A Fast GPU Schedule For À-Trous Wavelet-Based Denoisers

REINER DOLP, JOHANNES HANIKA, and CARSTEN DACHSBACHER,
Karlsruhe Institute of Technology, Germany

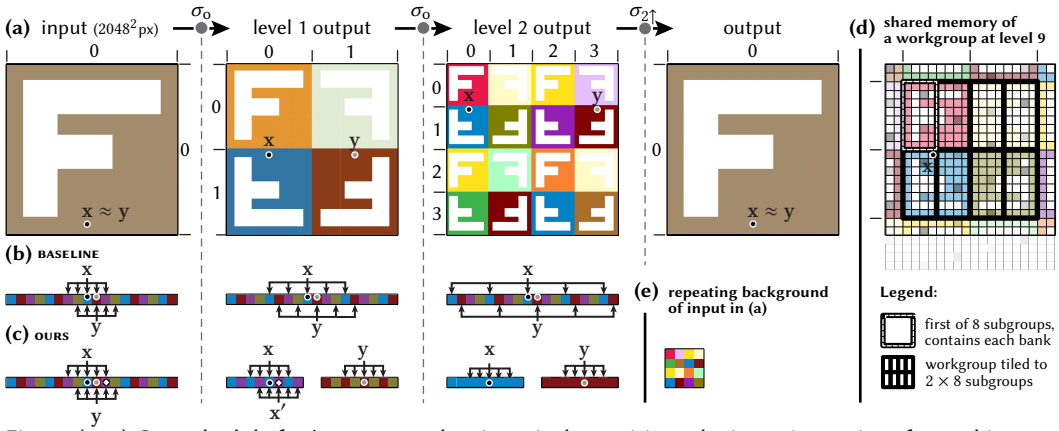

Fig. 1. **(a+e)** Our schedule for à-trous wavelets iteratively partitions the input image into four subimages to improve cache hit rates and consequently shader runtimes. **(b)** The baseline maintains the original pixel coordinates x, y causing neighboring threads to accesses a growing number of increasingly disjoint and dilated pixels. **(c)** In contrast, ours updates pixel coordinates to apply each iteration with an undilated kernel. Further, permitting shared memory caching in all iterations. **(d)** Subgroups within a workgroup are reshaped to rectangular regions. These are constructed in such a way that shifting them anywhere in the shared memory resident pixel tile does not result in a bank conflict.

Given limitations of contemporary graphics hardware, real-time ray-traced global illumination is only estimated using a few samples per pixel. This consequently causes stochastic noise in the resulting frame sequences which requires wide filter support during denoising for temporally stable estimates. The edge avoiding à-trous wavelet transform amortizes runtime cost by hierarchical filtering using a constant number of increasingly dilated taps in each iteration. While the number of taps stays constant, the runtime of each iteration increases in these usually memory-throughput bound shaders with increasing dilation, because the increasing non-locality negatively impacts cache hit rates. We present a scheduling approach that optimizes usage of the memory subsystem by permutating global invocation indices in such a way that each wavelet filter iteration is applied through undilated taps. In contrast to prior approaches, our method has identical performance characteristics in each iteration, effectively decreasing maintenance cost and improving performance predictability. Furthermore, we are able to leverage on-chip memory and hardware texture interpolation. Our permutation strategy is trivial to integrate into existing wavelet filters as a permutation before and after each level of the wavelet filter. We achieve speedups between 1.3 and 3.8 for usual wavelet configurations in Monte Carlo denoising and computational photography.

Authors' address: Reiner Dolp, reiner.dolp@kit.edu; Johannes Hanika, hanika@kit.edu; Carsten Dachsbacher, dachsbacher@kit.edu; Karlsruhe Institute of Technology, Am Fasanengarten 5, Karlsruhe, Germany, 76131.

2577-6193/2024/5-ART1 $15.00
https://doi.org/10.1145/3651299

CCS Concepts: • **Computing methodologies** → **Graphics processors**; **Image processing**; *Ray tracing*; *Computational photography*.

ACM Reference Format:
Reiner Dolp, Johannes Hanika, and Carsten Dachsbacher. 2024. A Fast GPU Schedule For À-Trous Wavelet-Based Denoisers. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 1, Article 1 (May 2024), 19 pages. https://doi.org/10.1145/3651299

1 INTRODUCTION

Image-space denoisers operate on fully accumulated samples. They are a popular choice as they are easy to integrate as a black-box post-processing step. However, even the runtime cost of denoisers explicitly designed for real-time applications is often prohibitive, leading people to amortize cost by combining denoising with upscaling [Kelly et al. 2021; Thomas et al. 2022], through sparse-sampling [Willberger et al. 2019; Zhdan 2021], quantization [Thomas et al. 2020], or by reducing the support of their filter kernels [Schied et al. 2018].

A common building block of real-time denoisers are à-trous wavelets [Dammertz et al. 2010], popularized in real-time rendering through spatio-temporal variance guided filtering (SVGF) [Schied et al. 2017]. À-trous wavelets maintain a constant number of filtering taps, but iteratively increase support of the filtering kernel by dilation (see fig. 1(c)). However, this constant number of filtering taps does not lead to a constant runtime across all wavelet iterations, due to decreasing reuse between dilated taps with each recursion level and consequently, lower cache hit rates and higher memory instruction latency (see section 4.2).

```
(value center v_x) = ping.read(σ_i(x));
(value output v_o, weight w) = 0, 0;
for ny = −r; ny < r; ny++ do
    for nx = −r; nx < r; nx++ do
        (offset o) = vec2(nx,ny);
        v_n = ping.read(σ_i(x + s * o));
        v_o, w += f_e(v_x, v_n);
    end
end
pong.write(σ_o(x), v_o/w);
```

Fig. 2. Baseline implementation of a shader applying an à-trous filter iteration l with filter radius r, kernel f_e, and dilation $s = 2^l$ to pixel $\mathbf{x}$ in invocation $\mathbf{x}$. Our method follows the idea that the mapping of pixels to invocations may be positively influenced by adding index access functions $\sigma_{i,o}$.

Image-space denoisers are usually memory-bound gathering operations parallelized by ping-pong buffering. Hence, computations between pixels are embarrassingly parallel, providing maximal flexibility to optimize memory throughput through scheduling, i.e. more efficient ordering of (memory) instructions without changes to the algorithm. More concretely, such a scheduling approach may be implemented by altering the mapping between pixel indices and invocations by adding index access functions $\sigma_{i,o}$ as shown in fig. 2.

Recent works [Hasselgren et al. 2020; Thomas et al. 2022, 2020] compare favorably against SVGF when their reconstruction quality to reconstruction time tradeoff is considered. However, these implementations rely on non-portable optimizers and runtimes, while comparing against straightforward, portable wavelet filters implemented without considering efficient usage of GPU hardware resources through scheduling.

In this work, we propose several performance-motivated changes the schedule of à-trous wavelets. These changes may be implemented through index access functions $\sigma_{i,o}$. The proposed index access functions are loop invariant. Hence, our method is trivial to integrate into existing wavelet-based denoisers by addition of a preamble and postamble to the baseline schedule (cf. fig. 4). Our contributions are:

- A scheduling strategy without memory overhead localizing global memory accesses in an à-trous wavelet scheme (Section 3.1) through a 2D embedding (Section 3.2) that is oblivious to signal boundaries (Section 3.4).

- Apart from optimizing cache hit rates and thus shader runtimes, this coalesced schedule enables usage of shared memory, i.e. fast on-chip memory with a user-controlled caching strategy, resulting in further runtime improvements (Section 3.5).
- Motivated by the high memory consumption of auxiliary feature-guided denoisers, a shared memory permutation strategy for 2D stencils to eliminate shared memory bank conflicts without introducing unused memory for padding (Section 3.5).

2 PRIOR WORK

Research in Monte Carlo denoising has focused on the design of novel denoising architectures targeting different sample counts and consequently varying runtime budgets for real-time [Schied et al. 2017; Thomas et al. 2022], interactive [Chaitanya et al. 2017] or offline rendering [Vogels et al. 2018]. As our work focuses on scheduling, we review prior work from a structural perspective, i.e. using their data dependencies. For a general survey of denoising algorithms in Monte Carlo rendering, we refer to the surveys of Zwicker et al. [Zwicker et al. 2015] and Sen et al. [Sen et al. 2015].

