Schedule Synthesis for Halide Pipelines on GPUs

SAVVAS SIOUTAS, Eindhoven University of Technology, The Netherlands
SANDER STUIJK, Eindhoven University of Technology, The Netherlands
TWAN BASTEN, Eindhoven University of Technology and ESI, TNO, The Netherlands
HENK CORPORAAL, Eindhoven University of Technology, The Netherlands
LOU SOMERS, Canon Production Printing and Eindhoven University of Technology, The Netherlands

The Halide DSL and compiler have enabled high performance code generation for image processing pipelines targeting heterogeneous architectures through the separation of algorithmic description and optimization schedule. However, automatic schedule generation is currently only possible for multi-core CPU architectures. As a result, expert knowledge is still required when optimizing for platforms with GPU capabilities. In this work, we extend the current Halide Autoscheduler with novel optimization passes in order to efficiently generate schedules for CUDA-based GPU architectures. We evaluate our proposed method across a variety of applications and show that it can achieve performance competitive with that of manually tuned Halide schedules, or in many cases even better performance. Experimental results show that our schedules are on average 10% faster than manual schedules and over 2x faster than previous autoscheduling attempts.

CCS Concepts: • **Software and its engineering → Compilers**; *Domain specific languages.*

Additional Key Words and Phrases: loop optimizations, image processing, Halide, GPU, scheduling

ACM Reference Format:
Savvas Sioutas, Sander Stuijk, Twan Basten, Henk Corporaal, and Lou Somers. 0000. Schedule Synthesis for Halide Pipelines on GPUs. *ACM Trans. Arch. Code Optim.* 0, 0, Article 0 (0000), 24 pages. https://doi.org/0000001.0000001

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 [3]. 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

Authors' addresses: Savvas Sioutas, Eindhoven University of Technology, Eindhoven, The Netherlands, s.sioutas@tue.nl; Sander Stuijk, Eindhoven University of Technology, Eindhoven, The Netherlands, s.stuijk@tue.nl; Twan Basten, Eindhoven University of Technology, ESI, TNO, Eindhoven, The Netherlands, a.a.basten@tue.nl; Henk Corporaal, Eindhoven University of Technology, Eindhoven, The Netherlands, h.corporaal@tue.nl; Lou Somers, Canon Production Printing , Eindhoven University of Technology, The Netherlands, l.j.a.m.somers@tue.nl.

XXXX-XXXX/0000/0-ART0 $15.00
https://doi.org/0000001.0000001

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 [17], 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 [5] 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. Most of these DSLs also employ schedulers that attempt to automatically optimize an algorithm for the given hardware. Such schedulers usually fall into one of the following two categories: analytical models that use heuristics and cost functions to generate a solution [12, 21] and autotuning frameworks that iteratively try different configurations in the search space and choose the best performing one [20, 26].

Halide [20] is one of the most prominent of these DSLs that targets image processing applications. It enables efficient exploration of the design space by separating the algorithmic description of a pipeline from its optimization schedule. Its compiler can generate code for the host as well as any GPU (as well as some extensions) that may be present on the platform. The optimization schedule of an application dictates the transformations that need to be applied on the code in order to maximize performance. Moreover, through the use of special keywords/directives in the schedule, parts of the pipeline may be offloaded to an accelerator without altering the original description of the application. This distinction of functional description and schedule aims to increase code portability, maintainability and readability.

Generating an efficient optimization schedule involves dealing with a trade-off space between parallelism, redundant computation and locality [20]. Due to the near-infinite number of possible optimizations, even for small pipelines, it is extremely challenging to find a point in the design space that results in near-optimal performance. In an attempt to tackle this issue most automatic approaches limit the search space by employing an overlapping tile analysis that attempts to maximize parallelism at the cost of extra recomputation [13]. Furthermore, unlike CPU platforms, 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.

