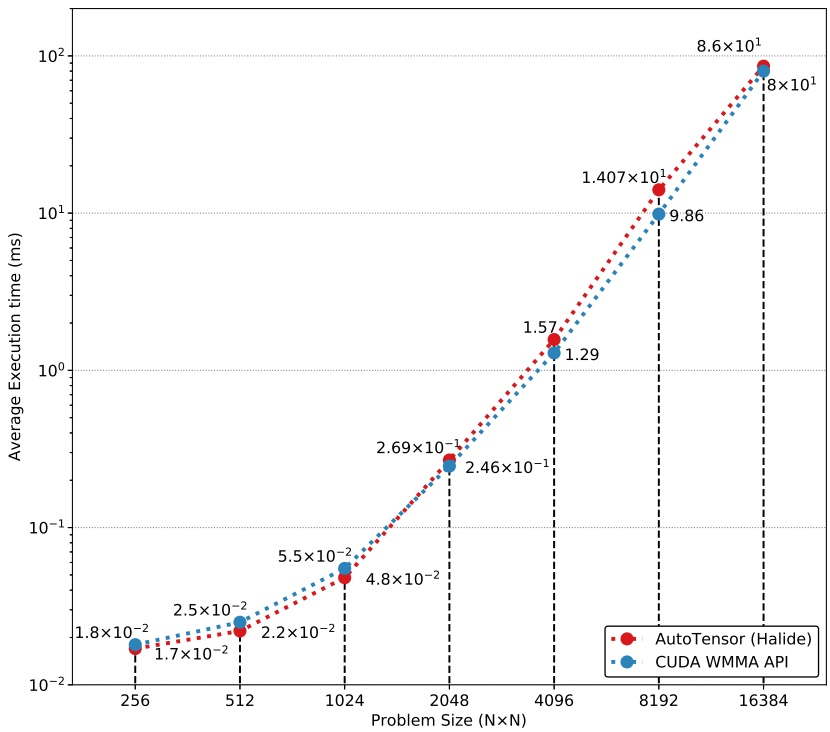


Figure 7.4: Average execution time of matrix multiplication on a wide range of problem sizes on NVIDIA RTX 2080Ti.

Due to this reason, we transpose array B before using it in the matrix multiplication. The execution time for this transposition is taken into account for the "AutoTensor" bar of Figure 7.3. Tensor cores speed up performance by up to 10x on larger problem sizes. The overhead of transposition accounts for less than 10% of the runtime. Moreover, our tensor core Halide implementation remains on average within 29% of the cuBLAS performance for mixed precision matrix multiplications, and it is even faster when using integer data types since cuBLAS cannot yet use tensor operations for Turing architectures: Most BLAS kernels in cuBLAS are optimized for the Volta architecture which did not have integer tensor core support. When both Halide and cuBLAS assume a row, column major layout for arrays A and B and no transposition is needed, Halide can stay within 18% of cuBLAS performance.

Since standard cuBLAS routines (SGEMM) cannot yet utilize the tensor core units when using int32 data types, we instead compare the performance of our implementations against the hand-optimized fast GEMM CUDA kernel found in [57] with imma (Integer Matrix Multiply Accumulate) intrinsics, specialized for the Turing architecture. Figure 7.4 shows an extended comparison for Int32 matrix multiplications on the same platform across a wider range of problem sizes. Both implementations assume a row, col storage layout for arrays A and B respectively. We see that the automatic Halide implementation achieves competitive performance (7% slower on average) with the manual CUDA code [57] in nearly all cases.

7.6 Summary

In this chapter we presented a new scheduling directive for the Halide DSL along with a set of intrinsics and lowering passes that enable automatic, efficient code generation for matrix multiplications leveraging the Tensor Core units present in the latest CUDA GPU architectures. We evaluated our extensions using a Turing based NVIDIA GPGPU and showed that through our custom intrinsics and passes, Halide can achieve performance competitive with cuBLAS and manual CUDA code on matrix multiplication kernels.

8

Conclusions and Future Work

This dissertation introduced a set of optimization frameworks for image processing and computer vision applications. The developed frameworks are capable of generating efficient code that targets both multi-core CPU as well as GPU architectures. This thesis shows that when targeting domain specific applications, analytical modeling and heuristics are enough to achieve consistent performance competitive to manual, expert-tuned solutions. Through properly defined analytical models that capture key architecture-specific parameters, the frameworks introduced in the previous chapters enable quick generation of optimization schedules without the need of time-consuming auto-tuning, thus also enabling cross-compilation for embedded edge devices.

It should be noted that all of the optimization algorithms introduced in Chapters 4, 5, and 6 can be used alongside any programming language or compiler. However, the Halide DSL and compiler enabled quick design-space exploration and evaluation of the proposed concepts thanks to the explicit separation of algorithmic description and *optimization schedule*. As a result, all tools developed to accompany the above chapters have been designed to be used alongside the Halide language. Furthermore, domain-specific compilers like Halide, TVM, PolyMage and Tensor Comprehensions allow developers to use a higher-level syntax to perform intricate low-level optimizations. When paired with automatic scheduling, they can be extremely useful especially when targeting specialized architectures with limited programmability such as the NVIDIA Tensor Core units. Halide was in general preferred over similar DSLs and source-to-source compiler

tools due to its active community and already established presence within industrial contexts.