À-trous Wavelets For Denoising. Edge-avoiding, undecimated wavelets were introduced to computer graphics by Fattal [Fattal 2009] in the context of computational photography. À-trous wavelets were first employed for Monte Carlo Denoising by Dammertz et al. [Dammertz et al. 2010]. Schied et al. [Schied et al. 2017] extended this approach to real-time rendering through temporal reprojection. By estimating variance using moments during reprojection and within the downward pass of the wavelet, they are able to estimate results using two alternating buffers without an upwards pass assembling all wavelets to a final estimate. Their following work [Schied et al. 2018] reduces the support of the wavelet as a runtime optimization. Hanika et al. [Hanika et al. 2011] use à-trous wavelets for image denoising without feature channels. Their approach preserves all intermediate buffers of the recursive spatial (downward) pass and executes an upward pass to assemble the final estimate. Given the upward pass only relies on the previously computed wavelets from the downward pass, the whole upward pass can be implemented in a single shader that concurrently reads all results from the downward pass. The implementations proposed by these works [Dammertz et al. 2010; Schied et al. 2017, 2018] and applied in industry implementations [Nvidia Inc. 2023] corresponds to the baseline implementation shown in fig. 2. In contrast to undilated filters, à-trous wavelets in the baseline schedule cannot exploit commonly applied optimizations for gathering based post-processing effects, such as hardware texture interpolation [Kawase 2003] and on-chip memory resident caching [Nvidia Inc. 2023] after the first filter iteration. Our method enables these optimizations as undecimated wavelets are applied through undilated filters.

Convolutional Neural Networks in Denoising. More recently research focus has shifted towards learned approaches using convolutional neural networks (CNNs) [Bako et al. 2017; Chaitanya et al. 2017; Kalantari et al. 2015; Thomas et al. 2022, 2020; Vogels et al. 2018; Xu et al. 2019]. A commonly employed architecture is the U-Net [Ronneberger et al. 2015]. Instead of direct prediction of the output image, kernel prediction [Bako et al. 2017; Vogels et al. 2018] computes a pairwise pixel similarity and then applies the similarity score encoded in a filter kernel to the input image. In the context of neural networks, undecimated wavelets may be modelled as increasingly dilated convolutions without pooling; while layers in pooling architectures such as U-Nets are dual to decimated wavelet iterations. Our method arranges undecimated wavelets as a tensor stacking decimated wavelets along the feature dimension. As such our method arranges undecimated wavelets in the memory layout commonly seen in neural network architectures.

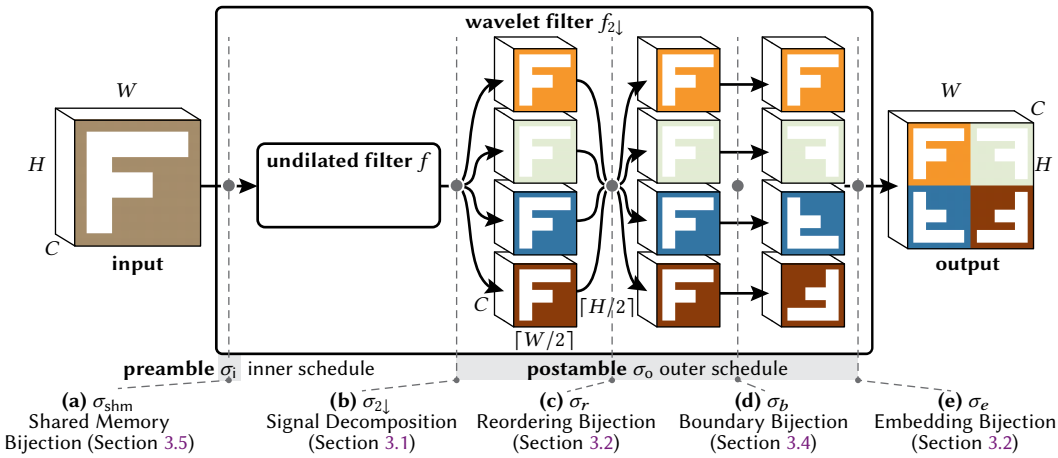

Fig. 3. Overview of our scheduling approach for a single (inner) filter iteration, which is repeatedly applied to form the full à-trous wavelet-based denoiser in fig. 1(a). A shader running an undilated filter kernel is augmented with loop-invariant bijections applied to the invocation indices before (preamble) and after (postamble) the filter kernel loop to implement an à-trous filter. The postamble scatters write locations in order to localize global memory read-locations of subsequent iterations without prior application of a bijection in the preamble. Subsequent iterations of the shader are able to ignore boundaries in their input since our embedding approximates texture mirroring for out of bounds memory accesses.

Input Features. There is a large variability in denoisers, as denoisers proposed in literature are usually heavily adapted to fit the rendering budgets and properties of the rendering engine or task [Barré-Brisebois et al. 2019; Boksansky et al. 2019; Kelly et al. 2021; Willberger et al. 2019]. Denoisers usually receive auxiliary features computed as a byproduct of the rendering process—such as a G-buffer—packed in an engine specific datagram. Further, samples may be separated into effects [Hofmann et al. 2023; Schied et al. 2017; Zhdan 2021] as more efficient denoisers are known for a path subspace. À-trous wavelets are popular for indirect diffuse lighting, where usually large filter radii are required.

Scheduling of Array Processing Code. Analysis and optimization through affine functions has a long tradition in compiler theory. Our proposed bank conflict resolution strategy is designed around the greatest common divisor (GCD) test [Muchnick 1997], a test for loop-carried data dependencies. Modern automatic parallelization and auto-vectorization leverages the polyhedral model [Barré-Brisebois et al. 2019; Grosser et al. 2011; Vasilache et al. 2018], which lifts loop nests into a system of affine constraints ($\mathbb{Z}$-polyhedra) representing every possible dynamic execution instance. While motivated by these approaches, our objective is to specifically design a manual optimization method as detailed in the next section.

3 METHOD

We give an overview of our complete method as a sequence of pixel coordinate transformations implementing the proposed index access functions along with an example implementation, before discussing each step of this sequence in detail (Sections 3.1 to 3.5).

Model. Independent of their distribution to buffers and images, our method transforms all per-pixel data identically. Thus, we assume input to the denoiser is given as a 3-dimensional array $I \in H \times W \times C$ where $H \times W$ are the dimensions of the input image and C are the image channels

along with auxiliary features An iterative denoiser reconstructs I by applying multiple iterations of the filter f. Each iteration is a stencil computation, that integrates pixels from a neighborhood N_x into each pixel $x \in 1 \times 1 \times C$—for example, through a weighted average $\sum_{y \in N_x} f_e(x, y)$ using a pixelwise similarity function f_e as shown in fig. 2. We first describe the application of our method to a shader implementing an undilated filter iteration f, resp. the first iteration of an à-trous wavelet. We then show that an iterative application of this shader results in the desired à-trous filtering scheme.

Undilated Filter Iteration. The shader $f_{2\downarrow}$ modified through our scheduling method computes $(f \circ \sigma)(I)$, which is the filtered image $f(I)$ with an additional memory layout transformation σ applied to the output image. The bijection σ takes the array index of a pixel in the input I and maps it to another index in the output array. From a high-level, the proposed σ subdivides the output of f into four subimages. In contrast to the baseline, this allows pixels in the *next* iteration to be accessed without dilation increasing memory subsystem performance. This σ is compatible with translation. Consequently, allowing us to transform all accessed memory indices at once, by inserting a preamble σ_i before, and a postamble σ_o after the stencil loop. This is equivalent to transforming the invocation index $\mathbf{x}$ directly instead of each memory access—reducing the number of computations of σ by a factor of $||N_\mathbf{x}||$. The adapted shader $f_{2\downarrow}$ is shown in fig. 3. The (optional) preamble σ_i optimizes shared memory throughput using a bijection σ_shm to resolve bank conflicts without row padding (Section 3.5), which is especially desirable if many auxiliary features C are cached for each pixel. In essence, σ_shm reshapes subgroups within each workgroup to rectangular regions. These are constructed in such a way that their translation within the stencil loop does not result in bank conflicts within the shared memory resident pixel tile (see Figure 1(d)). The postamble σ_o is defined as the composition of multiple bijective functions $\sigma_\text{o} = \sigma_{2\downarrow} \circ \sigma_r \circ \sigma_b \circ \sigma_e$. The bijection $\sigma_{2\downarrow}$ performs the mapping between undilated and à-trous filters by contracting even and odd signals into four independent output images (Section 3.1). The bijection σ_r permutes the order of the output images to influence σ_e, which arranges the output images as subimages at collision-free positions in the output (Section 3.2). The (optional) bijection σ_b flips the coordinate systems of the arranged subimages in order to approximate texture mirroring (Section 3.4). This approximation ensures memory accesses falling outside of the current subimage fall into similar regions in neighboring subimages, allowing us to omit the overhead of additional boundary checks in the stencil loop.