Halide currently employs an automatic scheduler that was first proposed by Mullapudi et al [13] and was later further updated by the community [8]. Recently, a new learned autoscheduler that combines learning and autotuning [1] was used to broaden the search space compared to prior work, in order to generate schedules for multi-core CPU platforms. Many other schedulers were proposed over the years but they are focused at CPU schedule generation and are therefore unable to use the GPU if it is available [23, 24] or exclude key optimizations from their schedule space (e.g. no fusion being considered) [10].

In this work we make the following contributions:

(1) We extend the current autoscheduler of Halide master [8] 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 work 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, while Section 5 demonstrates the experimental results that were obtained. Possible future work and conclusive remarks are discussed in Section 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.

2.1 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 [11]. 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 [18, 28, 29].

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 [9, 22]. 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) [16, 24]. 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.

2.2 Halide Autoscheduling

Automatic scheduling for Halide pipelines has been investigated a number of times in the past. Halide originally used an autotuner [20] that was later replaced with a more optimized one that uses genetic algorithms in order to find an efficient schedule [2]. 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 [13] 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 [8]. 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 [23, 24] tried to extend the search space considered while attempting to model cache and hardware prefetching behaviors. The analytical model proposed by Sioutas et al [24] 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 [13] and Halide master [8] autoschedulers. However, both above models [23, 24] along with the associated heuristics were tuned to CPU behavior with large caches, and due to favoring sliding window optimizations, they are incapable of exploiting the massive parallelism available on GPU architectures without sacrificing performance to thread synchronization overhead.

Finally, Adams et al [1] 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 [10] (29 to 33% faster) in CUDA based platforms but without yet retraining for GPUs. The latter [10] 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 paper extends the current CPU scheduler present in Halide master [8] with new heuristics and an updated analytical model that considers a broader space along with GPU parameters when generating optimization schedules for Halide pipelines.

2.3 Other DSLs and approaches

Besides Halide, there have been several other DSLs with GPU offloading support. Hipacc [12] 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 [19] 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 [21] 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 [14] 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 [6] and Tensor Comprehensions [26]. 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.

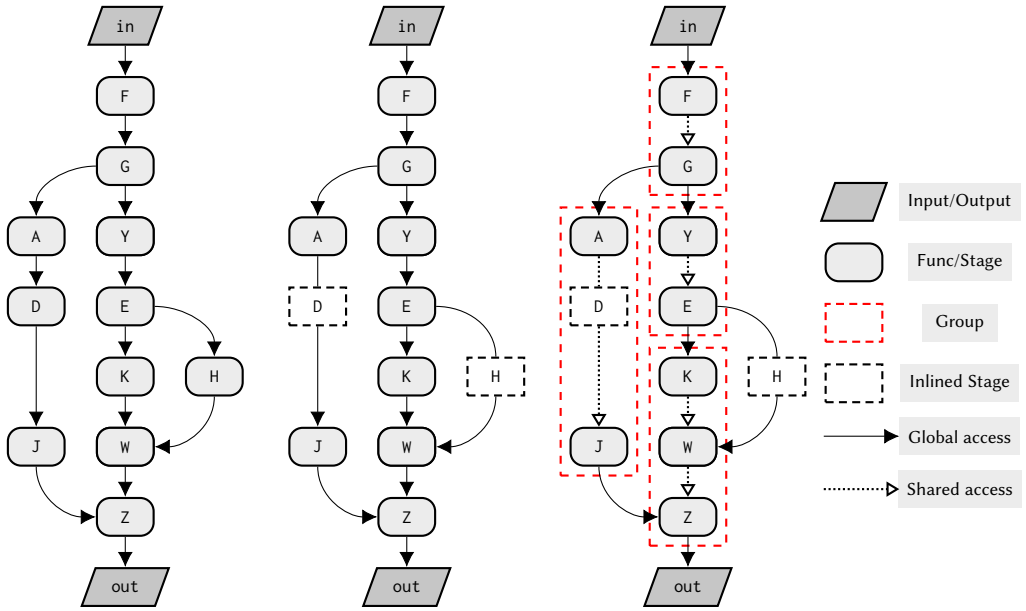
Outside the scope of DSLs, polyhedral compilers are often used to optimize image processing and tensor or stencil operations in GPUs [4, 16, 27]. These compilers employ polyhedral transformations in order to optimize affine programs. They aim to maximize parallelism through proper tile size selection 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.

