Compiling High Performance Recursive Filters
Supplementary Material

Gaurav Chaurasia[1] Jonathan Ragan-Kelley[2] Sylvain Paris[3] George Drettakis[4] Fredo Durand[1]

[1]MIT CSAIL [2]Stanford University [3]Adobe [4]Inria

1 Our syntax and equivalent pseudo code

Our compiler and language are built on top of Halide. Here, we describe our representation and generated code, by example, in terms of the resulting Halide constructs and the logical loops they generate.

We define IIR filters as a series of linear filters of some coefficients, each operating in a specific dimension and direction:

```
RecFilter F("filter");
RecFilterDim x("x", image_width);   // dimensions
RecFilterDim y("y", image_height);  // dimensions
F(x,y) = In(x,y);                   // init
F.add_filter(+x, {a0, a1, a2});     // 1st filter
F.add_filter(-y, {a0, a1, a2});     // 2nd filter
```

Listing 1: *Our syntax*

Internally, we maintain a Halide **Func** that computes the filters. The initialization simply creates a **Func** that uses the same expression to initialize all pixels. This also creates the causal and anticausal scan variables **rx, ex, ry, ey** for each dimension using the image size as their extents. Each new filter adds a new update definition. In the following example, the first filter adds a causal filter in the **x** dimension. We use the corresponding scan variable **rx** to create the update definition that updates the current pixel using previous pixels in the **x** dimension using the corresponding filter coefficients.

```
Func R(F.name());
RDom rx(0, image_width);      // causal scan var x
RDom ry(0, image_height);     // causal scan var y
Expr ex = image_width-1-ry;   // anticausal scan var x
Expr ey = image_height-1-ry; // anticausal scan var y

R(x,y)  = In(x,y);            // init

// update defs 1 and 2
R(rx,y) = a0*R(rx,y) + a1*R(rx-1,y) + a2*R(rx-2,y);
R(x,ey) = a0*R(x,ey) + a1*R(x,ey-1) + a2*R(x,ey-2);
```

Listing 2: *Internal Halide representation*

The corresponding loops are sketched as follows:

2 Tiling transformations

Given a pipeline of recursive filters specified as above, our goal is to mechanically transform it to enable high-performance parallel tiled computation. To this end, programmers rely on a **split** operator and specify tile size for each dimension to be split.

```
F.split(x, tile_width, y, tile_height ...);
```

This simple command results, under the hood, in complex code transformations that generate new passes corresponding to inter- and intra-tile computation. In this section, we describe the mathematical transformations corresponding to a split for different scenarios depending on whether recursive filters share the same dimensions or not and if they are causal or anticausal. In the next section, we discuss how the resulting passes can be organized for increased efficiency.

Algorithm 1 Pseudo-code for *filter*

for all y **do** // initialize
  for all x **do**
    $f(x,y) \leftarrow In(x,y)$
  end for
end for
for all y **do** // 1st filter or update def
  for $rx = 0$ to $image\_width - 1$ **do**
    $f(rx,y) \leftarrow a_0 f(rx,y) + a_1 f(rx-1,y) + a_2 f(rx-2,y)$
  end for
end for
for $ry = 0$ to $image\_height - 1$ **do** // 2nd filter or update def
  for all x **do**
    $\bar{ry} \leftarrow image\_height - 1 - ry$
    $f(x,\bar{ry}) \leftarrow a_0 f(x,ry) + a_1 f(x,\bar{ry}+1) + a_2 f(x,\bar{ry}+2)$
  end for
end for

To gain intuition, we first discuss the case of a single 1D filter before looking at multiple filters, either on the same axis or along orthogonal axes. We also begin with only causal filters, i.e., filters that process data in order of increasing indices. We explain at the end how to handle anti-causal filters that process data in the reverse order.

