GPU-Efficient Recursive Filtering and Summed-Area Tables

Diego Nehab[1] André Maximo[1] Rodolfo S. Lima[2] Hugues Hoppe[3]
[1]IMPA [2]Digitok [3]Microsoft Research

Abstract

Image processing operations like blurring, inverse convolution, and summed-area tables are often computed efficiently as a sequence of 1D recursive filters. While much research has explored parallel recursive filtering, prior techniques do not optimize across the entire filter sequence. Typically, a separate filter (or often a causal-anticausal filter pair) is required in each dimension. Computing these filter passes independently results in significant traffic to global memory, creating a bottleneck in GPU systems. We present a new algorithmic framework for parallel evaluation. It partitions the image into 2D blocks, with a small band of additional data buffered along each block perimeter. We show that these perimeter bands are sufficient to accumulate the effects of the successive filters. A remarkable result is that the image data is read only twice and written just once, independent of image size, and thus total memory bandwidth is reduced even compared to the traditional serial algorithm. We demonstrate significant speedups in GPU computation.

Links: DL PDF WEB VIDEO DATA CODE

1 Introduction

Linear filtering (i.e. convolution) is commonly used to blur, sharpen, or downsample images. A direct implementation evaluating a filter of support d on an $h \times w$-image has cost $O(hwd)$. For filters with a wide impulse response, the Fast Fourier Transform reduces the cost to $O(hw \log hw)$, regardless of filter support. Often, similar results can be obtained with a *recursive filter*, in which the computation reuses prior outputs, e.g. $y_i = x_i - \frac{1}{2} y_{i-1}$. Such feedback allows for an infinite impulse response (IIR), i.e. an effectively large filter support, at reduced cost $O(hwr)$, where the number r of recursive feedbacks (a.k.a. the filter *order*) is small relative to d. Recursive filters are a key computational tool in several applications:

- **Low-pass filtering.** Filters like Gaussian kernels are well approximated by a pair of low-order causal and anticausal recursive filters [e.g. Deriche 1992; van Vliet et al. 1998].
- **Inverse convolution.** If an image X is the result of convolving an image V with a compactly supported filter F, i.e. $X = V * F$, the original image can be recovered as $V = X * F^{-1}$. Although the inverse filter F^{-1} generally has infinite support, it can be expressed *exactly* as a sequence of low-order recursive filters.
- **Summed-area tables.** Such tables store the sum of all pixel values above and to the left of each pixel [Crow 1984]. They have many uses in graphics and vision. On a GPU, summed-area tables are typically computed with prefix sums over all columns then all rows of the image [Hensley et al. 2005]. Crucially, a prefix sum is a special case of a 1D first-order recursive filter.

These applications all have in common the fact that they invoke a *sequence* of recursive filters. First, a 2D operation is decomposed into separate 1D filters. Second, except for the case of summed-area tables, one usually desires a centered and well-shaped impulse response function, and this requires the combination of a causal and anticausal filter pair in each dimension.

As an example, consider the problem of finding the coefficient image V such that reconstruction with the bicubic B-spline interpolates the pixel values of a given image X. (Such a preprocess is required in techniques for high-quality image filtering [Blu et al. 1999, 2001].) The coefficients and image are related by the convolution $X = V * \frac{1}{6}[1\ 4\ 1] * \frac{1}{6}[1\ 4\ 1]^T$. The inverse convolution can be decomposed into four successive 1D recursive filters:

$$\begin{aligned} y_{i,j} &= 6x_{i,j} + \alpha y_{i-1,j}, && \text{(causal in } i\text{)}\\ z_{i,j} &= -\alpha y_{i,j} + \alpha z_{i+1,j}, && \text{(anticausal in } i\text{)}\\ u_{i,j} &= 6z_{i,j} + \alpha u_{i,j-1}, && \text{(causal in } j\text{)}\\ v_{i,j} &= -\alpha u_{i,j} + \alpha v_{i,j+1}, \text{ with } \alpha = \sqrt{3} - 2. && \text{(anticausal in } j\text{)} \end{aligned}$$

Although the parallelization of such recursive filters is challenging due to the many feedback data dependencies, efficient algorithms have been developed as reviewed in section 2.

However, an important element not considered in previous work is optimization over the entire sequence of recursive filters. If each filter is applied independently as is common, the entire image must be read from global memory, and then written back each time. Due to the low computational intensity of recursive filters, this memory traffic becomes a clear bottleneck. Because the data access pattern is different in successive filters (down, up, right, left), the different computation passes cannot be directly fused.