Current Limitations. The work presented in this dissertation can achieve performance competitive to manual solutions across a wide range of real-world applications and state-of-the-art architectures while also generating solutions in the order of seconds. However, its overall efficiency is limited by the scope of knowledge instilled during its design. More specifically, while analytical modeling can in principle capture all application and architecture-specific parameters necessary in order to adequately optimize an imaging pipeline, its efficiency is strictly tied to the nature of the deployment platform.

As hardware architectures continue to evolve, it is not impossible that future platforms will feature entirely different components, the behavior of which is not captured in current models. To this end, our proposed frameworks are limited to CPUs, GPUs and other architectures that can be described through a similar memory hierarchy subsystem connected with various computational units (processing core). In case of entirely different platforms, new models alongside novel heuristics will instead need to be designed.

In a similar fashion, our frameworks consider not only a subset of the available optimization techniques used within the image-processing domain, but also a limited number of transformations available through the Halide language. In an effort to be able to cover a wide range of applications, more specialized transformations such as multi-level or hybrid tiling [25, 36] as well as explicit data placement and loading fall outside the scope of this work, even though they have proven to be highly effective under conditions. However, the proposed models can in principle be extended in order to also consider such transformations.

More specifically, all of the proposed analytical models that were used in Chapters 4, 5 and 6 can potentially benefit from a more sophisticated analysis related to loop unrolling in order to use hardware registers in a more efficient way. As an example, the model defined in Chapter 4 can be extended to consider another level of the memory hierarchy that corresponds to the register file. Furthermore, the GPU scheduler of Chapter 6 can also immensely benefit from explicit data placement of the values associated input buffers or input values across stages. Halide can achieve this through the `.in()` directive, which enables loading of input data to be explicitly scheduled through a wrapper stage or Halide `Func`. Such wrappers can then be fused into their consumers just like a normal producing stage would. When targeting a GPU architecture, the placement of the buffers associated with these wrapper stages can be controlled such that they reside in a specific level of the memory

hierarchy (global memory, shared memory or registers). Applications such as convolutional layers and matrix multiplications that offer high data reuse across loop iterations can use this optimization and avoid redundant loading of input data.

Future-work directions. The aforementioned limitations can be tackled through various future-work directions. These directions can be divided into two categories: (i) approaches that aim to improve the programmability of non-traditional architectures while enabling efficient and automatic code generation and (ii) extended optimization frameworks that attempt to cover a larger search space by incorporating more techniques and code transformations while maintaining a hardware-specific perspective of the optimization process.

Code-generation support for novel architectures: With the increased popularity of computer vision and deep learning in the recent years more and more domain-specific accelerators such as the intel neural processing unit [76], the google TPU [33] or the are being introduced. As most of them often require unique programming models hidden behind various tools and libraries, their usage remains out of scope for the average user and is therefore limited to expert developers capable of efficiently, manually, programming them and integrating them in their products.

Domain-specific languages partially alleviate the above issue by incorporating passes within their compilers, that can both generate code as well as handle all necessary data/code offloading in order to support such architectures. However, current DSLs are often designed using individual intermediate representations and custom back-end code generators (in situations where LLVM is not available). As a result, all effort spent to support such architectures in one of these languages often has to be repeated in order to target a different one. A *unified* IR, shared across all relevant DSLs could instead allow for a much more concise and streamlined development process for future compilers and platforms. To an extent, this is already possible through the Multi-Level Intermediate Representation (MLIR) [42] which attempts to unify the high-level front-end IRs of various (mostly machine learning) frameworks through the use of a level of abstraction called "*flavors*" that operate on tensors. Ideally, this kind of infrastructure should also support variable granularity of computation (i.e., operations on loops instead of tensors) or, in an even more advanced form, enable generation of hardware description languages (HDL).

The future of analytical modeling and auto-scheduling. Traditional analytical models alongside heuristics enable quick and efficient design-space exploration as well as efficient code generation through the al-

gorithms introduced in this thesis. However, the rapid evolution of both in terms of application as well as architecture development may in the future become an obstacle towards automatic efficient code generation through analytical modeling alone. As more and more platform- or application-specific transformations will be needed to achieve performance comparable to manual solutions, generic models will either have to be replaced by domain-specific ones, each specialized in a subset of applications (i.e. BLAS kernels, DNNs, classical vision pipelines), or by hybrid optimization frameworks guided by some form of machine learning models.

Specialized schedulers can in principle achieve near-optimum performance by modeling the behavior of all hardware components, while making use of application-specific knowledge such as memory access patterns, specific data placement and computational intensity to efficiently utilize the platform's resources. On the other hand, a learned model similar to the one presented in [2] has the immense benefit of extensibility. In other words, a learned model can easily be extended to support a new optimization/transformation by repeating the training process. Moreover, such learned models are capable of detecting complex patterns and architectural behaviors of applications, and as a result are also capable of predicting the impact of various optimizations in situations that would be nearly impossible to capture purely through analytical models and heuristics that target the same range of applications. However, as the design space grows larger with the addition of more possible transformations (along with their impact on existing ones), human guidance and heuristics are of vital importance in order to steer the learning process. As a result, hybrid, extensible models that incorporate both human knowledge and artificial intelligence are expected to be a subject of significant research in the field's future.