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 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 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 2a. The `compute_root` scheduling directive tells the compiler to fully compute a stage before moving to the next one. When 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



(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.

Fig. 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.

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 2b. A separate CUDA kernel is launched for each stage and all pixels computed are stored in the global memory. Finally, Figure 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.

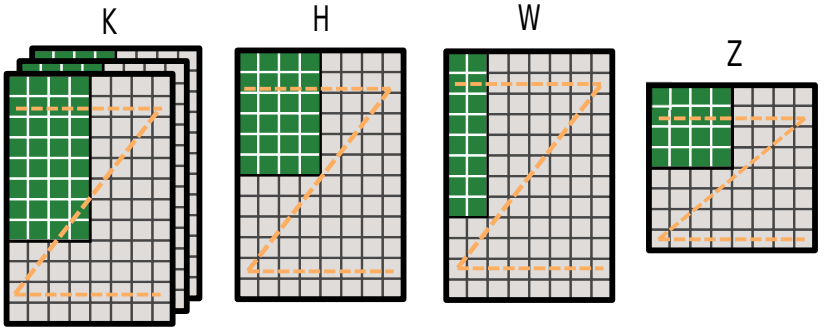
As a consequence, optimizing a pipeline as a whole involves generating schedules that affect both the intra-group as well as inter-group granularity [1, 24]. Inter-group scheduling focuses on the segmentation of the pipeline into groups of stages as well as inlining stages into their consumers such that maximum producer/consumer locality can be achieved. Scheduling stages within a group

```
//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.

Fig. 2. A naive implementation fully computes each stage in a different CUDA kernel and stores all data into the global memory.

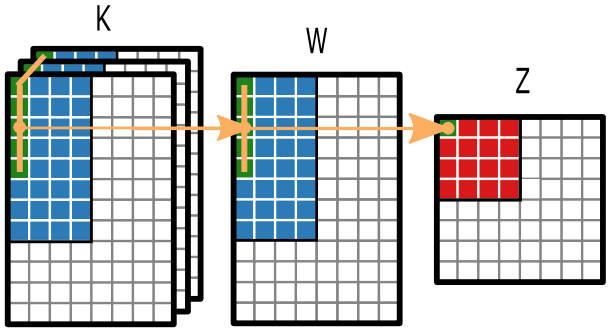
(intra-group) includes optimizations such as tiling, unrolling, selecting the variables that should be assigned as threads/blocks as well as determining the level of the consuming loop nest at which the computation of each producer should be placed. Figure 1b shows the new DAG after stages D and H have 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

```
//function definitions for KWZ group = 1 kernel
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.

Fig. 3. An overlapping tiles schedule computes all pixels needed for one intra-tile iteration (or thread block) and stores them in the shared memory.

seen in Figure 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 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.