Contribution In this paper, we present a new algorithmic framework to reduce memory bandwidth by *overlapping* computation over the full sequence of recursive filters, across both multiple dimensions and causal-anticausal pairs. The resulting algorithm scales efficiently to images of arbitrary size.

Our approach partitions the image into 2D blocks of size $b \times b$. This blocking serves several purposes. Inter-block parallelism provides sufficient independent tasks to hide memory latency. The blocks fit in on-chip memory for fast intra-block computations. And, the square blocks enables efficient propagation of data in both dimensions.

We associate a set of narrow rectangular buffers with each block's perimeter. These buffers' widths are the same as the filter order r. Intuitively, the goal of these buffers is to measure the filter response of each block in isolation (i.e. given zero boundary conditions), as well as to efficiently propagate the boundary conditions across the blocks given these measured responses.

The surprising result is that we are able to accumulate, with a single block-parallel pass over the input image, all the necessary responses to the filter sequence (down, up, right, left), so as to efficiently initialize boundary conditions for a second block-parallel pass that produces the final solution. Thus, the source image is read only twice, and written just once. A careful presentation requires algebraic derivations, and is detailed in sections 4–5.

Our framework also benefits the computation of summed-area tables (section 6). Even though in that setting there are no anticausal filters, the reduction in bandwidth due solely to the overlapped processing of the two dimensions still provides performance gains.

2 Related work

Prefix sums and scans A prefix sum, $y_i = x_i + y_{i-1}$, is a simple case of a first-order recursive filter. A *scan* generalizes the recurrence using an arbitrary binary associative operator. Parallel prefix sums and scans are important building blocks for numerous algorithms [Iverson 1962; Stone 1971; Blelloch 1989; Sengupta et al. 2007]. Recent work has explored the efficient GPU implementation of scan operations using optimization strategies such as hierarchical reduction, bandwidth reduction through redundant computation, and serial evaluation within shared memory [Sengupta et al. 2007; Dotsenko et al. 2008; Merrill and Grimshaw 2009]. An optimized implementation comes with the CUDPP library [2011].

Recursive filtering Recursive filters generalize prefix sums by considering a weighted combination of prior outputs as indicated in equation (1). This generalization can in fact be implemented as a scan operation with appropriately redefined basic operators [Blelloch 1990]. For recursive filtering of images on the GPU, Ruijters and Thévenaz [2010] exploit parallelism across rows and columns. To expose parallelism within each row and each column, we build on the work of Sung and Mitra [1986, 1992]. The idea is to partition the input into blocks and decompose the computation over each block into two parts: one based only on the block data and assuming zero initial conditions, and the other based only on initial conditions and assuming zero block data (see section 4). We then overlap computation between successive filters passes. As noted by Sung and Mitra, there is also related work on pipelined circuit evaluation of recursive filters [e.g. Parhi and Messerschmitt 1989], but such approaches are more expensive on general-purpose computers.

Tridiagonal systems The forward- and back-substitution passes associated with solving banded linear systems by LU-decomposition are related to recursive filters, particularly for Toeplitz matrices, although the resulting L and U matrices have nonuniform entries at least near the domain boundaries. Forward- and back-substitution passes can be implemented with parallel scan operations [Blelloch 1990]. On GPUs, fast tridiagonal solvers process rows and columns independently with cyclic reduction [Hockney and Jesshope 1981], using either pixel-shader passes over global memory [Kass et al. 2006], or within shared memory using a single kernel [Zhang et al. 2010; Göddeke and Strzodka 2011]. The use of shared memory imposes a limit on the maximum image size. Recent work explores overcoming this limitation [Lamas-Rodrígues et al. 2011]. Our blocking strategy is able to overlap causal and anticausal computations to run efficiently on images of arbitrary size. We hope it can be applied it to general tridiagonal systems in future work.

Summed-area tables The typical approach for efficiently generating summed-area tables on the GPU is to compute parallel prefix sums over all columns, and then over all rows of the result. These prefix sums can be computed with recursive doubling using multiple passes of a pixel shader [Hensley et al. 2005]. As previously mentioned, recent prefix-sum algorithms reduce bandwidth by using GPU shared memory [Harris et al. 2008; Hensley 2010]. In fact, summed-area tables can be computed with little effort using the high-level parallel primitives of the CUDPP library and publicly available matrix transposition routines.