Finally, future scheduling frameworks should be able to efficiently handle the heterogeneous aspects of modern platforms. To this end, they should be capable of deciding which parts of the applications (or stages of an imaging pipeline) should be offloaded on a specific accelerator. This can be achieved either through a learned model that predicts the performance benefit of an application running on an available accelerator compared to the host architecture or through a traditional heuristic-based analytical model.

Bibliography

[1] Abella, J. Near-Optimal Loop Tiling by Means of Cache Miss Equations and Genetic Algorithms. In *Proceedings of the 2002 International Conference on Parallel Processing Workshops* (Washington, DC, USA, 2002), ICPPW '02, IEEE Computer Society, p. 568.

[2] Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.-M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., and Ragan-Kelley, J. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph. 38*, 4 (July 2019), 121:1–121:12.

[3] Adams, A., Talvala, E.-V., Park, S. H., Jacobs, D. E., Ajdin, B., Gelfand, N., Dolson, J., Vaquero, D., Baek, J., Tico, M., Lensch, H. P. A., Matusik, W., Pulli, K., Horowitz, M., and Levoy, M. The Frankencamera: An Experimental Platform for Computational Photography. In *ACM SIGGRAPH 2010 Papers* (New York, NY, USA, 2010), SIGGRAPH '10, ACM, pp. 29:1–29:12.

[4] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (New York, NY, USA, 2014), PACT '14, ACM, pp. 303–316.

[5] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U. M., and Amarasinghe, S. OpenTuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)* (Aug 2014), pp. 303–315.

[6] Asano, S., Maruyama, T., and Yamaguchi, Y. Performance comparison of FPGA, GPU and CPU in image processing. In *2009 International Conference on Field Programmable Logic and Applications* (Aug 2009), pp. 126–131.

[7] Baghdadi, R., Ray, J., Romdhane, M. B., Del Sozzo, E., Akkas, A., Zhang, Y., Suriana, P., Kamil, S., and Amarasinghe, S. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Piscataway, NJ, USA, 2019), CGO 2019, IEEE Press, pp. 193–205.

[8] Bandishti, V., Pananilath, I., and Bondhugula, U. Tiling Stencil Computations to Maximize Parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 40:1–40:11.

[9] Bao, B., and Ding, C. Defensive loop tiling for shared cache. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Feb 2013), pp. 1–11.

[10] Besard, T., Foket, C., and De Sutter, B. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems 30*, 4 (April 2019), 827–841.

[11] Bondhugula, U. High Performance Code Generation in MLIR: An Early Case Study with GEMM, 2020.

[12] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not. 43*, 6 (June 2008), 101–113.

[13] Chame, J., and Moon, S. A Tile Selection Algorithm for Data Locality and Cache Interference. In *Proceedings of the 13th International Conference on Supercomputing* (New York, NY, USA, 1999), ICS '99, ACM, pp. 492–499.

[14] Chen, J., Paris, S., and Durand, F. Real-time edge-aware image processing with the bilateral grid. *ACM Trans. Graph. 26* (2007), 103.

[15] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2018), OSDI'18, USENIX Association, pp. 579–594.

[16] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cuDNN: Efficient Primitives for Deep Learning. *CoRR abs/1410.0759* (2014).

[17] Cociorva, D., Wilkins, J. W., Lam, C., Baumgartner, G., Ramanujam, J., and Sadayappan, P. Loop Optimization for a Class of Memory-constrained Computations. In *Proceedings of the 15th International Conference on Supercomputing* (New York, NY, USA, 2001), ICS '01, ACM, pp. 103–113.

[18] Coleman, S., and McKinley, K. S. Tile Size Selection Using Cache Organization and Data Layout. *SIGPLAN Not. 30*, 6 (June 1995), 279–290.

[19] Dally, W. J., Turakhia, Y., and Han, S. Domain-Specific Hardware Accelerators. *Commun. ACM 63*, 7 (June 2020), 48–57.

[20] Darbon, J., Cunha, A., Chan, T. F., Osher, S., and Jensen, G. J. Fast nonlocal filtering applied to electron cryomicroscopy. In *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro* (May 2008), pp. 1331–1334.

[21] Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R. C., and Yelick, K. Self adapting linear algebra algorithms and software. In *Proceedings of the IEEE* (2005), p. 2005.

[22] Farbman, Z., Fattal, R., and Lischinski, D. Convolution Pyramids. In *Proceedings of the 2011 SIGGRAPH Asia Conference* (New York, NY, USA, 2011), SA '11, ACM, pp. 175:1–175:8.

[23] Fraguela, B. B., Carmueja, M. G., Andrade, D., Joubert, G. R., Nagel, W. E., Peters, F. J., Plata, O., Tirado, P., Zapata, E., A, B. B. F., A, M. G. C., and A, D. A. Optimal tile size selection guided by analytical models. In *In PARCO* (2005), pp. 565–572.