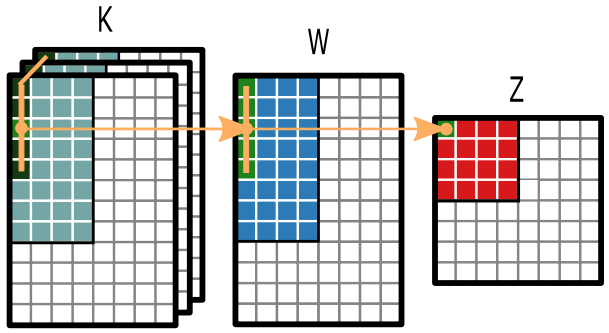
The equivalent CUDA pseudo-code can be found in Figure 3b. As already mentioned a single CUDA kernel is launched for the whole group and all pixels computed in a single intra-tile iteration are stored in the shared memory. Finally, Figure 3c shows a visual representation of the schedule, The blue and 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.

```
//function definitions for KWZ group
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);
W.compute_at(Z,x_o)
  //optimize the member stage
 .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.

Fig. 4. Nested fusion can significantly lower shared memory usage, without increasing redundant computation

An even more optimized implementation is shown in Figure 4 where computation of stage K has been moved inside the inner thread dimension of its consumer W to achieve what we call nested fusion in this paper. 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 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 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 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 [15]. 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.

4 GPU autoscheduler

This section presents the new optimization passes implemented in the Halide master [8] 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 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.

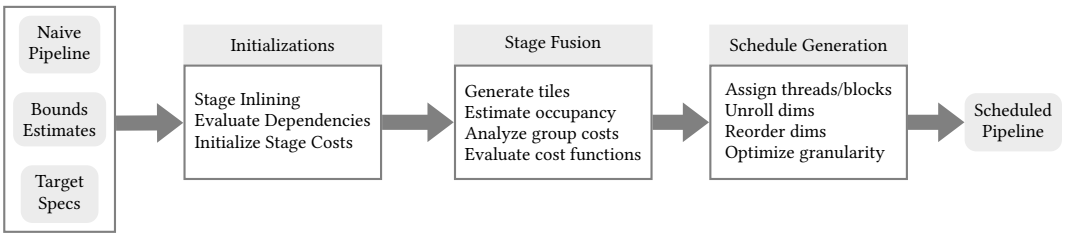


Fig. 5. Basic Scheduling Flow: The scheduler requires the loop bounds estimates along with a target specification description given by the user in order to produce an optimization schedule for a given pipeline. Most of the steps in the compilation flow have been extended in order to support automatic GPU scheduling.

4.1 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.

4.2 Stage Fusion & 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:

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

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

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

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 3 and 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 1. Specifically, A brief description of some of the terms used in the pseudocode is found in Table 1. The terms denoted with capital letters refer to architecture parameters used as constant constraints in the following algorithm.

	Description
`group`	A group of stages merged together into a single kernel.
`tiles`	A 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`	The total size of the shared memory allocation required for this group
`thread_block[dim]`	The thread block size across each `dim` dimension.
`thread_count`	The total number of threads required for a computation.
`active_SMs`	The number of active SMs during a computation.
`active_threads`	The number of active threads during a computation (subset of the total threads).
`occupancy`	The SM occupancy during the computation of a member stage.
`SM_COUNT`	The number of Streaming Multiprocessors in the GPU.
`MAX_THREADS_PER_BLOCK`	Maximum threads per block constraint.
`MAX_SHARED_MEM_PER_BLOCK`	Maximum allowed shared memory per block constraint.
`WARP_SIZE`	The number of threads that correspond to a single warp.

Table 1. Notation of terms used in Listing 1

Listing 1 shows the analysis that evaluates the costs (`memory` and `arithmetic`) for a given tiling configuration of a `group`. Similar to the CPU 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