3 Problem definition

Causal recursive filters of order r are characterized by a set of r *feedback* coefficients a_k in the following manner. Given a *prologue* vector $\boldsymbol{p} \in \mathbb{R}^r$ (i.e., the initial conditions) and an input vector $\boldsymbol{x} \in \mathbb{R}^h$, of any size h, the filter $F : \mathbb{R}^r \times \mathbb{R}^h \to \mathbb{R}^h$ produces the output vector $\boldsymbol{y} = F(\boldsymbol{p}, \boldsymbol{x})$ with the same size as the input $\boldsymbol{x}$:

$$y_i = x_i - \sum_{k=1}^{r} a_k y_{i-k}, \quad i \in \{0, \ldots, h-1\} \tag{1}$$

where on the right-hand side of (1) we set as initial conditions

$$y_{-k} = p_{r-k}, \quad k \in \{1, \ldots, r\}. \tag{2}$$

The definition of an anticausal filter $R\colon \mathbb{R}^h \times \mathbb{R}^r \to \mathbb{R}^h$ of order r is analogous. Given input vector $\boldsymbol{y} \in \mathbb{R}^h$ and *epilogue* vector $\boldsymbol{e} \in \mathbb{R}^r$, the output vector $\boldsymbol{z} = R(\boldsymbol{y}, \boldsymbol{e})$ is defined by

$$z_i = y_i - \sum_{k=1}^{r} a'_k z_{i+k}, \quad i \in \{0, \ldots, h-1\}, \tag{3}$$

where on the right-hand side of (3) we set as initial conditions

$$z_{h+k-1} = e_{k-1}, \quad k \in \{1, \ldots, r\}. \tag{4}$$

Causal and anticausal filters frequently appear in pairs. In such cases, we are interested in computing

$$\boldsymbol{z} = R(\boldsymbol{y}, \boldsymbol{e}) = R\big(F(\boldsymbol{p}, \boldsymbol{x}), \boldsymbol{e}\big). \tag{5}$$

In image processing applications, it is often desirable to apply filters independently to each row and to each column of an image (i.e., in a separable way). To process all image columns, we extend the definition of F to operate independently on all corresponding columns of an $r \times w$ prologue matrix P and an $h \times w$ input matrix X. The extended filter is described by $F\colon \mathbb{R}^{r\times w} \times \mathbb{R}^{h\times w} \to \mathbb{R}^{h\times w}$.

For row processing, we define a new extended filter F^{τ} that operates independently on corresponding rows of an $h \times w$ input matrix X and an $h \times r$ prologue matrix P'. This *transposed* filter is described by $F^{\tau}\colon \mathbb{R}^{h\times r} \times \mathbb{R}^{h\times w} \to \mathbb{R}^{h\times w}$. Anticausal filters can also be extended to operate independently over multiple columns, $R\colon \mathbb{R}^{h\times w} \times \mathbb{R}^{r\times w} \to \mathbb{R}^{h\times w}$, and over multiple rows, $R^{\tau}\colon \mathbb{R}^{h\times w} \times \mathbb{R}^{h\times r} \to \mathbb{R}^{h\times w}$.

With these definitions, we are ready to formulate the problem of applying the full sequence of four recursive filters (down, up, right, left) to an image. As shown in figure 1, given an input matrix X and the prologues (P, P') and epilogues (E, E') that surround its columns and rows, we wish to compute matrix V, where

$$\begin{aligned} Y &= F(P, X), & Z &= R(Y, E), \\ U &= F^{\tau}(P', Z), & V &= R^{\tau}(U, E'). \end{aligned} \tag{6}$$

In particular, we want to take advantage of the massive parallelism offered by modern GPUs.