Iterative Filtering. An iterative application of σ is shown in Figure 1(a). Apart from the last iteration l_max, which restores the original image layout through $\sigma_{l_\text{max}\uparrow}$ (Section 3.3), each iteration applies $f \circ \sigma$, iteratively partitioning each subimage embeded in the input image. As a result, contracting memory accesses in each iteration (Figure 1(b+c)) to an undilated filter. As all iterations are applied through an undilated filter, each iteration has identical performance characteristics (cf. section 4.2).

Discussion. Our method delocalizes write operations to localize read operations as read operations in a stencil filter have higher multiplicity. At the same time, this places most of the computational cost of σ into the postamble σ_o, which is independent of the stencil loop. Consequently, overhead of σ_o may be hidden through instruction-level parallelism. Opposed to a naive recursive application of $\sigma_{2\downarrow}$ to each output image, we embed subimages for two reasons. First, to allow memory reuse by ping-pong buffering. Second, to optimize occupancy. Through the embedding, a single shader dispatch processes all subimages concurrently without introducing additional inactive invocations through quantization to the workgroup size at subimage boundaries (see fig. 1(d)). In some variations,

```
   // resolve shared memory bank conflicts within the workgroup using σ_shm (Section 3.5)
1  (Workgroup Local Invocation Index lid).xy = lid.yx;                                  preamble σ_i
2  (Global Invocation Index gid) =(2D Workgroup Index vec2(x_q, h_q)) * (Workgroup Size W_q × H_q)) + lid.xy;
3  preload to shared memory tile[H_g+2*(radius r)][W_g+2*(radius r)];                   undilated filter f
4  (value v_x of center pixel x) = tile[lid+(radius r)]; (value output v_o, weight w) = 0, 0;
5  for ny = −r; ny < r; ny++ do
6      for nx = −r; nx < r; nx++ do     // loop nest visiting each (contracted) neighbour (nx, ny) ∈ N_x
7          (offset o) = (radius r) + vec2(nx,ny);              // multiplication with dilation s removed
8          v_o, w += f_e(v_x, tile[lid+offset]);
9      end
10 end
11 if itr ≠ l_max then                                                              postamble σ_o
       // contract wavelets with σ_2↓ according to wavelet decomposition (Section 3.1)
12     hid = ⌊gid/2⌋;
13     odd = gid& 1; // approximate texture mirroring through σ_b for the chosen embedding order σ_r
           (Section 3.4). The chosen σ_r is a no-op (Section 3.2).
14     if itr == 0 && odd then hid = ⌊size/2⌋-1-hid;
       // embed subimages of the wavelet decomposition in output of ping-pong buffers using σ_e
           (Section 3.2), carry read only data
15     gid = hid + odd*⌊size/2⌋;
16 else
       // invert layout in last iteration using σ_{l_max}↑ (Section 3.3)
17     (subimage size sizew) = ⌊(image size size)/s⌋;
18     (subimage id o) = ⌊gid/sizew⌋; (subimage local pixel index wid) = gid - sizew*o;
19     gid = o + wid*s;                                                  // interleave subimages
20     (quadrant local pixel index hid) = ⌊gid/2⌋; (quadrant id odd) = gid & 1;
21     if odd then hid = ⌊size/2⌋-1-hid; gid = hid*2+odd;                          // undo mirroring
22 end
23 pong.write(gid, v_o/w);
```

Fig. 4. Minimal implementation of our method for a filter radius $r = 2$ and workgroup size of 8×8, in which case our bank conflict resolution strategy simplifies to a transposition of the thread indices. The embedding is arranging subimages in ascending order on a uniform grid (Section 3.2).

such as the example discussed next, the embedding makes our method invariant to the iteration index l.

Example. In the remainder of this paper, we discuss each bijection in detail along with implementation variants and tradeoffs. As a motivational example, consider the concrete implementation variant of our method in fig. 4.

The original implementation of f is shown in lines 5 to 10. Assuming the original implementation is an à-trous filter in the baseline schedule, we remove the filter dilation $s = 2^l$ increasing with each iteration l to contract the neighborhood $N_\mathbf{x}$ (Line 7). Since our method ensures a constant working set size equal to the first undilated wavelet iteration, we further access neighboring pixels through shared memory. Preloading a tile from global to shared memory is a common operation and elided for brevity (Line 3). As a result, we arrive at a reasonably optimized implementation of an undilated filter (Lines 3 to 10). The inserted preamble implements a special case of our permutation-based bank conflict resolution σ_{shm}, which for the assumed workgroup shape of 8×8 can be implemented as a transposition of local invocation indices (Lines 1 to 2). More generally, σ_{shm} reshapes subgroups from the row-major order of the baseline schedule to a rectangular tiling. In the postamble, the bijection $\sigma_{2\downarrow}$ (Line 13) decomposes the output signal into even and odd signals along each dimension

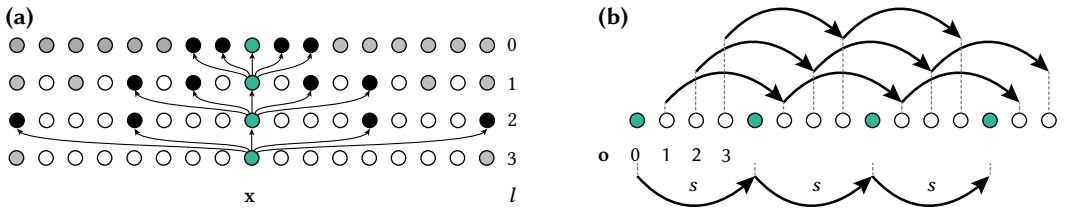

Fig. 5. **(a)** Spatial filtering schema of the one-dimensional à-trous wavelet transform for a single input pixel and filter radius $r = 2$. The number of sampled neighbors with non-zero coefficients (black) is constant in each level l, while the dilation s increases. Arrows show data dependencies between levels l. **(b)** Interpretation of an affine index access function $f(i) = si + o$. The dilation s, as shown here for a one dimensional à-trous wavelet in iteration $l = 2$, partitions space into s disjunct subsets. The offset $\mathbf{o}$, resp. loop initialization, uniquely identifies each subset $[\mathbf{o}]$. Each chain of arrows represents a single subset, whereas subset $[0]$ is highlighted in color. The cardinality of each subset differs by at most one if the length of the data set is not an integer multiple of s.

to form a set of shifted, decimated signals differing in their offset, i.e. the four subimages per input (sub)image. The bijection σ_e (Line 15) embeds the decomposed signals on a rectilinear grid. In this implementation variant, σ_r arranges subimages in ascending order, sorted by their offset in the input image, in which case its sufficient to apply the bijection σ_b (Line 14) approximating texture mirroring for the embedded subimages in the first iteration only. In the last iteration, we instead invert all applied bijections to restore the original image layout. We discuss the shown optimized bijection (Lines 16 to 22) to invert multiple applications of $f_{2\downarrow}$ in section 3.3.

3.1 Signal Decomposition

The core of our method is an iterative subdivision of the input image into four independent subimages through the bijection $\sigma_{2\downarrow}$. We show the correctness of this transformation by modelling local data dependencies of each output pixel as affine functions. We then extend this notion from individual pixels to affine dependencies for the whole image.

Data Dependency of Affine Memory Accesses. As demonstrated in fig. 4 lines 5 to 10, stencil computations can usually be modeled through a loop nest centered around the output element. Given a single loop `for(i=o;i<w;i+=s)` with loop counter $i \in \mathbb{Z}$, initialization o, condition $i < w$, and loop increment s, all dynamic instances of this loop nest can be modeled through an affine function $is + o$ along with inequalities modeling the constraints imposed by the initialization and condition, which we conveniently express as $i \in [o, w)$. As can be seen in fig. 5(b), the initialization o partitions the domain $\mathbb{Z}$ into s disjunct subsets. As this describes a congruence relation modulo s, we denote $[o]_s$ as the subset of all values described by the affine function with initialization $o \in [0, s)$. On a bounded domain of width w, the size of the induced subsets may differ by one if the value range w is not divisible by the loop increment s. There is a prefix of $N = (w \bmod s)$ large congruence classes, and suffix of $r - N$ small congruence classes. Thus, the maximal wasted memory for padding to a single cardinality for all subsets is at most the number of small subsets $N < s$.