2.1 Background: single 1D filter

We start by decomposing a 1D recursive filter f of order k that traverses the data in causal order, i.e., in the order of increasing indices. The derivation that follows is known, for instance, since it can be seen as a special case of Nehab's derivation in 2D [2011]. We include it for completeness since it is a central building block of our approach. In our paper, we use the definition most commonly used for image processing where the filter is first initialized with the input image I and then updated using a recursive formulation. In this context, the definition of a filter is:

$$\text{initialization:} \quad f(x) \leftarrow I(x) \tag{1a}$$

$$\text{recursion:} \quad f(x) \leftarrow \sum_{j=0}^{k} a_j f(x-j) \tag{1b}$$

where $\{a_i\}$ are the coefficients defining the filter.

Then, we decompose the image domain into tiles of size $t > k$ and express the pixel position in this coordinate system. We name x_o the tile index and $x_\text{i} \in [0; t-1]$ the pixel index within the tile, that is, $x = tx_\text{o} + x_\text{i}$. We use these variables to split the sum in Equation (1b) into an intra-tile term including only the pixels within the tile x_o and an inter-tile term with pixels outside it. For clarity, we introduce $f_{x_\text{o}}(x_\text{i}) = f(x_\text{o}t + x_\text{i})$ and $m_\text{i} = \min(x_\text{i}, k)$ to get:

$$f_{x_\text{o}}(x_\text{i}) \leftarrow \sum_{j=0}^{m_\text{i}} a_j f_{x_\text{o}}(x_\text{i}-j) + \sum_{j=m_\text{i}+1}^{k} a_j f_{x_\text{o}}(x_\text{i}-j) \tag{2}$$

When the right term is not empty, it accesses data in the previous

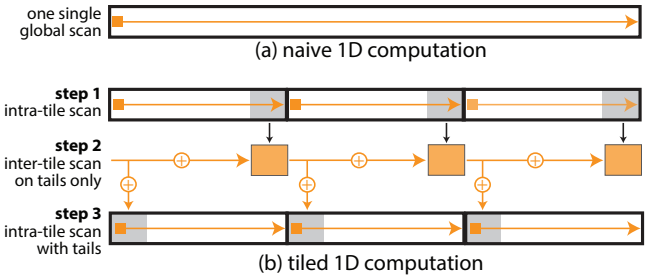

Figure 1: *Single 1D filter. (a) Naive sequential computation. (b) We transform it into a tiled scheme to improve performance. Step 1: we perform a scan on each tile in parallel. Step 2: we update the tails (the last elements of each tile) in order. Because of the order constraint, we cannot parallelize it; this is a fast operation nonetheless because of the small number of elements touched. Step 3: we update each tile in parallel to get the final result.*

tile, which we emphasize by changing the indices:

$$f_{x_\mathrm{o}}(x_\mathrm{i}) \leftarrow \underbrace{\sum_{j=0}^{m_\mathrm{i}} a_j f_{x_\mathrm{o}}(x_\mathrm{i}-j)}_{\text{intra-tile computation}} + \underbrace{\sum_{j=1}^{k-m_\mathrm{i}} a_j f_{x_\mathrm{o}-1}(t-j)}_{\text{correction term}} \quad (3)$$

This expression is the first step toward the tiled computation scheme. The left term depends only on data within a tile and thus, can be computed independently for each tile in parallel if we ignore the right term. It is a recursive filter f^{intra} with same coefficients as the original filter but computed at the tile level:

$$\text{initialization: } f^{\text{intra}}_{x_\mathrm{o}}(x_\mathrm{i}) \leftarrow \begin{cases} I(x_\mathrm{o}t + x_\mathrm{i}) & \text{if } x_\mathrm{i} \in [0; t-1] \\ 0 & \text{otherwise} \end{cases} \quad (4a)$$

$$\text{recursion: } f^{\text{intra}}_{x_\mathrm{o}}(x_\mathrm{i}) \leftarrow \sum_{j=0}^{k} a_j f^{\text{intra}}_{x_\mathrm{o}}(x_\mathrm{i}-j) \quad (4b)$$