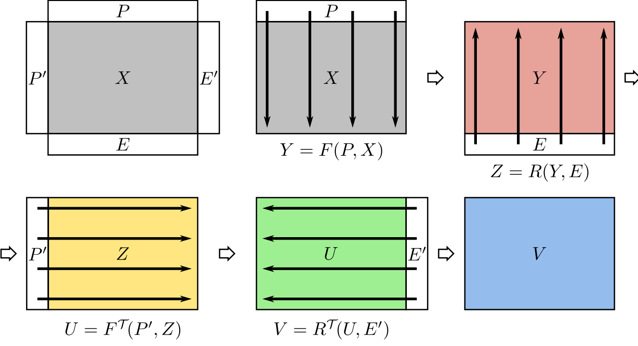


Figure 1: *The problem is to compute for an image X a sequence of four 1D recursive filters (down, up, right, left), with boundary conditions given by prologue P,P' and epilogue E,E' matrices.*

Design goals on GPUs Modern GPUs are composed of p streaming multiprocessors (SMs), each containing c computational cores. For example, our test hardware (an NVIDIA GTX 480) contains $p = 15$ SMs, each with $c = 32$ cores. Because the cores in each SM execute instructions in lockstep, divergence in execution paths should be minimized for maximum performance. SMs execute independently from each other, but typically run the same program, called a *kernel*, operating over distinct parts of the input.

Although bandwidth to global memory is higher than on a CPU, access must be coalesced for maximum performance. Ideally, all cores should access aligned, consecutive memory addresses. Furthermore, hiding memory latency requires a large number of independent tasks to be available for scheduling. Each SM contains a small amount of fast memory (48KB in our case) shared among its cores. This memory is typically used as a managed cache to reduce the number of expensive accesses to global memory.

Mapping recursive filtering to GPU architectures presents several challenges. One is the dependency chain in the computation, which reduces the number of tasks that can be executed independently. The other problem is the high bandwidth-to-FLOP ratio, particularly in low-order filters. Without a large number of schedulable tasks to hide memory latency, processing cores become idle waiting for memory transactions to complete, and performance suffers.

Given these considerations, the main design goal of our work is to increase the amount of parallelism *without* increasing the total memory bandwidth required to complete the computation.

4 Prior parallelization strategies revisited

In large images, the independence of column and row processing offers a significant level of parallelism. With that in mind, here is the approach followed by Ruijters and Thévenaz [2010]:

Algorithm RT

RT.1 In parallel for each column in X, apply F sequentially according to (1). Store Y.

RT.2 In parallel for each column in Y, apply R sequentially according to (3). Store Z.

RT.3 In parallel for each row in Z, apply $F^{\mathcal{T}}$ sequentially according to (1). Store U.

RT.4 In parallel for each row in U, apply $R^{\mathcal{T}}$ sequentially according to (3). Store V.

Algorithm RT uses bandwidth proportional to $8hw$, and completes in $O(\frac{hw}{cp}4r)$ steps. The number of tasks that can be performed in parallel is limited by h or w. In a GPU with 15 SMs and 32 cores per SM, we need at least 480 rows and columns to keep all cores busy. Much larger images are required for effective latency hiding.

Roadmap for improvements A common approach to improving performance is to restructure the computation so memory accesses are coalesced (during row as well as column processing). This is one of the reasons we employ a blocking strategy (section 4.1). We introduce inter-block parallelism by adapting the method of Sung and Mitra [1986] to modern GPU architectures (section 4.3). The inter-block parallelism also lets us fuse computations, thereby reducing traffic to global memory (section 4.4).

The fundamental contributions of our work begin in section 5.1, where we develop an overlapping strategy for merging causal and anticausal filtering passes. The idea culminates in section 5.2, where we present an overlapped algorithm that merges row and column processing as well. Finally, in section 6 we specialize it to compute summed-area tables. The resulting reduction in memory bandwidth combined with the high degree of parallelism enable our overlapped algorithms to outperform prior approaches.

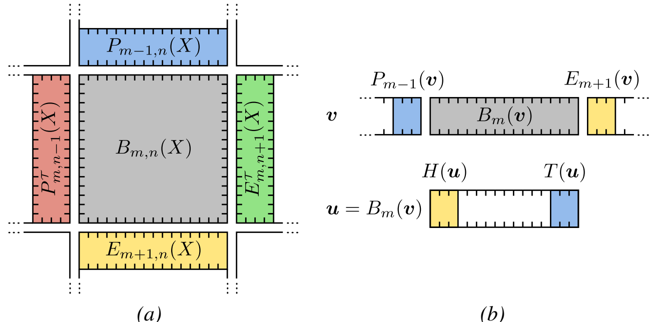


Figure 2: *(a) 2D block notation showing a block and its boundary data from adjacent blocks. (b) 1D block notation, showing vectors as rows for layout reasons.*

4.1 Block notation

We partition the image into blocks of $b \times b$ elements, where b matches the number of threads executed in lockstep (the *warp* size) within an SM, i.e., $b = c = 32$ throughout our work.