```
1  Function evaluate_group_costs(group):
2   //evaluate load costs
3   for each stage in group.inputs:
4     memory_cost += stage.memory_cost / consecutive_loads
5   for each stage in group.members:
6     memory_cost += stage.memory_cost
7     footprint += stage.footprint
8   //evaluate GPU terms
9   thread_count=1
10  for each dim in (group.output and loop_threads):
11    thread_block[dim] = tiles[dim]
12    thread_count *= thread_block[dim]
13  //get the extents required for each stage across each dimension based on tile sizes
14  local_bounds = dependence_analysis(group.tiles);
15  //initialize metrics
16  occupancy = 1.0
17  active_threads = MAX_THREADS_PER_BLOCK
18  active_SMs = SM_COUNT
19  for each stage in group.members:
20    stage.thread_count=1
21    for each dim in (stage.dims and loop_threads):
22      stage.thread_block[dim] = local_bounds.stage[dim]
23      stage.thread_count *= stage.thread_block[dim]
24      //keep track of the maximum on each dimension for the overall thread block size
25      thread_block[dim] = max(thread_block[dim], stage.threads[dim])
26    {stage.occupancy, stage.active_threads, stage.active_SMs} = estimate_occupancy(stage.thread_count, footprint)
27    //ensure occupancy above threshold
28    if(stage.occupancy < OCCUPANCY_THRESHOLD)    return invalid_configuration
29    //evaluate arithmetic costs
30    arithmetic_cost += stage.arithmetic_cost / (stage.occupancy * stage.active_threads)
31    occupancy = min(occupancy, stage.occupancy)
32    active_threads = min(active_threads, stage.active_threads)
33    active_SMs = min(active_SMs, stage.active_SMs)
34    thread_count = max(thread_count, stage.thread_count)
35  //ensure resource requirements within target constraints
36  if(thread_count % WARP_SIZE != 0 || thread_count > MAX_THREADS_PER_BLOCK ||
37     active_SMs < SM_COUNT || footprint > MAX_SHARED_MEM_PER_BLOCK )  return invalid_configuration
38  //evaluate total cost
39  group_cost =  arithmetic_cost + memory_cost / (occupancy * active_threads)
40  return group_cost
```

Listing 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.

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 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 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 [15] 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 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.

4.3 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 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 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 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 5. On the example seen in Figure 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.

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 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 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 [1, 13].

	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 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 3. Benchmarks, corresponding number of channels, functional stage and compile time using AutoGPU respectively.