These intra-tile values $f^{\text{intra}}_{x_\mathrm{o}}(x_\mathrm{i})$ can be "corrected" into the filter values $f_{x_\mathrm{o}}(x_\mathrm{i})$ by adding to them a linear combination of the last k elements of the previous tile $x_\mathrm{o}-1$. We call these elements the *tail* of that tile. We provide the detail of the derivation in Sec. 3. It is based on an induction on x_i comparing Equations 3 and 4b. Further, the coefficients of the linear combination are functions of the $\{a_j\}$ coefficients only, that is, they can be precomputed. We precompute a $t \times k$ *filter matrix* $\mathbf{P}$ that allows us to compute the vector $\mathbf{f}_{x_\mathrm{o}} = \big(f_{x_\mathrm{o}}(0), \ldots, f_{x_\mathrm{o}}(t-1)\big)$ from $\mathbf{f}_{x_\mathrm{o}-1}$ and the last k elements of $\mathbf{f}^{\text{intra}}_{x_\mathrm{o}} = \big(f^{\text{intra}}_{x_\mathrm{o}}(0), \ldots, f^{\text{intra}}_{x_\mathrm{o}}(t-1)\big)$:

$$\mathbf{f}_{x_\mathrm{o}} \leftarrow \mathbf{f}^{\text{intra}}_{x_\mathrm{o}} + \mathbf{P}\mathcal{T}_k(\mathbf{f}_{x_\mathrm{o}-1}) \quad (5)$$

where $\mathcal{T}_k$ is the operator that selects the last k elements or rows of its arguments. The filter matrix $\mathbf{P}$ computes the effect of the filter of any k elements on subsequent t elements. The details of the algorithm to construct $\mathbf{P}$ are given in Sec. 3.

One could use Equation 5 naively by first computing all the $\{f^{\text{intra}}_{x_\mathrm{o}}\}$ and then updating them by going through them in order and applying Equation 5. This scheme is inefficient because the last step goes through all the values in a constrained order, which prevents parallelism. Instead, a tile-friendly procedure is as follows (Fig. 1).

1. Intra-tile computation: We compute the intra-tile values $f^{\text{intra}}_{x_\mathrm{o}}(x_\mathrm{i})$ using Equation 4 for each tile. This can be done in parallel at the tile level (Fig. 1, step 1).

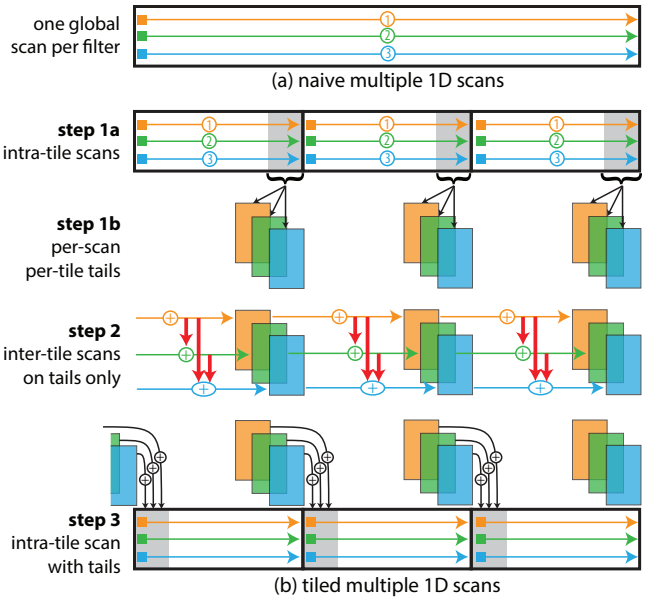

Figure 2: *Mutliple 1D filters. (a) Naive sequential computation. One can apply the single-filter scheme (Fig. 1b) once per filter but this does not fully exploit the locality of operations. (b) We treat all filters at once to minimize alternation between intra- and inter-tile steps. Step 1a: we apply all filters within each tile in parallel. Step 1b: we save a copy of the tails after each filter. Step 2: we sequentially update the tails using the tails of all the preceding filters at the previous tile. Step 3: we update each tile in parallel using all the tails at the previous tile to get the final result (step 3).*