Data Dependencies of À-trous Wavelets. We will now consider how the global data dependencies of a dilated stencil loop centered on each pixel can be modeled analogously as an affine function. Given a stencil with radius r centered around pixel x with a dilation factor of s, we can uniquely identify each memory access of f with an iteration vector $(x, s\mathbf{n})$, where $n \in [-r, r]$ describes the

relative offset of a non-zero tap integrated into x. The memory locations accessed by x are given by N_x (black and green in fig. 5). The union of all center pixels $x \in \mathbb{Z}$ with a non-empty intersection of dependent memory accesses N_x can be described by an unbounded joint affine function (gray in fig. 5(a)). Consequently, a stencil with dilation s and arbitrary support $r > 0$, decomposes the domain $\mathbb{Z}$ into s independent subsets. Figure 5(b) shows an example for a one-dimensional stencil with dilation $s = 4$ as it occurs in level $l = 2$ of an à-trous wavelet. We only consider rectangular schedules, which apply the identical transformation to both orthogonal image signals along width W and height H of an image I, and can thus directly transfer observations for one-dimensional affine functions to the two-dimensional à-trous transform. We define $[\mathbf{o}]_l = \{x \mid x = 2^l i + \mathbf{o}, x \in I\}$ as the $\mathbf{o} = (o_x, o_y)$-th subimage at level l of the input image I. Thus, $I = [(0,0)]_0$. In a hierarchical application, each wavelet level l partitions the subimages further into even and odd signals along each axis. Therefore, level l consists out of $O \times O = (2^l) \times (2^l)$ subimages. Data dependencies between subimages $[\mathbf{o}]_l$ decompose recursively. There are no data dependencies between two taps at level l', if the taps were partitioned into different subimages in a previous iteration $l < l'$. Consequently, each subimage $[\mathbf{o}]_l$ is independently schedulable, only being directly dependent on its parent subimage $[\lfloor \mathbf{o}/2 \rfloor]_{l-1}$.

Delaying discussions of varying subimage sizes until section 3.2, we define the bijection

$$\sigma_{l\downarrow} = (o_x, o_y, c, x, y) \mapsto (o_x + l(x\%2^l), o_y + l(y\%2^l), c, \lfloor x/2^l \rfloor, \lfloor y/2^l \rfloor),$$

which takes $O \times O$ wavelet images as a 5-dimensional array $O \times O \times C \times W \times H$ and decomposes each subimage according to its data dependencies into 2^l subimages represented by an array of shape $2^l O \times 2^l O \times C \times \lceil W/2^l \rceil \times \lceil H/2^l \rceil$. In figure 3, we illustrate the application of $\sigma_{2\downarrow}$ where a single subimage $[(0,0)]_0$ is decomposed into four subimages. Mapping this array representation to a memory layout varying the innermost dimensions $W \times H$ most quickly, data dependencies to neighboring stencil taps $N_{\mathbf{x}}$ are localized.

3.2 Image Domain Embedding

We define the embedding $\sigma_r \circ \sigma_e$, which reshapes the output of $\sigma_{2\downarrow}$ back to the input shape $1 \times 1 \times C \times W \times H$. Such a mapping reduces memory consumption and maximizes device occupancy.

Motivation. In a baseline schedule, each center pixel of the iterative filter is mapped to a single GPU invocation. Consequently, the decomposition into congruence classes $[\mathbf{o}]_l$ induces a partition of invocations into independent sets. While this suggests that each subimage could be mapped to a single shader, embedding of all subimages into a single domain after each iteration has two benefits. First, we can reuse opaque image memory through ping-pong buffering. Second, we can dispatch a single shader that processes all subimages while being oblivious to the dimensions of subimages apart from optional weight modifications for out of bounds taps. Using a single shader dispatch for all subimages of an iteration trivially works around the quantization of subimage bounds to integer multiplies of the workgroup size. Otherwise, the occupancy of later iterations is impacted through inactive invocations and dispatch grid sizes that insufficiently saturate the hardware. Figure 1(d), shows an example where a single workgroup processes multiple subimages at once, which would otherwise reduce occupancy through inactive invocations if unembedded subimages are directly mapped to the device.

Embedding Bijection. In general, the shape of subimages $[\mathbf{o}]_l$ may vary by up to a pixel along each dimension, resulting in up to four subimage shapes. The function $g_l : \mathbb{N}^2 \mapsto \mathbb{N}^2$ maps subimage indices to a pixel offset in the input $W \times H$. By introducing padding to the width W and height H, subimages may be embedded on a uniform grid reducing computational cost of g_l. To ensure divisibility for all iterations, the padding introduced in the output image of the first iteration must

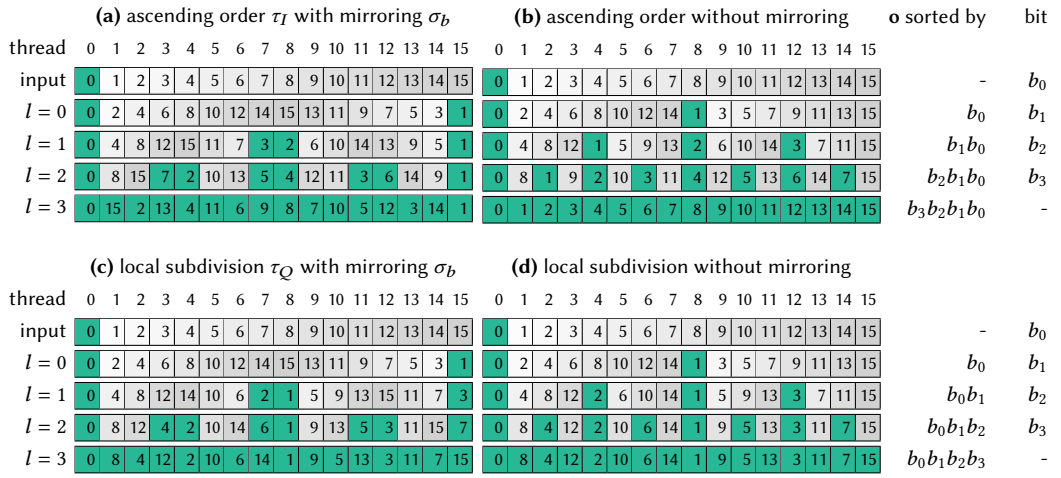

Fig. 6. Example of our method for both subimage orderings (Section 3.2) with and without the boundary bijection (Section 3.4) on an 16×1 image with the maximal number of wavelet iterations 4. The bits of an pixel index $b_n, ..., b_2, b_1, b_0$ are given with the most significant bit b_n (MSB) first. Mirroring ensures out of bound taps are close to the center tap, preventing visual light leaking. Representants $\mathbf{o}$ of each subimage are colored green. Shading indicates the orientation of the coordinate system of each subimage. Invocation indices indicate the mapping of the stencil centers to the pixel indices in each level—the invocation index reads the pixel in its column and writes it to the position as indicated by the order in the subsequent level below.

be divisible by the number of subimages in the input of the last iteration $2^{l_{\max}-1}$. This padding introduces at most $2^{l_{\max}-1} - 1$ pixels along each axis. Consequently, the bijection reshaping each iterations output is given as

$$\sigma_e = (o_x, o_y, c, x, y) \mapsto (0, 0, c, g_l(\mathbf{o}) + (x, y)).$$

Subimage Order. In the following, we discuss two possible definitions of the reordering bijection

$$\sigma_r = (o_x, o_y, c, x, y) \mapsto (\tau(o_x), \tau(o_y), c, x, y)$$

influencing the positioning of subimages $[\mathbf{o}]_l$ on the grid by applying a permutation τ before σ_e. First, local subdivision τ_Q, which subdivides each subimage individually, addressing subimages through a quad tree. Second, global subdivision, where we set τ_I to an identity mapping between subimage identifiers $\mathbf{o}$ and grid positions, which embeds subimages in ascending order.