[24] Goodfellow, I. J., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. http://www.deeplearningbook.org.

[25] Grosser, T., Cohen, A., Holewinski, J., Sadayappan, P., and Verdoolaege, S. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2014), CGO '14, ACM, p. 66–75.

[26] Halide. Halide GitHub Repository (MIT License), 2018. (commit 402171e7a4dfacb0bd93297cbdfb600a325fe745).

[27] Halide. Halide GitHub Repository (MIT License), 2018. (commit a6129313b29a9f434ad28d425af689bcde4f13e7).

[28] Harris, C., and Stephens, M. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference* (1988), pp. 147–151.

[29] Holewinski, J., Pouchet, L.-N., and Sadayappan, P. High-performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing* (New York, NY, USA, 2012), ICS '12, ACM, pp. 311–320.

[30] Holewinski, J., Pouchet, L.-N., and Sadayappan, P. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing* (New York, NY, USA, 2012), ICS '12, ACM, p. 311–320.

[31] Intel Corporation. Intel Math Kernel Library, 2020. version 2020.

[32] Jangda, A., and Bondhugula, U. An Effective Fusion and Tile Size Model for Optimizing Image Processing Pipelines. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2018), PPoPP '18, ACM, pp. 261–275.

[33] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., and et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA '17, ACM, p. 1–12.

[34] Kamil, S., Husbands, P., Oliker, L., Shalf, J., and Yelick, K. Impact of Modern Memory Subsystems on Cache Optimizations for Stencil Computations. In *Proceedings of the 2005 Workshop on Memory System Performance* (New York, NY, USA, 2005), MSP '05, ACM, pp. 36–43.

[35] Kennedy, K., and McKinley, K. S. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing* (Berlin, Heidelberg, 1994), U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., Springer Berlin Heidelberg, pp. 301–320.

[36] Kim, D., Renganarayanan, L., Rostron, D., Rajopadhye, S., and Strout, M. M. Multi-Level Tiling: M for the Price

of One. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2007), SC '07, ACM.

[37] Kim, J., Lee, J. K., and Lee, K. M. Accurate Image Super-Resolution Using Very Deep Convolutional Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2016), pp. 1646–1654.

[38] Knijnenburg, P. M. W., Kisuki, T., Gallivan, K., and O'Boyle, M. F. P. The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling: Research Articles. *Concurr. Comput. : Pract. Exper. 16*, 2-3 (Jan. 2004), 247–270.

[39] Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., and Sadayappan, P. Effective Automatic Parallelization of Stencil Computations. *SIGPLAN Not. 42*, 6 (June 2007), 235–244.

[40] Lam, M. D., Rothberg, E. E., and Wolf, M. E. The Cache Performance and Optimizations of Blocked Algorithms. *SIGPLAN Not. 26*, 4 (Apr. 1991), 63–74.

[41] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (USA, 2004), CGO '04, IEEE Computer Society, p. 75.

[42] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. MLIR: A Compiler Infrastructure for the End of Moore's Law, 2020.

[43] Li, T.-M., Gharbi, M., Adams, A., Durand, F., and Ragan-Kelley, J. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH) 37*, 4 (2018), 139:1–139:13.

[44] Low, T. M., Igual, F. D., Smith, T. M., and Quintana-Orti, E. S. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw. 43*, 2 (Aug. 2016).

[45] Low, T. M., Igual, F. D., Smith, T. M., and Quintana-Orti, E. S. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw. 43*, 2 (Aug. 2016), 12:1–12:18.

[46] Lu, Q., Krishnamoorthy, S., and Sadayappan, P. Combining Analytical and Empirical Approaches in Tuning Matrix Transposition. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2006), PACT '06, ACM, pp. 233–242.

[47] Manjikian, N., and Abdelrahman, T. S. Fusion of Loops for Parallelism and Locality. *IEEE Trans. Parallel Distrib. Syst. 8*, 2 (Feb. 1997), 193–209.

[48] McKinley, K. S., Carr, S., and Tseng, C.-W. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst. 18*, 4 (July 1996), 424–453.

[49] Mehta, S., Beeraka, G., and Yew, P.-C. Tile Size Selection Revisited. *ACM Trans. Archit. Code Optim. 10*, 4 (Dec. 2013), 35:1–35:27.

[50] Mehta, S., Garg, R., Trivedi, N., and Yew, P. *TurboTiling: Leveraging prefetching to boost performance of tiled codes*, vol. 01-03-June-2016. Association for Computing Machinery, 6 2016.

[51] Membarth, R., Reiche, O., Hannig, F., Teich, J., Korner, M., and Eckert, W. HIPAcc: A Domain-Specific Language and Compiler for Image Processing. *IEEE Trans. Parallel Distrib. Syst. 27*, 1 (Jan. 2016), 210–224.

[52] Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J., and Fatahalian, K. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph. 35*, 4 (July 2016), 83:1–83:11.

[53] Mullapudi, R. T., Vasista, V., and Bondhugula, U. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 429–443.