2. Inter-tile tail computation: We use Equation 5 only to update the tail of each tile, i.e.: $\mathcal{T}_k(\mathbf{f}_{x_\mathrm{o}}) \leftarrow \mathcal{T}_k(\mathbf{f}^{\text{intra}}_{x_\mathrm{o}}) + \mathcal{T}_k(\mathbf{P})\mathcal{T}_k(\mathbf{f}_{x_\mathrm{o}-1})$. The order is constrained but the operation is fast because it concerns only k pixels per tile (Fig. 1, step 2).

3. Intra-tile update: Now that the tails are computed, Equation 5 can be applied to each tile in parallel (Fig. 1, step 3).

This procedure is illustrated in Figure 1. It transforms the original filter (Eq. 1) into 3 sets of subfunctions:
– the intra-tile filters $\{f^{\text{intra}}_{x_\mathrm{o}}\}$,
– the tails $\{\mathcal{T}_k(\mathbf{f}_{x_\mathrm{o}})\}$ recursively computed over the entire image,
– and the final intra-tile values $\{f_{x_\mathrm{o}}\}$.

We now build on more complex cases using the above transformation as a template.

2.2 Multiple 1D filters along the same dimension

We now consider the case where several filters are applied along the same axis. A simple approach can be to apply the method described in the previous section to each filter. The drawback of this approach is that it alternates between intra-tile and inter-tile operations, which degrades the locality of the computation and slows down data access. In this section, we introduce a transformation that handles all the filters at once and avoids this alternation. The derivation proceeds similarly to the single-filter case. We provide the details in Sec. 3 and outline the main steps and results below.

We consider n recursive filters $f_1, \ldots, f_n$, applied one after each other on the same 1D data, that is, f_2 is initialized using the result of f_1 and so on. As previously, we define their intra-tile counterparts, $\{f^{\text{intra}}_{1,x_\mathrm{o}}\}, \ldots, \{f^{\text{intra}}_{n,x_\mathrm{o}}\}$. The main difference with the single-filter case is that updating these functions into the final results also uses the tails of the previously applied filters in addition to its own. That