Local Subdivision Order. Local subdivision naturally results from an iterative application of $\sigma_{2\downarrow}$ to each subimage. Each subimage $[\mathbf{o}]_l$ stored in a coordinate range in the input image, is partitioned by its least significant bit of the *local* pixel indices $g_l(\mathbf{o}) - \mathbf{x}$ and written back to the same coordinate range in the output image. As illustrated in fig. 6(d), this leads to a reversal of bits as the l-th bit is inserted as the lowest bit to form the l-bit wide subimage identifier. Consequently, local subdivision is defined as

$$\tau_Q(\mathbf{o} = b_l ... b_2 b_1 b_0) = b_0 b_1 b_2 ... b_l = \texttt{rev}_2(\mathbf{o}) >> (\texttt{bitsize}(\mathbf{o}) - l),$$

i.e. reversal of all l significant bits b_i of the subimage identifier $\mathbf{o}$. In a practical implementation, the function $\texttt{rev}_2$ reverses the bits of the $\texttt{bitsize}(\cdot) \geq l$-wide unsigned integer format used to store $\mathbf{o}$.

Ascending Order. An iterative application using an identity function $\tau_I(\mathbf{o}) = \mathbf{o}$ partitions *global* pixel indices. As shown in fig. 6(b), iteration l partitions the embedding at the l-th bit, inserting it as the highest bit into the subimage identifier, effectively partitioning the original pixel position at l with the lower bits representing o, and higher bits the local subimage position.

Discussion. Notably, an iterative global subdivision may be implemented as a single application of $\sigma_{2\downarrow}$ to the whole input image, i.e. the *embedded* subimages. Further, as the order between small and large subimages is maintained, Euclidian division automatically packs subimages on a rectilinear grid without padding. In an iterative application of local subdivision τ_Q, iteration l permanently partitions memory regions based on the l-th bit for all subsequent iterations $l' > l$. In contrast to τ_Q, an embedding in ascending order introduces additional data dependencies between subimages as potentially reused memory regions are not disjoint. (As an example, consider memory dependencies between subimage $[(1,0)]_1$ and $[(0,0)]_0$ in fig. 6.) However, in the proposed schedule, where subimages of a level are processed in a single shader dispatch with ping-pong buffering, these additional data dependencies are already satisfied.

3.3 Interleaving

To restore the original image layout after multiple iterations of our method, we define

$$\sigma_{l_{\max}\uparrow} = (\sigma_{\mathrm{o}}^{-1})^l = (\sigma_e \circ \sigma_b \circ \sigma_r)^{-1}(\sigma_{2\downarrow}^{-1}),$$

which first unembeds the subimages $(\sigma_e \circ \sigma_b \circ \sigma_r)^{-1}$, before (re)interleaving

$$(\sigma_{2\downarrow}^{-1})^{l_{\max}} = (\mathbf{o}, c, x, y) \mapsto (0, 0, c, \mathbf{o} + 2^l(x, y)).$$

The bijection σ_e^{-1} computes grid positions and subimage-local pixel indices from global pixel indices. The bijections σ_r^{-1} and σ_b^{-1} are self-inverse, with σ_r^{-1} restoring the subimage identifier from the embedded grid position and σ_b^{-1} normalizing the pixel coordinate system of each subimage.

3.4 Treatment of Signal Boundaries

In this section, we define the boundary permutation σ_b, which approximates texture mirroring for the embedded subimages in order to reduce overhead of out-of-bounds memory accesses.

Motivation. We strive for permutations to be pure post- and preprocesses not causing any additional arithmetic or control overhead within the filter loop. In the baseline schedule, the overhead of boundary checks can often be mitigated by exploiting texture mirroring or by setting out-of-bounds accesses to a special value with zero weight, i.e. $f_e(\cdot) = 0$—either using the texture sampler, or by resolving out-of-bounds accesses once while preloading data from global to local memory. However, our embedding $\sigma_r \circ \sigma_e$ prevents these optimizations as boundary checks between embedded subimages are required.

Boundary Bijection. We flip the axes of subimages embedded in the two dimensional domain in a checkerboard as shown fig. 1(a). Given the position $\mathbf{o}$ of the subimage on the embedding grid, the boundary bijection is flips the axis of every other subimage of shape $W_{[\mathbf{o}]_l} \times H_{[\mathbf{o}]_l}$, i.e.

$$\sigma_b = (o_x, o_y, c, x, y) \mapsto (o_x, o_y, c, o_x \ \&\ 1 \ ?\ x\ :\ W_o - x - 1, o_y \ \&\ 1 \ ?\ y\ :\ H_o - y - 1).$$

Discussion. As it allows cross-talk between neighboring subimages, σ_b is non-semantics preserving as the computed result $\sigma f(x)$ differs from $f(x)$ for pixels close to the image boundaries. However, as discussed in more detail in section 4.1, taps from neighboring subimages fall within the footprint of the filter support quantized to the dilation factor, effectively converting the à-trous filter to a undilated filter in boundary regions. Furthermore, it increases the number of taps integrated

from within the current subimage, as the boundary may be crossed multiple times during iterative filtering.

We can implement the boundary permutation as a pure postprocess in the postamble without additionally computing the coordinate system orientation as a side product of $(\sigma_e \circ \sigma_b)^{-1}$ in the preamble as our exposition assumes either a symmetric filter kernel or a filter kernel deriving its weights in a data-driven way from the pixel values. In these cases, the filter kernel is independent of the coordinate system orientation of subimages. Otherwise, either the filter kernel or the data would have to be flipped to match orientation.

Iterative Filtering. In an iterative application of a symmetric or data-driven filter kernel, the output of the prior iteration still would have to be unembedded to determine the coordinate system orientation of the output in the current iteration. We propose the following optimizations for iterative filtering. For ascending order τ_I, uneven pixels are always embedded at uneven positions since subimages are positioned using the LSB b_0 first. Thus, we can statically specialize a shader that flips half of the input image in the postamble of the first iteration, ignoring the orientation of subimages in all subsequent iterations. For local subdivision order τ_Q the boundary permutation is applied in each iteration. The orientation stays the same for pixels coordinates, where the most significant bit (MSB) of the prior subimage identifier b_{l-1} and the new MSB b_l are identical.

The difference between both image domain embedding schemes with mirroring is illustrated in fig. 6.

3.5 Shared Memory Bijection

A common optimization technique for stencils is shared memory resident overlapped tiling, i.e. for a workgroup size of $W_g \times H_g$, a rectangular region $C \times W_s \times H_s = C \times W_g + 2r \times H_g + 2r$ of the input image containing all memory locations accessed by a filter kernel with radius r is cached in fast on-chip memory (see fig. 1(d)). Efficient addressing of on-chip memory requires bank conflict resolution, which is commonly achieved through row-padding. As the memory accessed by a workgroup grows exponentially with each à-trous iteration, the baseline schedule is unable to leverage scarce on-chip memory, leading state-of-the-art implementations [Nvidia Inc. 2023] to only exploit on-chip memory in the first iteration. Our method converts wavelets to an undilated filter with constant filter support across all iterations. However, even for the first iteration, mapping an overlapped tile to shared memory can be non trivial, if many features C are cached with row-padding. Therefore, in this section, we propose a bijection σ_{shm} to resolve bank conflicts without padding. In essence, we derive the shape of a sliding window, which is bank conflict-free for any memory offset within the shared memory resident tile (Figure 1(d)). We then use this shape to tile the subgroups within a workgroup.

The GCD-Test. The core idea of our method is to design a permutation σ_{shm} that avoids concurrent access to the same bank through the GCD-Test [Muchnick 1997], an alias analysis technique. To do so, we model the affine constraint imposed by bank conflicts explicitly by comparing the stride s of an affine memory access with the stride implied by the number of shared memory banks c_b in the system. Assuming successive words are stored in successive banks in a cyclic pattern, this implies

$$n \cdot \hat{c}_b = m \cdot s \iff n \cdot \hat{c}_b - m \cdot s = 0, \text{ where } n, m \in \mathbb{N} \tag{1}$$

and $\hat{c}_b$ is the number of shared memory banks after correcting for the size of the array elements stored in shared memory. We assume shared memory banks and array elements to be of equal size ($c_b = \hat{c}_b$) and refer to the supplementary material for differing bit widths. Solving this linear Diophantine equation, for example using the extended Euclidean algorithm, gives us the greatest common divisor $d = \gcd(s, -\hat{c}_b) = \gcd(s, \hat{c}_b)$ and the coefficients describing the smallest positive