[54] Mullapudi, R. T., Vasista, V., and Bondhugula, U. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGARCH Comput. Archit. News 43*, 1 (Mar. 2015), 429–443.

[55] NVIDIA Corporation. NVIDIA cuBLAS library, 2008. version 10.0.

[56] NVIDIA Corporation. CUDA Templates for Linear Algebra Subroutines, 2017.

[57] NVIDIA Corporation. NVIDIA CUDA Samples, 2017.

[58] NVIDIA Corporation. NVIDIA Tensor Cores, 2017.

[59] NVIDIA Corporation. CUDA Occupancy Calculator, 2019. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.

[60] NVIDIA Corporation. NVIDIA Nsight Compute, 2019. version 2019.5.0.

[61] Olschanowsky, C., Strout, M. M., Guzik, S., Loffeld, J., and Hittinger, J. A Study on Balancing Parallelism, Data Locality, and Recomputation in Existing PDE Solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Piscataway, NJ, USA, 2014), SC '14, IEEE Press, pp. 793–804.

[62] Paris, S., Hasinoff, S. W., and Kautz, J. Local Laplacian Filters: Edge-aware Image Processing with a Laplacian Pyramid. In *ACM SIGGRAPH 2011 Papers* (New York, NY, USA, 2011), SIGGRAPH '11, ACM, pp. 68:1–68:12.

[63] Park, N., Hong, B., and Prasanna, V. K. Analysis of memory hierarchy performance of block data layout. In *Proceedings International Conference on Parallel Processing* (2002), pp. 35–44.

[64] Peemen, M., Mesman, B., and Corporaal, H. Inter-tile Reuse Optimization Applied to Bandwidth Constrained Embedded Accelerators. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition* (San Jose, CA, USA, 2015), DATE '15, EDA Consortium, pp. 169–174.

[65] PolyMage project. PolyMage Repository (Apache 2.0 License 2016), 2016. (commit 0ff0b46456605a5579db09c6ef98cb247dd2131d).

[66] Pouchet, L.-N., Bastoul, C., Cohen, A., and Cavazos, J. Iterative Optimization in the Polyhedral Model: Part Ii, Multidimensional Time. *SIGPLAN Not. 43*, 6 (June 2008), 90–100.

[67] Prajapati, N., Ranasinghe, W., Rajopadhye, S., Andonov, R., Djidjev, H., and Grosser, T. Simple, Accurate, Analytical Time Modeling and Optimal Tile Size Selection for GPGPU Stencils. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2017), PPoPP '17, ACM, pp. 163–177.

[68] Pu, J., Bell, S., Yang, X., Setter, J., Richardson, S., Ragan-Kelley, J., and Horowitz, M. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. Archit. Code Optim. 14*, 3 (Aug. 2017).

[69] Pulli, K., Baksheev, A., Kornyakov, K., and Eruhimov, V. Real-time Computer Vision with OpenCV. *Commun. ACM 55*, 6 (June 2012), 61–69.

[70] Qasem, A., and Kennedy, K. Profitable Loop Fusion and Tiling Using Model-driven Empirical Search. In *Proceedings of the 20th Annual International Conference on Supercomputing* (New York, NY, USA, 2006), ICS '06, ACM, pp. 249–258.

[71] Qiao, B., Reiche, O., Hannig, F., and Teich, J. Automatic Kernel Fusion for Image Processing DSLs. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems* (New York, NY, USA, 2018), SCOPES '18, ACM, p. 76–85.

[72] Qiao, B., Reiche, O., Hannig, F., and Teich, J. From Loop Fusion to Kernel Fusion: A Domain-specific Approach to Locality Optimization. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Piscataway, NJ, USA, 2019), CGO 2019, IEEE Press, pp. 242–253.

[73] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 519–530.

[74] Ravishankar, M., Holewinski, J., and Grover, V. Forma: A DSL for Image Processing Applications to Target GPUs and Multi-core CPUs. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs* (New York, NY, USA, 2015), GPGPU-8, ACM, pp. 109–120.

[75] Rawat, P. S., Hong, C., Ravishankar, M., Grover, V., Pouchet, L.-N., Rountev, A., and Sadayappan, P. Resource Conscious Reuse-Driven Tiling for GPUs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (New York, NY, USA, 2016), PACT '16, ACM, p. 99–111.

[76] Intel Corporation. Intel Neural Network Processor, 2020.

[77] Rhemann, C., Hosni, A., Bleyer, M., Rother, C., and Gelautz, M. Fast cost-volume filtering for visual correspondence and beyond. In *CVPR 2011* (June 2011), pp. 3017–3024.

[78] Shirako, J., Sharma, K., Fauzia, N., Pouchet, L.-N., Ramanujam, J., Sadayappan, P., and Sarkar, V. Analytical Bounds for Optimal Tile Size Selection. In *Proceedings of the 21st International Conference on Compiler Construction* (Berlin, Heidelberg, 2012), CC'12, Springer-Verlag, pp. 101–121.

[79] Sioutas, S., Stuijk, S., Basten, T., Corporaal, H., and Somers, L. Schedule Synthesis for Halide Pipelines on GPUs. *ACM Trans. Archit. Code Optim. 17*, 3 (Aug. 2020).