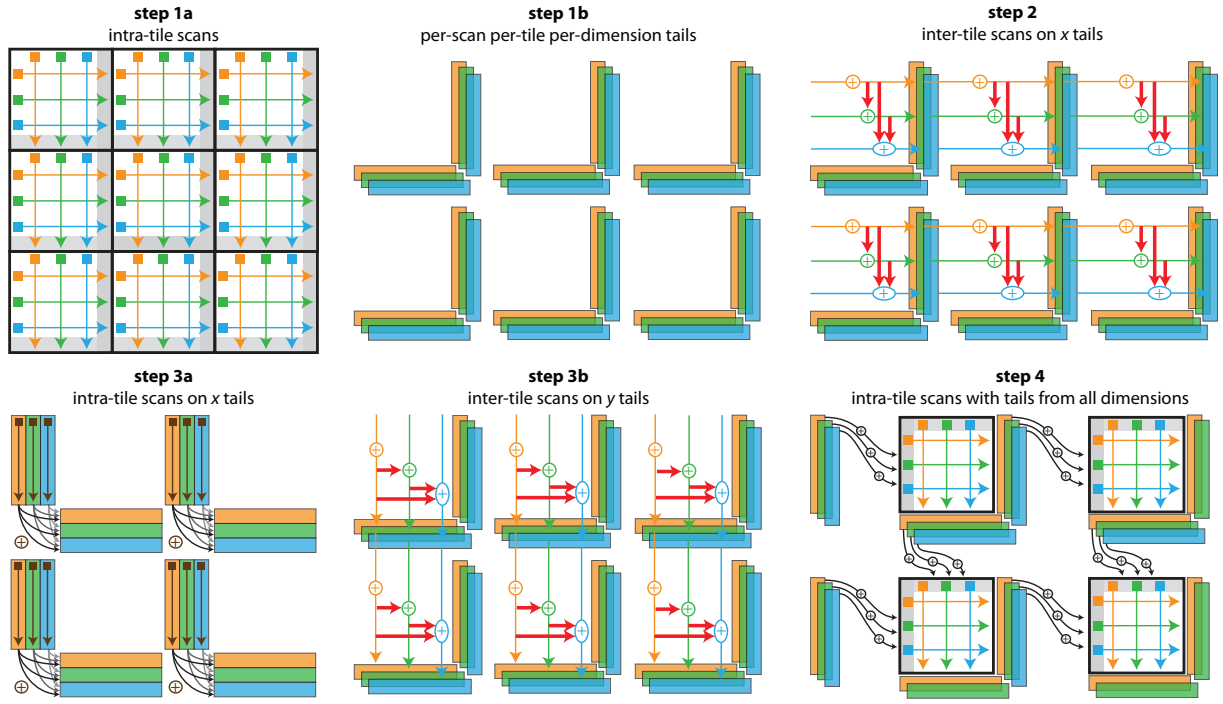

Figure 3: *Multiple 2D filters. Step 1a: we compute all filters within each tile in parallel. Step 1b: we copy the tails after each filter. Step 2: we complete the tails of all the filters along the x dimension using the 1D case as in Fig. 2. Step 3a: to account for cross-dimensional residuals, we apply the y filters on all x tails (vertical arrows) and incorporate these into all the y tails (corner arrows). Step 3b: we use these residuals to complete the y tails. Step 4: we compute the final result using the completed tails of previous tiles from all filters.*

is, for the j^{th} filter, we have j matrices $\{\mathbf{P}_{j,\ell}\}$ with $\ell \in [1; j]$ to account for the tails at the previous tile:

$$\mathbf{f}_{j,x_\text{o}} \leftarrow \mathbf{f}^{\text{intra}}_{j,x_\text{o}} + \sum_{\ell=1}^{j} \mathbf{P}_{j,\ell} \mathcal{T}_{k_j}(\mathbf{f}_{\ell,x_\text{o}-1}) \tag{6}$$

We give the details of the $\mathbf{P}_{j,\ell}$ matrices in Sec. 3. We build upon this equation to design our transformation (Fig. 2).

1. Intra-tile computation and tail storage: For each filter in order, we compute its intra-tile subfunctions $\{f^{\text{intra}}_{j,x_\text{o}}\}$ in parallel (Fig. 2, step 1a). We do this computation in place, i.e., we overwrite $f^{\text{intra}}_{j,x_\text{o}}$ with $f^{\text{intra}}_{j+1,x_\text{o}}$. Since we need the tails $\{\mathcal{T}_{k_j}(\mathbf{f}^{\text{intra}}_{j,x_\text{o}})\}$ for all j, we copy them to a new location before overwriting them. At the end of this step, we have in memory $f^{\text{intra}}_{n,x_\text{o}}$ and all $\{\mathcal{T}_{k_j}(\mathbf{f}^{\text{intra}}_{j,x_\text{o}})\}$ (Fig. 2, step 1b).

2. Inter-tile computation: We update the tails using Equation 6 applied only to tails (Fig. 2, step 2): $\mathcal{T}_{k_j}(\mathbf{f}_{j,x_\text{o}}) \leftarrow \mathcal{T}_{k_j}(\mathbf{f}^{\text{intra}}_{j,x_\text{o}}) + \sum_{\ell=1}^{j} \mathcal{T}_{k_j}(\mathbf{P}_{j,\ell}) \mathcal{T}_{k_j}(\mathbf{f}_{\ell,x_\text{o}-1})$.

3. Intra-tile update: We update the tiles in parallel using the same equation (Fig. 2, step 3).

This transformation generates the same 3 categories of subfunctions as the single-filter case.