As shown in figure 2(a), we define operators to select a block and its boundary data in adjacent blocks. $B_{m,n}(X)$ identifies in matrix X the $b \times b$ data block with index (m, n). $P_{m-1,n}(X)$ extracts the $r \times b$ submatrix in the same shape and position as the column-prologue to block $B_{m,n}(X)$. Similarly, $E_{m+1,n}(X)$ denotes the $r \times b$ submatrix in the position of the column-epilogue to block $B_{m,n}(X)$. For row processing, transposed operators $P^{\mathcal{T}}_{m,n-1}(X)$ and $E^{\mathcal{T}}_{m,n+1}(X)$ are defined accordingly.[1]

It will often be useful to select prologue- and epilogue-shaped submatrices from a $b \times b$ block. To that end, we define a *tail* operation T (for prologues) and a *head* operation H (for epilogues). Transposed counterparts $T^{\mathcal{T}}$ and $H^{\mathcal{T}}$ are defined analogously:

$$\begin{aligned} P_{m,n}(X) &= T\big(B_{m,n}(X)\big), \quad E_{m,n}(X) = H\big(B_{m,n}(X)\big), \\ P^{\mathcal{T}}_{m,n}(X) &= T^{\mathcal{T}}\big(B_{m,n}(X)\big), \; E^{\mathcal{T}}_{m,n}(X) = H^{\mathcal{T}}\big(B_{m,n}(X)\big). \end{aligned} \tag{7}$$

For simplicity, when describing unidimensional operations, we will drop the vertical block index n, as shown in figure 2(b).

Given these definitions, the blocked version of the problem formulation in (6) is to compute all output blocks $B_{m,n}(V)$ where:

$$\begin{aligned} B_{m,n}(Y) &= F\big(P_{m-1,n}(Y), B_{m,n}(X)\big), \\ B_{m,n}(Z) &= R\big(B_{m,n}(Y), E_{m+1,n}(Z)\big), \\ B_{m,n}(U) &= F^{\mathcal{T}}\big(P^{\mathcal{T}}_{m,n-1}(U), B_{m,n}(Z)\big), \\ B_{m,n}(V) &= R^{\mathcal{T}}\big(B_{m,n}(U), E^{\mathcal{T}}_{m,n+1}(V)\big). \end{aligned} \tag{8}$$

All algorithms described from now on employ blocking to guarantee coalesced memory access patterns. Typically, each stage loads an entire block of data to shared memory, performs computation in shared memory, and finally writes the data back to global memory.

4.2 Important properties

We first introduce key properties of recursive filtering that will significantly simplify the presentation. The first property of interest is *superposition*, and it comes from linearity. The effects of the input and the prologue (or epilogue) on the output can be computed independently and later added together:

$$F(\boldsymbol{p}, \boldsymbol{x}) = F(\boldsymbol{0}, \boldsymbol{x}) + F(\boldsymbol{p}, \boldsymbol{0}), \tag{9}$$

$$R(\boldsymbol{y}, \boldsymbol{e}) = R(\boldsymbol{y}, \boldsymbol{0}) + R(\boldsymbol{0}, \boldsymbol{e}). \tag{10}$$

[1] Note that $P^{\mathcal{T}}(\cdot)$ is *not* the transpose of $P(\cdot)$ (but has the same shape).

Notice that the first term $F(\mathbf{0}, \boldsymbol{x})$ does not depend on the prologue, whereas the second term $F(\boldsymbol{p}, \mathbf{0})$ does not depend on the input. Similar remarks hold for R. We will use superposition extensively to split dependency chains in the computation.

Because filtering operations are linear, we can express them as matrix products. This will help reveal useful associative relations. For any d, let I_d denote the identity matrix of size $d \times d$. Then,

$$
\begin{aligned}
&F(\boldsymbol{p}, \mathbf{0}) = F(I_r, \mathbf{0})\, \boldsymbol{p} = A_{FP}\, \boldsymbol{p}, \quad A_{FP} = F(I_r, \mathbf{0}),\\
&R(\mathbf{0}, \boldsymbol{e}) = R(\mathbf{0}, I_r)\, \boldsymbol{e} = A_{RE}\, \boldsymbol{e}, \quad A_{RE} = R(\mathbf{0}, I_r),\\
&F(\mathbf{0}, \boldsymbol{x}) = F(\mathbf{0}, I_b)\, \boldsymbol{x} = A_{FB}\, \boldsymbol{x}, \quad A_{FB} = F(\mathbf{0}, I_b), \text{ and}\\
&R(\boldsymbol{y}, \mathbf{0}) = R(I_b, \mathbf{0})\, \boldsymbol{y} = A_{RB}\, \boldsymbol{y}, \quad A_{RB} = R(I_b, \mathbf{0}).
\end{aligned} \tag{11}
$$