[80] Sioutas, S., Stuijk, S., Basten, T., Somers, L., and Corporaal, H. Programming Tensor Cores from an Image Processing DSL. In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems* (New York, NY, USA, 2020), SCOPES '20, ACM, p. 36–41.

[81] Sioutas, S., Stuijk, S., Corporaal, H., Basten, T., and Somers, L. Loop Transformations Leveraging Hardware Prefetching. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (New York, NY, USA, 2018), CGO 2018, ACM, pp. 254–264.

[82] Sioutas, S., Stuijk, S., Waeijen, L., Basten, T., Corporaal, H., and Somers, L. Schedule Synthesis for Halide Pipelines Through Reuse Analysis. *ACM Trans. Archit. Code Optim. 16*, 2 (Apr. 2019), 10:1–10:22.

[83] Tavarageri, S., Pouchet, L. N., Ramanujam, J., Rountev, A., and Sadayappan, P. Dynamic selection of tile sizes. In *2011 18th International Conference on High Performance Computing* (Dec 2011), pp. 1–10.

[84] Temam, O., Fricker, C., and Jalby, W. Cache Awareness in Blocking Techniques. In *in Journal of Programming Languages* (1998).

[85] The Regents of the University of California, through Lawrence Berkeley National Laboratory. 'Empirical Roofline Tool (ERT)' Copyright (c), 2019. (commit 96c4bbb41ad178d8d331696bebc2af6245af3e3c).

[86] TVM. TVM GitHub Repository (Apache-2.0 license), 2019. (commit 9ff44969e3b566a8f1a7a50c327f63a3427984420.

[87] Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR abs/1802.04730* (2018).

[88] Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., and Catthoor, F. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim. 9*, 4 (Jan. 2013), 54:1–54:23.

[89] Vocke, S., Corporaal, H., Jordans, R., Corvino, R., and Nas, R. Extending Halide to Improve Software Development for Imaging DSPs. *ACM Trans. Archit. Code Optim. 14*, 3 (Aug. 2017).

[90] Wahib, M., and Maruyama, N. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2014), pp. 191–202.

[91] Wang, G., Lin, Y., and Yi, W. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing* (USA, 2010), GREENCOM-CPSCOM '10, IEEE Computer Society, p. 344–350.

[92] Whaley, R. C., and Dongarra, J. J. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 1998), SC '98, IEEE Computer Society, pp. 1–27.

[93] Whaley, R. C., Petitet, A., and Dongarra, J. J. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing 27* (2001), 3–25.

[94] Williams, S., Waterman, A., and Patterson, D. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM 52*, 4 (Apr. 2009), 65–76.

[95] Wolf, M. E., and Lam, M. S. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1991), PLDI '91, ACM, pp. 30–44.

[96] Wolf, M. E., and Lam, M. S. A data locality optimizing algorithm. pp. 30–44.

[97] Wolf, M. E., Maydan, D. E., and Chen, D.-K. Combining Loop Transformations Considering Caches and Scheduling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (USA, 1996), MICRO 29, IEEE Computer Society, p. 274–286.

[98] Xue, J. Aggressive Loop Fusion for Improving Locality and Parallelism. In *Proceedings of the Third International Conference on Parallel and Distributed Processing and Applications* (Berlin, Heidelberg, 2005), ISPA'05, Springer-Verlag, pp. 224–238.

[99] Yi, Q., and Kennedy, K. Improving Memory Hierarchy Performance through Combined Loop Interchange and Multi-Level Fusion. *The International Journal of High Performance Computing Applications 18*, 2 (2004), 237–253.

[100] Yotov, K., Li, X., Ren, G., Garzaran, M. J. S., Padua, D., Pingali, K., and Stodghill, P. Is Search Really Necessary to Generate High-Performance BLAS? *Proceedings of the IEEE 93*, 2 (Feb 2005), 358–386.

[101] Yuki, T., Renganarayanan, L., Rajopadhye, S., Anderson, C., Eichenberger, A. E., and O'Brien, K. Automatic Creation of Tile Size Selection Models. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2010), CGO '10, ACM, pp. 190–199.

[102] Zhang, L., Carter, J. B., Hsieh, W. C., and McKee, S. A. Memory system support for image processing. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)* (1999), pp. 98–107.

[103] Zhou, X., Giacalone, J.-P., Garzarán, M. J., Kuhn, R. H., Ni, Y., and Padua, D. Hierarchical Overlapped Tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (New York, NY, USA, 2012), CGO '12, ACM, pp. 207–218.

Publications

Main Author

[1] Savvas Sioutas, Sander Stuijk, Twan Basten, Henk Corporaal, and Lou Somers. Schedule Synthesis for Halide Pipelines on GPUs. *ACM Trans. Archit. Code Optim.*, 17(3), August 2020.

[2] Savvas Sioutas, Sander Stuijk, Twan Basten, Lou Somers, and Henk Corporaal. Programming Tensor Cores from an Image Processing DSL. In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '20, page 36–41, New York, NY, USA, 2020. Association for Computing Machinery.