2.3 Multiple filters along different dimensions

We now derive the transformation for the generic case where multiple filters are applied along different axes. The first remark is that linear recursive filters are separable along dimensions, they commute, and we can reorder them to group them per axis. That is, we can assume without loss of generality that all the x filters come first, then all the y filters, and so on. We start with the 2D case and then describe its extension to more dimensions.

2.3.1 Two-dimensional case

We consider a 2D domain with n_x filters along the x axis followed by n_y filters along the y axis; this is illustrated in Fig. 3 with a different color per filter. We use the same strategy and perform all the global inter-tile computation only on the tails while doing dense per-pixel computation always in an intra-tile fashion so that it can be parallelized. We also minimize the alternation between inter-tile and intra-tile computation. Because of the complex dependencies introduced by the multiple dimensions, there is no concise update formula equivalent to Equations 5 and 6. Instead we proceed step by step, allowing Equation 6 to be applied multiple times while respecting all dependencies. We use the notation $\mathbf{f}_{j_x,j_y,x_\text{o},y_\text{o}}$ for the (x_o, y_o) tile processed by the j_x^{th} first x filters and the j_y^{th} first y filters, and $\mathcal{T}_x$ and $\mathcal{T}_y$ for the tails in the x and y directions respectively. For brevity's sake, when not ambiguous, we use the shorter notation $\mathbf{f}_{j_x,j_y}$ and omit the length of the tail on the $\mathcal{T}_x$ and $\mathcal{T}_y$ operators.

1. Intra-tile computation and tail storage: For each filter in order, we compute its intra-tile subfunctions in parallel (Fig. 3, step 1a). Similarly to the 1D case, we do the computation in place, and after each filter, we copy the tails along its direction into a new memory location (Fig. 3, step 1b). At the end, we have in memory $\mathbf{f}^{\text{intra}}_{n_x,n_y}$ and the $\{\mathcal{T}_x(\mathbf{f}^{\text{intra}}_{j_x,0})\}$ and $\{\mathcal{T}_y(\mathbf{f}^{\text{intra}}_{n_x,j_y})\}$ tails.

2. Update of the x tails: Since the x filters happen first, we process them similarly to the 1D case and apply Equation 6 to update the tails $\{\mathcal{T}_x(\mathbf{f}^{\text{intra}}_{j_x,0})\}$ to get $\{\mathcal{T}_x(\mathbf{f}_{j_x,0})\}$ (Fig. 3, step 2).

3. Update of the y tails: The y tails are more complex because we need to account for the x filters. The $\{\mathcal{T}_y(\mathbf{f}^{\text{intra}}_{n_x,j_y})\}$ tails that we have stored in the first step have only a partial information about the x filters because they are computed only from intra-tile data. Thus, before we can update these tails along the y axis as previously, we need to "update them with the complete x information". For this, we

use the $\{\mathcal{T}_x(\mathbf{f}_{n_x,0})\}$ tails that we computed in Step 2. We process the y filters in order. We begin with the explanation of the first filter, i.e., $j_y = 1$.

3a. y Filtering of the x tails: We apply the intra-tile y filter to the $\{\mathcal{T}_x(\mathbf{f}_{j_x,0})\}$ tails and keep only the y tails (Fig. 3, step 3a, vertical arrows). We name the results $\{\mathcal{T}_y\big(\mathcal{T}_x(\mathbf{f}^{\text{intra}_y}_{j_x,1})\big)\}$. Since $\mathcal{T}_x$ and $\mathcal{T}_y$ commute, these are also $\{\mathcal{T}_x\big(\mathcal{T}_y(\mathbf{f}^{\text{intra}_y}_{j_x,1})\big)\}$, which is useful in the next step.