Here, A_{FB} and A_{RB} are $b \times b$ matrices, A_{FP} is $b \times r$, and A_{RE} is $b \times r$. Notice that whereas the columns of A_{FB} and A_{RB} are shifted impulse responses (which allows for efficient storage), the columns of A_{FP} and A_{RE} are generally *not* shifts of each other.

Row processing is analogous, except for transposition. This is the justification for our notation choice:

$$
\begin{aligned}
&F^{\mathcal{T}}(\boldsymbol{p}^T, \mathbf{0}) = \boldsymbol{p}^T\,(A_{FP})^T, \quad R^{\mathcal{T}}(\mathbf{0}, \boldsymbol{e}^T) = \boldsymbol{e}^T\,(A_{RE})^T,\\
&F^{\mathcal{T}}(\mathbf{0}, \boldsymbol{x}^T) = \boldsymbol{x}^T\,(A_{FB})^T, \quad R^{\mathcal{T}}(\boldsymbol{y}^T, \mathbf{0}) = \boldsymbol{y}^T\,(A_{RB})^T.
\end{aligned} \tag{12}
$$

A trivial property of matrix product lets us write

$$
T(A\,\boldsymbol{p}) = T(A)\,\boldsymbol{p} \quad \text{and} \quad H(A\,\boldsymbol{e}) = H(A)\,\boldsymbol{e}. \tag{13}
$$

Finally, as shown in appendix A,

$$
T(A_{FP}) = A_F^b \quad \text{and} \quad H(A_{RE}) = A_R^b, \tag{14}
$$

where A_F^b, A_R^b are precomputed $r \times r$-matrices that depend only on the feedback coefficients a_k, a'_k of filters F and R, respectively.

4.3 Inter-block parallelism

We start the description in one dimension for simplicity. When attempting to perform block computations independently, we are faced with the difficulty that each output block

$$
B_m(\boldsymbol{y}) = F\big(P_{m-1}(\boldsymbol{y}), B_m(\boldsymbol{x})\big) \tag{15}
$$

depends on the prologue $P_{m-1}(\boldsymbol{y})$, i.e., on the tail end of the previous output block, $T\big(B_{m-1}(\boldsymbol{y})\big)$. This creates a dependency chain that we must work around.

By the superposition property (9), we can decompose (15) as

$$
B_m(\boldsymbol{y}) = F\big(\mathbf{0}, B_m(\boldsymbol{x})\big) + F\big(P_{m-1}(\boldsymbol{y}), \mathbf{0}\big). \tag{16}
$$

The first term can be computed independently for each block (inter-block parallelism) because it assumes a zero prologue. We use symbol $\bar{\boldsymbol{y}}$ to denote this *incomplete causal output*:

$$
B_{m,n}(\bar{\boldsymbol{y}}) = F\big(\mathbf{0}, B_m(\boldsymbol{x})\big), \tag{17}
$$

where each block is filtered as if its prologue was null.

The second term of (16) is more challenging due to the dependency chain. Nevertheless, as soon as the dependency is satisfied, each element in a block can be computed independently (intra-block parallelism). To see this, apply property (11):

$$
\begin{aligned}
F(P_{m-1}(\boldsymbol{y}), \mathbf{0}) &= F(I_r, \mathbf{0})\, P_{m-1}(\boldsymbol{y}) && (18)\\
&= A_{FP}\, P_{m-1}(\boldsymbol{y}) && (19)
\end{aligned}
$$

and recall that A_{FP} depends only on the feedback coefficients.

Sung and Mitra [1986] perform unidimensional recursive filtering on a CRAY X-MP computer. Their algorithm works in two stages. First, each processor works on a separate block, computing $B_m(\bar{\boldsymbol{y}})$ sequentially. Second, the blocks are corrected sequentially, according to (16) and (19), with each block's elements updated in parallel across processors. Although this algorithm requires twice the bandwidth and computation of the sequential algorithm, the amount of available parallelism is increased by a factor h/b (i.e., the number of blocks) in the first stage, and by a factor b in the second stage.