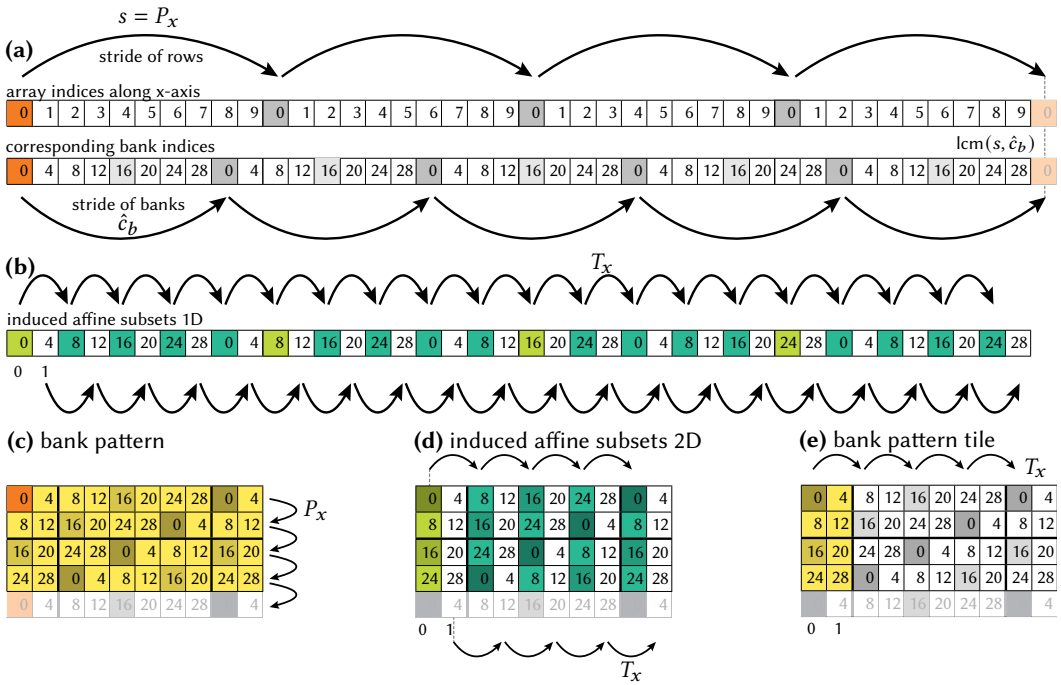

Fig. 7. Overview of our padding-free bank-conflict resolution strategy. The example uses a shared memory region with width $W_s = 10$ and a component size of 128-bit in a system with 32 32-bit shared memory banks. The same graphic can also serve as the solution for a system with eight banks and a component size of 32-bit if each shown bank index is divided by four. **(a)** The core idea is to compare the stride of an affine column access with the stride of the bank to find the point at which both affine functions conflict and thus realign to repeat the mapping. **(b)** This induces an affine function partitioning banks into disjunct subsets. **(c)** Shows the affine access and shared memory of (a) in two-dimensional row-major order. **(d)** Shows (b) in two dimensions. **(e)** A mapping of invocation indices within a subgroup to any set of columns from different affine subsets resolves all bank conflicts. We call the resulting region a bank pattern tile if adjacent columns are picked to form a subgroup.

non-trivial solution

$$n = \frac{\hat{c}_b}{d} \quad \text{and} \quad -m = \frac{s}{d}.$$

Figure 7(a) provides a visual intuition of the above equations with $\hat{c}_b = 8$ and $s = 10$. The coefficient $m = 10/d = 5$ gives the number of cycles through each bank required until the array accesses and the shared memory banks realign, and thus the point of the first bank conflict. Since the number of banks coincides with the subgroup size, this is equal to the number of subgroups. The coefficient $n = 8/d = 4$ gives the number of array cells accessed before the bank conflict occurs.

Construction. Setting the stride s of the affine memory access to the width W_s of the shared memory region allows an analysis of bank conflicts within a column of a row-major shared memory layout. In this case, n gives the number of rows after which the periodic mapping between array indices and bank indices repeats. As a consequence, resolving bank conflicts in a region $P_x \times P_y = W_s \times n$ resolves bank conflicts for any shared memory footprint $W_s \times H_s$ – effectively bounding the height of our analysis. We call this region, shown in fig. 7(c), a *bank pattern*. At the same time, n gives the maximal height of a column $1 \times n$ that is bank conflict free. Analog to our discussion section 3.1, the greatest common divisor d induces a partition of these columns into

disjunct affine subsets of shared memory banks, which is shown in fig. 7(b). Consequently, reducing the problem from a two-dimensional conflict resolution problem to a one-dimensional problem over columns. Selecting an arbitrary column from each subset forms a bank conflict free access pattern for a subgroup. If adjacent columns are picked to form a subgroup of shape $T_x \times T_y = d \times n$ as shown in fig. 7(d), we call the region a *bank pattern tile*.

Properties. Bank pattern tiles are translation invariant and periodic at column boundaries as d divides W_s by construction. Rows are periodic if n divides H_s. We consider translation by one to construct a proof by induction. Shifting a bank pattern tile vertically by one, a row j will be replaced by a row $(j \pm n)\%H_s$ with an identical sequence of bank indices from a neighboring bank pattern. Shifting a bank pattern tile horizontally by one, a column j will be replaced by another column $(j \pm d)\%W_s$ from the same affine subset, which is backed by the same set of banks—albeit in a different sequence. We discuss normalization of banks within columns in the supplementary.

Limitations. As the bank pattern height may not divide the height of the workgroup H_g, reshaped subgroups within a workgroup may be unable to cover the workgroup shape. Any odd shared memory width W_s, for example, yields an elongated bank pattern $1 \times \hat{c}_b$. While for any stencil radius r, odd shared memory widths $W_s = W_g + 2r$ may not occur for usual choices of W_g, we discuss strategies to alleviate these limitations by reshaping bank pattern tiles in the supplementary.

Application. To resolve bank conflicts in a 2-dimensional workgroup, we define the bijection

$$\sigma_{\text{shm}} = (0, 0, c, x, y) \mapsto (0, 0, c, \begin{bmatrix} x_{\text{g}} * W_g \\ y_{\text{g}} * H_g \end{bmatrix} + \begin{bmatrix} d(i_{\text{sg}} \% (W_g / d)) \\ n(i_{\text{sg}} / (W_g / d)) \end{bmatrix} + \begin{bmatrix} i_{\text{inv}} \% d \\ i_{\text{inv}} / d \end{bmatrix}),$$

which tiles the workgroup shape with subgroups (second summand) reshaped to bank pattern tiles (third summand). The transform is applied to 2-dimensional local invocation indices maintaining the global workgroup offset (first summand). The workgroup identifier $x_{\text{g}}, y_{\text{g}}$, the index of the subgroup within the workgroup i_{sg}, and the index of the invocation within the subgroup i_{inv} are usually available directly without deriving them from x, y. An example application is shown in fig. 1(d), where 4×2 subgroups of shape 4×8 cover a 16×16 workgroup. If H_g is equal to n, σ_{shm} may alternatively be implemented as a transpose, as shown in fig. 4 for a 8×8 workgroup.

4 EVALUATION

We evaluate the effects of our scheduling method on reconstruction quality and execution efficiency. In section 4.1 we discuss the effect of the non-semantics preserving treatment of image boundaries on reconstruction quality. In section 4.2 we evaluate the throughput of key GPU subsystems and its effect on execution time.

4.1 Reconstruction Quality At Image Boundaries

The weighted contribution of neighboring pixels to a pixel in the boundary region for different boundary strategies is shown in fig. 8. If out-of-bounds memory accesses are set to zero weight, our scheduling method preserves the semantics of the baseline schedule (first column). If out-of-bounds memory accesses are not prevented, a naive embedding σ_e without checkerboarding of the coordinate axis σ_b will retrieve filter taps from the opposite side of neighboring subimages (second column). This crosstalk between wavelets visually manifests as light leaking to the opposing side of the input image. In contrast, the proposed approximation of mirroring σ_b (third column) automatically retrieves close-by filter taps from neighboring subimages, effectively converting the à-trous filter to a dense filter in boundary regions. More precisely, let D be a tiling of the input image I into $2^l \times 2^l$ tiles, which contain one pixel from each subimage—except for partial tiles at

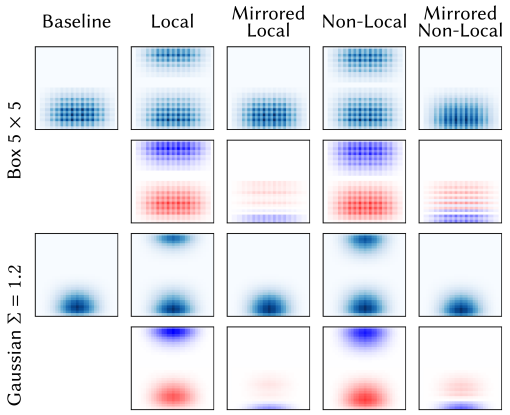

Fig. 8. Effect of different boundary strategies in the two-dimensional embedding. The image size is 32×32. Even rows show the weight contribution of the neighboring pixels to the measured center pixel $x = (16, 30)$. Difference images in odd rows encode deviations from the baseline.