5.1 Halide GPU Scheduling

We compare our solutions to the manual schedules obtained from the Halide official repository [8] as well as the ones generated by the Li et al scheduler [10]. 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 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 shows the results obtained with the NVIDIA RTX 2080Ti platform. Our solutions outperform the Li et al scheduler [10] 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 [10] 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

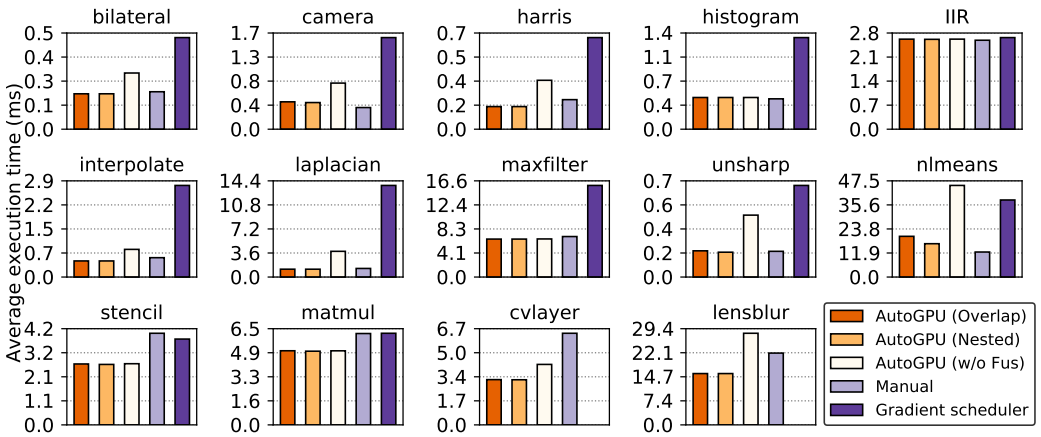


Fig. 6. Average Execution time (ms) NVIDIA RTX 2080 Ti
Comparison of the average runtime of our proposed method (without stage fusion, with overlapped tiling and with nested fusion applied on all groups) with the manual tuned Halide schedules and the Li et al autoscheduler [10]

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).

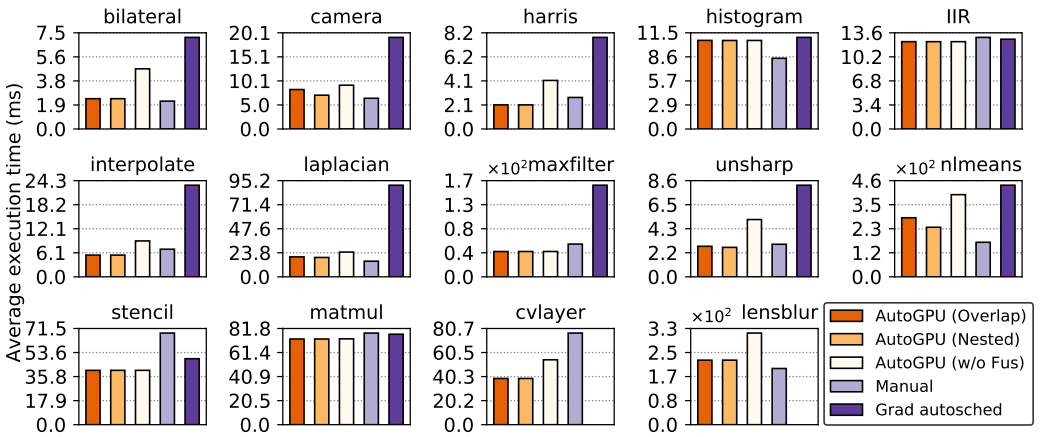


Fig. 7. Average Execution time (ms) NVIDIA AGX Xavier
Comparison of the average runtime of our proposed method (without stage fusion, with overlapped tiling and with nested fusion applied on all groups) with the manual tuned Halide schedules and the Li et al autoscheduler [10]

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 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 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 [10].

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 3.

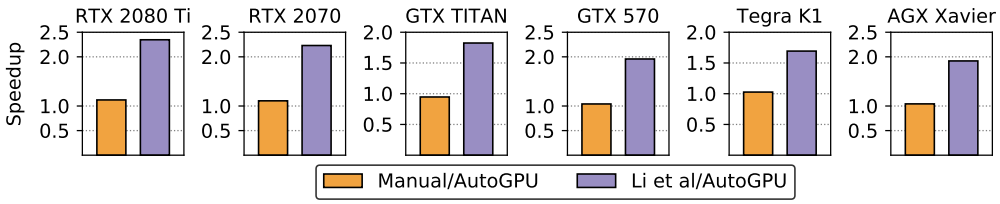


Fig. 8. Speedup of AutoGPU compared to manual and Li et al scheduling: AutoGPU refers to our scheduler when all optimization passes are enabled.

In order to further investigate our results, the roofline model for the RTX 2080 Ti platform [30] was derived for six of the benchmarks as shown in Figure 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 [7] in order to count the number of DRAM (and other memories for the hierarchical roofline) transactions, and Floating ooint Operations (FLOPs). Peak performance and bandwidth was measured using the Empirical Roofline Toolkit (ERT) [25].

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 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, 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

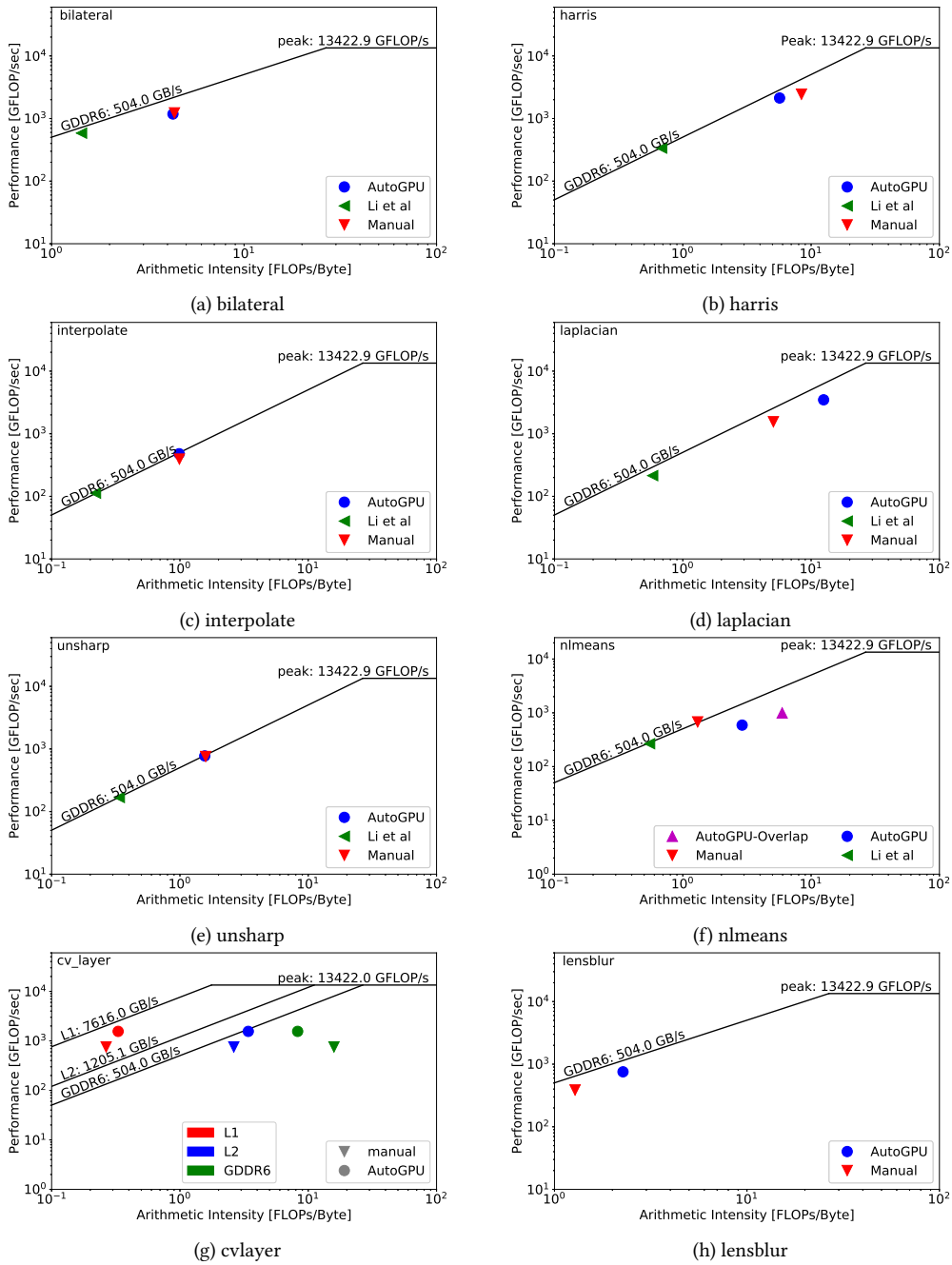


Fig. 9. Roofline models for a subset of the applications used as benchmarks. The ceiling values correspond to the maximum achieved memory bandwidth of the RTX 2080 Ti architecture and the maximum achievable performance.

the ones presented in Figure 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 [10] was unable to generate valid schedules in a few benchmarks due to the reduced maximum threads/memory per block constraints.

5.2 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 [19] 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 run-time, 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.[1]