3b. x Update of the xy tails: For the rows selected by the $\mathcal{T}_y$ operator, we have $\{\mathcal{T}_x\big(\mathcal{T}_y(f^{\text{intra}_y}_{j_x,1})\big)\}$ tails with the complete result of the x filters (from Step 3a) and dense data $\{\mathcal{T}_y(\mathbf{f}^{\text{intra}}_{n_x,1})\}$ with only intra-tile results (from Step 1). This is the same configuration as the 1D case and we update the y tails by applying Equation 6 along the x axis. This gives us $\mathcal{T}_y(\mathbf{f}^{\text{intra}_y}_{n_x,1})$, i.e., the y tails with the complete x result but still only the intra-tile y result (Fig. 3, step 3a, corner arrows).

3c. y Update of the y tails: We now apply Equation 6 along the y axis and restricted to the y tails to get the $\mathcal{T}_y(\mathbf{f}_{n_x,1})$ tail with the complete xy result that we sought (Fig. 3, step 3b).

3d. Other y filters: We process the y-filters for $j_y > 1$ the same way. The only difference is that we have now more than one y filter to take into account in Equation 6.

4. Final intra-tile update: We now have x and y tails with complete results. We use an extension of Equation 6 for updating all the data from the tails (Fig. 3, step 4):

$$\mathbf{f}_{j_x,j_y,x_\text{o},y_\text{o}} \leftarrow \mathbf{f}^{\text{intra}}_{j_x,j_y,x_\text{o},y_\text{o}} + \sum_{\ell_x=1}^{j_x} \mathbf{P}_{x,j_x,\ell_x} \mathcal{T}_x(\mathbf{f}_{\ell_x,0,x_\text{o}-1,y_\text{o}}) + \sum_{\ell_y=1}^{j_y} \mathbf{P}_{y,j_y,\ell_y} \mathcal{T}_y(\mathbf{f}_{n_x,\ell_y,x_\text{o},y_\text{o}-1}) \quad (7)$$

We give the details of the $\mathbf{P}_{y,j_y,\ell_y}$ matrices in Sec. 3. This transformation introduces two new categories of subfunctions: the y intra-tile filters applied only to the x tails, and the x update of the xy tails.

2.3.2 Higher-dimension case

We extend the transformation that we presented in the previous section to more dimensions by repeating the same procedure that we did to go from one to two axes. That is, we always start by computing intra-tile results and tails after each filter. Then, we process the dimensions in order and update the tails along them. This requires filtering the tails of all the previous dimensions and using them to update the tails of the current dimension. Finally, the intra-tile data is updated using the complete tails adjacent to the tile. This approach ensures that the dense data are processed only twice and in parallel, once at the beginning and once at the end, the rest of the computation involves only tails.

Because updating of the tails of a filter requires taking into account the tails of all the previous filters, the number of generated subfunctions grows quadratically with the number of filters. This makes exploring the scheduling space challenging when developers code each subfunction by hand. With our approach, this is offloaded to the compiler, which makes the exploration easier.

2.4 Anticausal filters

Anticausal filters process data in the order of decreasing indices. We can handle them by introducing two minor modifications to the schemes presented earlier in this section. First, all the intra-tile and inter-tile data traversals related to these filters are also done in reverse order. Second, to update the intra-tile data, we use the tails behind the current tile, that is, in Equations 6 and 7, the data in tile (x_o, y_o) are updated using the tails of the $(x_\text{o}+1, y_\text{o})$ tile if a filter is along the x axis or $(x_\text{o}, y_\text{o}+1)$ if it is along the y axis.

3 Filter matrix computation

We first define the scan matrix $\mathbf{B}$ and tail completion matrix matrix $\mathbf{R}$. $\mathbf{R}$ computes the result of applying a filter on a tile of all elements equal to 0 with tail of previous tile equal to 1. $\mathbf{B}$ computes the result of applying a filter on a tile of all elements equal to 1 and tail of previous tile equal to 0. We will later use these matrices to define $\mathbf{P}$ and $\mathbf{P}_{j,\ell}$ used in the original text.

Note that the procedure to compute the $\mathbf{B}$ and $\mathbf{R}$ is independent of the causality of the filter. Causality will be accounted for when we compute $\mathbf{P}$ and $\mathbf{P}_{j,\ell}$ which are used in the final expressions in the main paper.