[3] Savvas Sioutas, Sander Stuijk, Henk Corporaal, Twan Basten, and Lou Somers. Loop Transformations Leveraging Hardware Prefetching. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 254–264, New York, NY, USA, 2018. ACM.

[4] Savvas Sioutas, Sander Stuijk, Luc Waeijen, Twan Basten, Henk Corporaal, and Lou Somers. Schedule Synthesis for Halide Pipelines through Reuse Analysis. *ACM Trans. Archit. Code Optim.*, 16(2), April 2019.

Co-author

[1] Luc Waeijen, Savvas Sioutas, Yifan He, Maurice Peemen, and Henk Corporaal. Automatic Memory-Efficient Scheduling of CNNs. In Dionisios N. Pnevmatikatos, Maxime Pelcat, and Matthias Jung, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 387–400, Cham, 2019. Springer International Publishing.

A Tools

A.1 Reuse Scheduler

This section describes the process in order to reproduce the results obtained in the evaluation section of Chapter 5.

1. Dependencies

Hardware Dependencies: An Intel or ARM CPU.

Software Dependencies: To build and run the provided source code the following frameworks are required:

- Clang/LLVM 7.0 or higher (for Linux)
- Linux distribution (tested on Ubuntu 18.04)
- Make 4.1 or higher
- Git 2.17 or higher

2. Installation

Acquiring LLVM:

Linux binaries for LLVM 7.0 along with the matching version of Clang can be found through http://llvm.org/releases/download.html. Both llvm-config and clang must be somewhere in the path.

Acquiring and building Halide with the proposed scheduler:

The source code for Halide with the reuse scheduler can be found through:

```
$ git clone
   https://github.com/TUE-EE-ES/HalideReuseScheduler.git
```

Point Halide to llvm-config and clang:

```
$ export LLVM_CONFIG=<path to llvm>/build/bin/llvm-config
$ export CLANG=<path to llvm>/build/bin/clang
```

To build Halide:

```
$ cd Halide
$ make
$ make distrib
```

3. Benchmarking

This subsection explains the process in order to reproduce the results obtained in the evaluation section of Chapter 5.

To reproduce the results for all benchmarks:

```
$ cd benchmarks
$ source run_tests.sh
```

All runtimes should be listed in a new file named "`results.txt`" located in the `benchmarks` folder.

To run an individual benchmark (e.g. harris) first set up the environment variables needed by the autoscheduler with by exporting the environment variables located in the above script:

```
$ cd harris
$ make test
```

The above process can be repeated for the rest of the applications. All runtimes are expected to have a variation of +- 5% but a similar ratio across each implementation compared to the one seen in the presented figures.

The source code of the reuse scheduler can be found in the `AutoSchedule.cpp` file, within the `src/` directory.

A.2 GPU Scheduler

This section describes the process in order to reproduce the results obtained in the evaluation section of Chapter 6.

1. Dependencies

Hardware Dependencies: A CUDA GPU of at least 3.2 compute capability.

Software Dependencies: To build and run the provided source code the following frameworks are required:

- Clang/LLVM 8.0 or higher (for Linux)
- Linux distribution (tested on Ubuntu 18.04)
- Make 4.1 or higher
- Git 2.17 or higher
- NVIDIA CUDA driver 10.0 or later
- Python 2.7 /w matplotlib and numpy

2. Installation

Acquiring LLVM:

Linux binaries for LLVM 8.0 along with the matching version of Clang can be found through http://llvm.org/releases/download.html. Both llvm-config and clang must be somewhere in the path.

Acquiring and building Halide with AutoGPU:

The source code for Halide with AutoGPU can be found through:

```
$ git clone https://github.com/TUE-EE-ES/HalideAutoGPU.git
```

Point Halide to llvm-config and clang:

```
$ export LLVM_CONFIG=<path to llvm>/build/bin/llvm-config
$ export CLANG=<path to llvm>/build/bin/clang
```

To build Halide:

```
$ cd Halide
$ make
$ make distrib
```

3. Benchmarking

This subsection explains the process in order to reproduce the results obtained in Figures 6.6 and 6.7.

To reproduce the results of Figure 6.6 run the all benchmarks for the RTX GPU and then plot the graphs with matplotlib:

```
$ cd benchmarks
$ source run_tests_2080ti.sh
```

All runtimes should be listed in a new file named "`results_ti.txt`" located in the `benchmarks` folder. To plot the graphs:

```
$ python plot_figures_2080ti.py
```

To reproduce the AGX Xavier results repeat the above process using the AGX scripts instead:

```
$ source run_tests_xavier.sh
```

All runtimes should be listed in a new file named "`results_xavier.txt`" located in the `benchmarks` folder. To plot the graphs:

```
$ python plot_figures_xavier.py
```

To run an individual benchmark (e.g. harris) first set up the environment variables needed by the autoscheduler with:

```
$ cd benchmarks
$ source setup_env.sh
```

Compute Capability of the target platform can be set by changing the `HL_TARGET` environment variable set in the above script. For example changing the target feature `cuda_capability_61` to `cuda_capability_35` changes the target's compute capability from 6.1 to 3.5.

```
$ cd harris
$ make test
```

The above process can be repeated for the rest of the applications. All runtimes are expected to have a variation of +- 5% but a similar ratio across each implementation compared to the one seen in the presented figures.