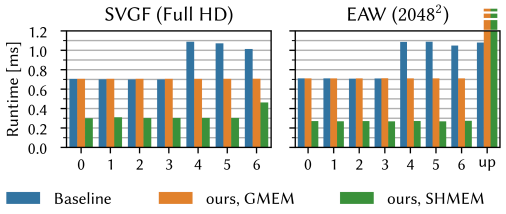

Fig. 9. Runtime of the baseline schedule and our proposed schedules.

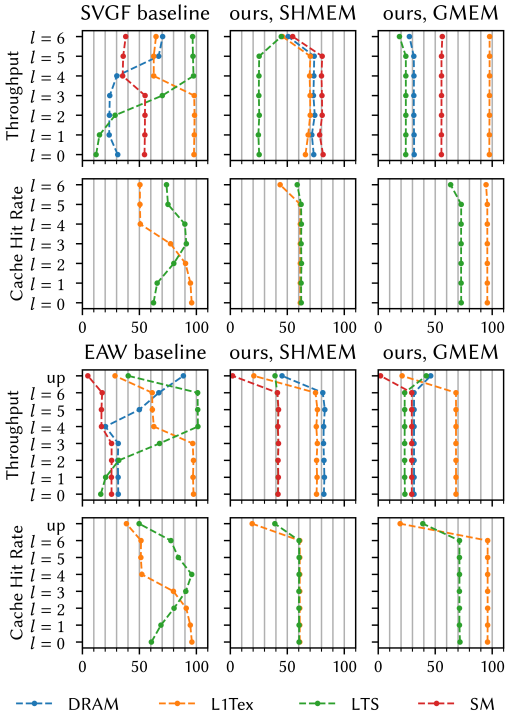

Fig. 10. Cache hit rates and throughputs of hardware units for the baseline schedule and our proposed schedule with shared memory ('SHMEM') and without shared memory ('GMEM'). Throughputs are measured in percent relative to the maximum theoretical throughput.

boundaries if I is not divisible by 2^l. Then all out of bounds memory taps fall into zero weight taps within the support of the à-trous kernel in I expanded to the tiles in D intersected by the support. This invariant holds for any positioning of wavelets τ on a rectilinear of the embedding permutation $\sigma_r \circ \sigma_e$. This remapping of out-of-bounds memory accesses has two side effects. First, structural artifacts of the wavelet filter, especially prominent in the box filter (first and second row), are reduced along the axis orthogonal to the boundary, whereas the non-locality of the ascending embedding τ_I benefits this smoothing (last column). Second, in the baseline, pixels close to the boundary have low contribution as the iterative filter is truncated at the first out-of-bounds memory access. Consequently, taps within bounds are only integrated in early iterations where all ancestor taps are within bounds. As σ_e does not truncate the filter if a tap is out-of-bounds, filter taps may cross the boundary multiple times, preserving the correct weight for contributions from pixels within bounds. As a result, weight of pixels close to the boundary is increased for both τ_I and τ_Q.

4.2 Performance Evaluation

Hardware & Software. We evaluate our method on an NVIDIA RTX 3070. Measurements are taken in stable power mode. Given the regularity of the workload, variance between measurements is minor and consequently not reported. We evaluate our method using two à-trous algorithms.

First, edge avoiding wavelets (EAW) [Hanika et al. 2011], which are designed for computational photography and denoise a color image without auxiliary buffers. Second, spatio-temporal variance guided filtering (SVGF) [Schied et al. 2017], which is a real-time denoising algorithm designed for removal of Monte Carlo noise. Our baseline implementation [Alber 2024] has slightly adapted edge stopping functions with depth gradients explicitly stored in the G-buffer and an elided Gaussian blur of luminance variance. We refer to the code provided in the supplementary material for details.

Performance Characteristics. Figure 10 compares the throughput of hardware units and their cache hit rates in the baseline schedule to our proposed schedule. Figure 9 plots shader runtimes. We note that the runtime of the denoiser is independent of the framebuffer contents.
Independent of the iteration, the workload is memory-bounded. In the baseline schedule, the active working set of each workgroup increases with each iteration l. Consequently, we observe the workload gradually shifting to increasingly distant units of the memory hierarchy. The workload is initially L1Tex limited. L2 bandwidth from L1 and L2 bandwidth to VRAM increase with each iteration and is saturated for $l \geq 4$, at which point the workload is limited by L2. The increasing non-locality of the workload is also observable in the cache hit rates. L1 hit rates gradually decrease from 96% to 31%. Data is increasingly served from L2 with hit rates increasing to 96% in iteration $l = 4$. As a result of poor memory subsystem performance, multiprocessor throughput is low, further decreasing as latency increases. The most prominent stall reasons are long scoreboard and TEX throttle—indicating that a high number of texture fetches with high latency are inflight. If the decreasing performance of the memory subsystem manifests in runtime variation depends on the specific combination of à-trous filter and device. We especially observe increasing runtimes for iterations saturating L2 bandwidth for both evaluated algorithms on an NVIDIA RTX 3070. However, in other proprietary implementations, not further evaluated in this paper, we either observe a near-linear increase in runtime with each iteration; or a constant runtime as decompression of the G-buffer on the special function unit masks poor memory subsystem performance [Willberger et al. 2019].
In contrast, our method has near-constant throughput and cache hit rates for each filter iteration, resulting in an almost uniform shader runtime for each iteration. Thus, indicating the dynamically varying output pattern of our localization is not integral for performance. Comparing an implementation of our method without shared memory to the baseline, throughputs of the memory subsystem are near-identical to the first iteration of the baseline. SM throughput, however, is increased in our variants—27.2% for ascending order, 29.6% for local subdivision order, compared to 25.6%-17.4% in the baseline schedule. While our permuted filter iterations issue 15.8%, resp. 8.1%, more instructions for subdivision, resp. ascending order, their runtime is near-identical to the baseline schedule, indicating overhead of our method is hidden by instruction-level parallelism. Memory localization without shared memory usage does not result in a speedup for the first iteration, only reducing runtime for later iterations with an approximate speedup of 1.5. Preloading to shared memory results in a speedup of 2.5 for the first iteration. The speedup over later iterations of the baseline is 3.8. We note that our method with shared memory enabled has lower L1Tex hit rates, as memory transactions after data preloading are served from shared memory.
The last SVGF iteration restores the original image layout. The non-locality of this write operation negatively impacts runtimes, increasing the runtime from 0.30ms to 0.46ms. Given the low multiplicity of write to read operations, we do however still observe a speedup of 2.19 over the baseline. The last iteration of EAW merges the result of all iterations. Our method delocalizes read operations in this upward pass negatively impacting runtimes—increasing the runtime of our shared memory method to 3.88ms compared to 1.05ms in the baseline. As a result, the overall speedup for the whole algorithm is reduced to 1.2.

Staged Decompression. Auxiliary features in the G-buffer are usually packed and encoded in a rendering engine specific format. Decoding this data has high arithmetic cost within the stencil loop—especially if data is non-linearly compressed. For SVGF, we decompress data once during preloading to shared memory, utilizing 40 byte per pixel to store the data of each neighbor. This accounts for 0.16ms of the reported runtime reduction, equating to an additional speedup of 1.24.

Portability. We report runtime speedups for other devices to demonstrate hardware portability. For SVGF, on a NVIDIA RTX 2070, the baseline has a runtime of 0.88 to 1.62ms. In contrast, our method has a constant runtime of 0.66ms—equivalent to a speedup of 1.33 to 2.42. For EAW, on an AMD Radeon RX 7900 XT with variable clock rates, we measure a runtime from 0.09 to 0.16ms for the baseline, 0.11 to 0.12ms for our method without shared memory and local subdivision, 0.068 to 0.071ms with shared memory. For the ascending variant, we measure 0.10 to 0.11ms without shared memory, 0.052 to 0.59ms with shared memory—equivalent to a speedup of 1.72 to 2.85. Delocalization in the upward pass results in a runtime of 0.575ms compared to 0.464ms in the baseline.