3.1 Computation of $\mathbf{B}$

Let the feedforward coefficient of the scan be a_0 and the feedback coefficients be $\{a_1, a_2 \cdots a_k\}$. The procedure to build $\mathbf{B}$ is as follows:

Algorithm 2 Computation of scan matrix $\mathbf{B}$

```
B ← t × t zero matrix
for x = 0 to t − 1 do
    B(x, x) ← a_0
end for
for y = 0 to t − 1 do
    for x = 0 to t − 1 do
        for j = 1 to k do
            if y − j ≥ 0 then
                B(x, y) ← B(x, y) + a_j B(x, y − j)
            end if
        end for
    end for
end for
```

The above algorithm has to be modified if the input images has clamped border as follows:

Algorithm 3 Computation of scan matrix $\mathbf{B}$ for clamped borders

```
B ← t × t zero matrix
for x = 0 to t − 1 do
    B(x, x) ← a_0
end for
for y = 0 to t − 1 do
    for x = 0 to t − 1 do
        for j = 1 to k do
            if y − j ≥ 0 then
                B(x, y) ← B(x, y) + a_j B(x, y − j)
            else
                if x = 0 then
                    B(x, y) ← B(x, y) + a_j
                end if
            end if
        end for
    end for
end for
```

3.2 Computation of $\mathbf{R}$

We describe the procedure to build the matrix $\mathbf{R}$ in Algorithm 4.

Algorithm 4 Computation of tail completion matrix $\mathbf{R}$

```
R ← t × k zero matrix
for y = 0 to t − 1 do
    for x = 1 to k do
        if x + y < k + 1 then
            R(x, y) ← a_{x+y}
        end if
        for j = 1 to k do
            if y − j ≥ 0 then
                R(x, y) ← R(x, y) + a_j R(x, y − j)
            end if
        end for
    end for
end for
```

3.3 Filter matrices $\mathbf{P}$ and $\mathbf{P}_{j,\ell}$

In a single tiled filter, the final result can be computed using the contribution of the tails from the previous tile. $\mathbf{P}$ is therefore simply the last k rows of $\mathbf{R}$.

Consider multiple recursive filters in the same dimensions. The main paper explains that the final result after j-th filter must include the effect of all $\ell \in [0, j]$ tails for each of the proceeding filters. These dependencies can be captured by the matrix $\mathbf{P}_{j,\ell}$ which propagates the tail of ℓ-th filter to the final result of the j-th filter with $\ell \in [0, j]$. We describe the computation of this matrix using the R_ℓ and B_ℓ for each of the ℓ-th filters computed as described above.

Assuming that all filters ℓ have the same causality as filter j, $\mathbf{P}_{j,\ell}$ is simply the concatenation of all matrices from previous filters and the matrix from the j-th filter as under:

$$\mathbf{P}_{j,\ell} = \left(\Pi_{i=\ell}^{j-1}\mathbf{B}_i\right)\mathbf{R}^j$$

In the general case of mix causalities, we define $\mathbf{B}_{i,j}$ as:

$$\mathbf{B}_{i,j} = \begin{cases} \mathbf{B}_i & \text{if filters } i \text{ and } j \text{ have same causality} \\ \tilde{\mathbf{I}}\mathbf{B}_i\tilde{\mathbf{I}} & \text{otherwise} \end{cases}$$

where $\tilde{\mathbf{I}}$ is the antidiagonal matrix. We then modify the expression for $\mathbf{P}_{j,\ell}$ as follows:

$$\mathbf{P}_{j,\ell} = \left(\Pi_{i=\ell}^{j-1}\mathbf{B}_{i,j}\right)\mathbf{R}^j$$

References

NEHAB, D., MAXIMO, A., LIMA, R. S., AND HOPPE, H. 2011. GPU-efficient recursive filtering and summed-area tables. *ACM Trans. Graph. 30*, 6 (Dec.), 176:1–176:12.