The source code of the AutoGPU scheduler can be found in the `AutoSchedule.cpp` file, within the `benchmarks/autoscheduler/` directory.

A.3 Tensor Core code generator

1. Dependencies

This section describes the process in order to reproduce the results obtained in the evaluation section of Chapter 7.

Hardware Dependencies: A CUDA GPU with tensor cores is required (Turing - sm75). All experiments were conducted on an NVIDIA RTX 2080Ti GPU.

Software Dependencies: To build and run the provided source code the following frameworks are required:

- Clang/LLVM 9.0 or higher (for Linux)
- Linux 5.0 (tested on Ubuntu 18.04)
- Make 4.1 or higher
- Git 2.17 or higher
- NVIDIA CUDA driver 10.0 or later

2. Installation

Acquiring LLVM:

Linux binaries for LLVM 9.0 along with the matching version of Clang can be found through http://llvm.org/releases/download.html. Both llvm-config and clang must be somewhere in the path.

Acquiring and building Halide TCU:

The source code for Halide with TCU support can be found through:

```
$ git clone https://github.com/TUE-EE-ES/HalideTCU.git
```

Point Halide to llvm-config:

```
$ export LLVM_CONFIG=<path to llvm>/build/bin/llvm-config
```

To build Halide:

```
$ cd Halide
$ make
```

3. Benchmarking

This subsection explains the process in order to reproduce the results obtained in Figures 7.3 and 7.4.

To reproduce the results of Figure 7.3 set the `CUDA_SDK` variable and build the mixed precision matmul benchmark along with the necessary transposition. For the int32 matmul run the benchmark in the `mat_mul_int` folder. The output should correspond to the average execution time for each of the three implementations:

```
$ export CUDA_SDK=<path_to_cuda>
$ cd apps
$ cd mat_mul
$ make test MATRIX_SIZE=1024 TRANSPOSE=1
```

To reproduce the Halide runtimes of Figure 7.4 build the int32 matmul benchmark. For the WMMA runtimes use the imma example code provided by NVIDIA [57]:

```
$ cd apps
$ cd mat_mul_int
$ make test MATRIX_SIZE=1024 TRANSPOSE=0
```

Changing the value of `MATRIX_SIZE` after `make clean` generates an implementation for other problem sizes. All runtimes are expected to have a variation of +- 10% but a similar ratio across each implementation compared to the one seen in the above figures. The new passes and extensions for the TCU code generation can be found in `Halide/src`: `Inject_Tensor_Ops.cpp`, `top_equiv.cpp`, `CodeGen_PTX_Dev.cpp`, `Func.cpp` and smaller additions to other parts of the compiler and runtime.

Acknowledgments

As my journey as a PhD student reaches its end, I cannot help but reminisce on the various experiences that I went through, as well as personal relationships that I developed and how these shaped the person that I am today. To this end, I would like to extend my gratitude to the people that each in their own way, sometimes directly while other times without that being their primary intention, assisted me throughout my PhD studies.

First and foremost I would like to express my gratitude to my first and second promotors, prof. Henk Corporaal and prof. Twan Basten as well as co-promotor dr. Sander Stuijk for their supervision and guidance throughout my career as a PhD candidate. Their feedback during our progress meetings was invaluable and I sincerely cannot fathom reaching this point without their help. I would also like to thank dr. Lou Somers, who would often bring my attention to more practical, down-to-earth issues, whenever any academic, abstract concepts would try to take over my research.

Moreover, I would like to thank the members of the doctoral committee for agreeing to participate in my defense during such difficult times, as well as for taking the time to read my drafts and provide constructive feedback which helped improve the quality of this thesis. A big thank you to the rest of the staff of the `ES` group, especially Marja, the group secretary, for her extensive administrative support whenever it was needed. I also want to thank all members of the `PARsE` group for their patience during my rather boring rants about pixels and compute/store levels whenever I came up with some new scheduling algorithm. Another big thank you goes to the rest of the PhD students and members of the `ES` group, Alessandro, Ilde, Sayandip, Kamlesh, Roel, Luc for the long discussions we've had at the bar without which the pressure and stress I accumulated over various periods might have been too much to bear (thank you Luc for listening to my rather frequent philosophical and existential nonsense!).

Last but not least, I would like to thank my family for supporting me and encouraging me throughout the years: my brother Thomas for paving the way by coming to the Netherlands years before me and later convincing me to pursue a PhD degree at TU/e. My father and mother (and her cooking!) not only for visiting me in Eindhoven as often as they could but also for their patience to my never-ending complaints.

Curriculum Vitae

Savvas Sioutas was born on the 1st of January 1992 in Athens, Greece. He obtained his Diploma in Electrical and Computer Engineering with a specialization in Electronics and Computers at the University of Patras, Greece in 2015. He joined the Electronic Systems group of Eindhoven University of Technology in 2016 as a PhD candidate. The results of this PhD project are presented in this thesis.