5 CONCLUSION

We introduced a scheduling technique to accelerate à-trous wavelet-based denoisers. Our optimization approach follows the general idea that on modern hardware with deep memory hierarchies, the arithmetic overhead of memory index computations can either be hidden by memory latency through instruction-level parallelism or amortized by the increased memory throughput. We derived a permutation of GPU invocation indices localizing image coordinates to coalesce global memory accesses and to enable shared memory usage. The permutation is loop-invariant avoiding arithmetic and control overhead within the stencil loop. We optionally eliminate boundary checks and improve neighborhood sampling at image boundaries without any overhead through an embedding of shifted decimated image signals into the original two-dimensional image domain. Since our method enables the usage of shared memory, a promising future research direction is to decouple shader dispatches from wavelet iterations. Given the exponentially growing filter support of à-trous wavelets, non-overlapped tiling [Bondhugula et al. 2016; Grosser et al. 2014, 2013] seems especially promising. Our method improves the memory efficiency of undecimated à-trous wavelets independent of the hyperparameters of the denoiser, such as the number of wavelet levels and auxiliary features. Furthermore, our schedule has constant cost in each iteration, which we believe will help others to design efficient denoisers tailored to the needs of their rendering architecture.

ACKNOWLEDGMENTS

This work has been supported by the Helmholtz Association (HGF) under the joint research school “HIDSS4Health – Helmholtz Information and Data Science School for Health” and through the Pilot Program Core Informatics.

REFERENCES

Lucas Alber. 2024. Markov Chain Path Guiding for Real-Time Global Illumination and Single-Scattering. (2024). Institute for Visualization and Data Analysis (IVD), Karlsruhe Institute of Technology (Master Thesis).

Steve Bako, Thijs Vogels, Brian Mcwilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony Derose, and Fabrice Rousselle. 2017. Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 36, 4, Article 97 (2017), 14 pages. https://doi.org/10.1145/3072959.3073708

Colin Barré-Brisebois, Henrik Halén, Graham Wihlidal, Andrew Lauritzen, Jasper Bekkers, Tomasz Stachowiak, and Johan Andersson. 2019. Hybrid Rendering for Real-Time Ray Tracing. (2019), 437–473. https://doi.org/10.1007/978-1-4842-4427-2_25

Jakub Boksansky, Michael Wimmer, and Jiri Bittner. 2019. Ray Traced Shadows: Maintaining Real-Time Frame Rates. (2019), 159–182. https://doi.org/10.1007/978-1-4842-4427-2_13

Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. 2016. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (2016), 1285–1298. https://doi.org/10.1109/TPDS.2016.2615094

Chakravarty R Alla Chaitanya, Anton S Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics* 36, 4 (2017), 1–12. https://doi.org/10.1145/3072959.3073601

Holger Dammertz, Daniel Sewtz, Johannes Hanika, and Hendrik PA Lensch. 2010. Edge-avoiding a-trous wavelet transform for fast global illumination filtering. In *Proceedings of High Performance Graphics*. 67–75. https://doi.org/10.2312/EGGH/HPG10/067-075

Raanan Fattal. 2009. Edge-avoiding wavelets and their applications. *ACM Transactions on Graphics* 28, 3 (2009), 1–10. https://doi.org/10.1145/1531326.1531328

Tobias Grosser, Albert Cohen, Justin Holewinski, Ponuswamy Sadayappan, and Sven Verdoolaege. 2014. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 66–75. https://doi.org/10.1145/2581122.2544160

Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, Ponuswamy Sadayappan, and Sven Verdoolaege. 2013. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. 24–31. https://doi.org/10.1145/2458523.2458526

Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*.

Johannes Hanika, Holger Dammertz, and Hendrik Lensch. 2011. Edge-Optimized À-Trous Wavelets for Local Contrast Enhancement with Robust Denoising. *Computer Graphics Forum (Proceedings of Pacific Graphics)* 30, 7 (2011), 1879–1886. https://doi.org/10.1111/j.1467-8659.2011.02054.x

Jon Hasselgren, Jacob Munkberg, Marco Salvi, Anjul Patney, and Aaron Lefohn. 2020. Neural Temporal Adaptive Sampling and Denoising. *Computer Graphics Forum* 39, 2 (2020), 147–155. https://doi.org/10.1111/cgf.13919

Nikolai Hofmann, Jon Hasselgren, and Jacob Munkberg. 2023. Joint Neural Denoising of Surfaces and Volumes. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 6, 1 (2023), 1–16. https://doi.org/10/ggd8dh

Nima Khademi Kalantari, Steve Bako, and Pradeep Sen. 2015. A Machine Learning Approach for Filtering Monte Carlo Noise. *ACM Trans. Graph.* 34, 4, Article 122 (jul 2015), 12 pages. https://doi.org/10.1145/2766977

Masaki Kawase. 2003. Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L (Wreckless). (2003). http://www.daionet.gr.jp/~masa/archives/GDC2003_DSTEAL.ppt Game Developers Conference (GDC).

Patrick Kelly, Yuriy O'Donnell, Kenzo ter Elst, Juan Cañada, and Evan Hart. 2021. Ray Tracing in Fortnite. In *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Apress, 791–821. https://doi.org/10.1007/978-1-4842-7185-8_48

Steven S. Muchnick. 1997. *Advanced compiler design implementation*. Morgan Kaufmann. 279–284 pages.

Nvidia Inc. 2023. NVIDIA Real-Time Denoisers. https://github.com/NVIDIAGameWorks/RayTracingDenoiser. retrieved 01/09/2023.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International conference on medical image computing and computer-assisted intervention (MICAII)*. Springer, 234–241. https://doi.org/10.1007/978-3-319-24574-4_28

Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. 2017. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In *Proceedings of High Performance Graphics*. Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. https://doi.org/10.1145/3105762.3105770

Christoph Schied, Christoph Peters, and Carsten Dachsbacher. 2018. Gradient Estimation for Real-time Adaptive Temporal Filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 2, Article 24 (2018), 16 pages. https://doi.org/10.1145/3233301

Pradeep Sen, Matthias Zwicker, Fabrice Rousselle, Sung-Eui Yoon, and Nima Khademi Kalantari. 2015. Denoising your Monte Carlo renders: recent advances in image-space adaptive sampling and reconstruction. In *ACM SIGGRAPH Courses*. 255. https://doi.org/10.1145/2776880.2792740

Manu Mathew Thomas, Gabor Liktor, Christoph Peters, Sungye Kim, Karthik Vaidyanathan, and Angus G Forbes. 2022. Temporally Stable Real-Time Joint Neural Denoising and Supersampling. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 5, 3 (2022), 1–22. https://doi.org/10.1145/3543870

Manu Mathew Thomas, Karthik Vaidyanathan, Gabor Liktor, and Angus G. Forbes. 2020. A Reduced-Precision Network for Image Reconstruction. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 39, 6, Article 231 (2020), 12 pages. https://doi.org/10.1145/3414685.3417786

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. (2018). http://arxiv.org/abs/1802.04730

Thijs Vogels, Fabrice Rousselle, Brian Mcwilliams, Gerhard Röthlin, Alex Harvill, David Adler, Mark Meyer, and Jan Novák. 2018. Denoising with Kernel Prediction and Asymmetric Loss Functions. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 37, 4, Article 124 (2018), 15 pages. https://doi.org/10.1145/3197517.3201388

Thomas Willberger, Clemens Musterle, and Stephan Bergmann. 2019. Deferred Hybrid Path Tracing. In *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress, 475–492. https://doi.org/10.1007/978-1-4842-4427-2_26

Bing Xu, Junfei Zhang, Rui Wang, Kun Xu, Yong-Liang Yang, Chuan Li, and Rui Tang. 2019. Adversarial Monte Carlo Denoising with Conditioned Auxiliary Feature Modulation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 38, 6, Article 224 (2019), 12 pages. https://doi.org/10.1145/3355089.3356547

Dmitry Zhdan. 2021. ReBLUR: A Hierarchical Recurrent Denoiser. In *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Apress, 823–844. https://doi.org/10.1007/978-1-4842-7185-8_49

Matthias Zwicker, Wojciech Jarosz, Jaakko Lehtinen, Bochang Moon, Ravi Ramamoorthi, Fabrice Rousselle, Pradeep Sen, Cyril Soler, and S-E Yoon. 2015. Recent advances in adaptive sampling and reconstruction for Monte Carlo rendering. In *Computer Graphics Forum*, Vol. 34. 667–681. https://doi.org/10.1111/cgf.12592

Received 5 January 2024; revised 29 March 2024; accepted 6 April 2024