6 Conclusions and future work

In this work 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 [1]. 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.

References

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (July 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd*

[1] No manual Halide, or other implementations were provided for this network.

International Conference on Parallel Architectures and Compilation (PACT '14). ACM, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

[3] S. Asano, T. Maruyama, and Y. Yamaguchi. 2009. Performance comparison of FPGA, GPU and CPU in image processing. In *2009 International Conference on Field Programmable Logic and Applications*. 126–131. https://doi.org/10.1109/FPL.2009.5272532

[4] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 193–205. http://dl.acm.org/citation.cfm?id=3314872.3314896

[5] T. Besard, C. Foket, and B. De Sutter. 2019. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (April 2019), 827–841. https://doi.org/10.1109/TPDS.2018.2872064

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

[7] NVIDIA Corporation. 2019. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute-2019_5 version 2019.5.0.

[8] Halide. 2018. Halide GitHub Repository (MIT License). https://github.com/halide/Halide (commit a6129313b29a9f434ad28d425af689bcde4f13e7).

[9] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. Association for Computing Machinery, New York, NY, USA, 311–320. https://doi.org/10.1145/2304576.2304619

[10] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)* 37, 4 (2018), 139:1–139:13.

[11] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. 1996. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.* 18, 4 (July 1996), 424–453. https://doi.org/10.1145/233561.233564

[12] Richard Membarth, Oliver Reiche, Frank Hannig, Jurgen Teich, Mario Korner, and Wieland Eckert. 2016. HIPAcc: A Domain-Specific Language and Compiler for Image Processing. *IEEE Trans. Parallel Distrib. Syst.* 27, 1 (Jan. 2016), 210–224. https://doi.org/10.1109/TPDS.2015.2394802

[13] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (July 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

[14] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 429–443. https://doi.org/10.1145/2786763.2694364

[15] Nvidia. 2019. Cuda occupancy calculator. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html

[16] Nirmal Prajapati, Waruna Ranasinghe, Sanjay Rajopadhye, Rumen Andonov, Hristo Djidjev, and Tobias Grosser. 2017. 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 (PPoPP '17)*. ACM, New York, NY, USA, 163–177. https://doi.org/10.1145/3018743.3018744

[17] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. 2012. Real-time Computer Vision with OpenCV. *Commun. ACM* 55, 6 (June 2012), 61–69. https://doi.org/10.1145/2184319.2184337

[18] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. 2018. Automatic Kernel Fusion for Image Processing DSLs. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES '18)*. Association for Computing Machinery, New York, NY, USA, 76–85. https://doi.org/10.1145/3207719.3207723

[19] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. 2019. 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 (CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 242–253. http://dl.acm.org/citation.cfm?id=3314872.3314901

[20] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. 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 (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[21] Mahesh Ravishankar, Justin Holewinski, and Vinod Grover. 2015. 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 (GPGPU-8)*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/2716282.2716290

[22] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noel Pouchet, Atanas Rountev, and P. Sadayappan. 2016. Resource Conscious Reuse-Driven Tiling for GPUs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. Association for Computing Machinery, New York, NY, USA, 99–111. https://doi.org/10.1145/2967938.2967967

[23] Savvas Sioutas, Sander Stuijk, Henk Corporaal, Twan Basten, and Lou Somers. 2018. Loop Transformations Leveraging Hardware Prefetching. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 254–264. https://doi.org/10.1145/3168823

[24] Savvas Sioutas, Sander Stuijk, Luc Waeijen, Twan Basten, Henk Corporaal, and Lou Somers. 2019. Schedule Synthesis for Halide Pipelines Through Reuse Analysis. *ACM Trans. Archit. Code Optim.* 16, 2, Article 10 (April 2019), 22 pages. https://doi.org/10.1145/3310248

[25] through Lawrence Berkeley National Laboratory The Regents of the University of California. 2019. 'Empirical Roofline Tool (ERT)' Copyright (c). https://bitbucket.org/berkeleylab/cs-roofline-toolkit/src/master/ (commit 96c4bbb41ad178d8d331696bebc2af6245af3e3c).

[26] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[27] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[28] M. Wahib and N. Maruyama. 2014. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 191–202. https://doi.org/10.1109/SC.2014.21

[29] Guibin Wang, YiSong Lin, and Wei Yi. 2010. 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 (GREENCOM-CPSCOM '10)*. IEEE Computer Society, USA, 344–350. https://doi.org/10.1109/GreenCom-CPSCom.2010.102

[30] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785

A Sources

1. Dependencies
This section describes the process in order to reproduce the results obtained in the Evaluation section of the paper.

(1) *Hardware Dependencies*: A CUDA GPU of at least 3.2 compute capability.
(2) *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

(1) *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.
(2) *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 and 7.

(1) To reproduce the results of Figure 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
```

(2) 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 `cuda_capability_61` target feature 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 located in `benchmarks/autoscheduler/`.