Inspired by the bandwidth reduction strategy described by Dotsenko et al. [2008] in the context of scan primitives, we proceed differently. Notice that the dependency chain only involves prologues. Using properties (11), (13), and (14), we can make this apparent:

$$
\begin{aligned}
P_m(\boldsymbol{y}) &= T\big(B_m(\boldsymbol{y})\big) = T\big(F(P_{m-1}(\boldsymbol{y}), B_m(\boldsymbol{x}))\big) && (20)\\
&= T\big(F\big(\mathbf{0}, B_m(\boldsymbol{x})\big)\big) + T\big(F\big(P_{m-1}(\boldsymbol{y}), \mathbf{0}\big)\big) && (21)\\
&= T\big(B_m(\bar{\boldsymbol{y}})\big) + T\big(F(I_r, \mathbf{0})\, P_{m-1}(\boldsymbol{y})\big) && (22)\\
&= P_m(\bar{\boldsymbol{y}}) + T(A_{FP})\, P_{m-1}(\boldsymbol{y}) && (23)\\
&= P_m(\bar{\boldsymbol{y}}) + A_F^b\, P_{m-1}(\boldsymbol{y}). && (24)
\end{aligned}
$$

Therefore, in contrast to the work of Sung and Mitra [1986], we refrain from storing the entire blocks $B_m(\bar{\boldsymbol{y}})$. Instead, we store only the *incomplete prologues* $P_m(\bar{\boldsymbol{y}})$. Equation (24) then allows us to sequentially compute one complete prologue from the previous by means of multiplication by the $r \times r$ matrix A_F^b and an r-vector addition with the incomplete $P_m(\bar{\boldsymbol{y}})$. The resulting algorithm is:

Algorithm 1

1.1 In parallel for all m, compute and store each $P_m(\bar{\boldsymbol{y}})$.

1.2 Sequentially for each m, compute and store the $P_m(\boldsymbol{y})$ according to (24) and using the previously computed $P_m(\bar{\boldsymbol{y}})$.

1.3 In parallel for all m, compute and store output block $B_m(\boldsymbol{y})$ using (15) and the previously computed $P_{m-1}(\boldsymbol{y})$.

The algorithm does some redundant work by computing the recursive filter within each block in both stages 1.1 and 1.3. However, the reduction in memory bandwidth due to not writing the block in stage 1.1 provides a net benefit.

Although stage 1.2 is still sequential, it typically executes faster than the remaining stages. This is because we focus on small filter orders r with block size b large in comparison. This causes stage 1.2 to account for a small fraction of the total I/O and computation. Although we have not experienced a need for it, a hierarchical parallelization strategy is possible, as outlined in section 7.

The derivation for the anticausal counterpart to algorithm 1 follows closely along the same lines and is omitted for brevity. We include the anticausal versions of equations (15), (17), and (24) for completeness. Assuming $\boldsymbol{z} = R(\boldsymbol{y}, \boldsymbol{e})$:

$$
\begin{aligned}
B_m(\boldsymbol{z}) &= R\big(B_m(\boldsymbol{y}), E_{m+1}(\boldsymbol{z})\big) && (25)\\
B_m(\bar{\boldsymbol{z}}) &= R\big(B_m(\boldsymbol{y}), \mathbf{0}\big) && (26)\\
E_m(\boldsymbol{z}) &= E_m(\bar{\boldsymbol{z}}) + A_R^b E_{m+1}(\boldsymbol{z}). && (27)
\end{aligned}
$$

Row operations are analogous and are also omitted.

When applying causal and anticausal filter pairs to process all columns and rows in an image, the four successive applications of algorithm 1 require $O\big(\frac{hw}{cp}(8 + 4\frac{1}{b}(r^2+r)\big)$ steps to complete. The hw/b tasks that can be performed independently provide the scheduler with sufficient freedom to hide memory access latency.

The repeated passes over the input increase memory bandwidth to $(12 + 16r/b)hw$, which is significantly more than the $8hw$ required by algorithm RT. Fortunately, we can reduce bandwidth by employing the *kernel fusion* technique.

4.4 Kernel fusion

As it is known in the literature [Kirk and Hwu 2010], kernel fusion is an optimization that consists of directly using the output of one kernel as the input for the next *without* going through global memory. The fused kernel executes the code of both original kernels, keeping intermediate results in shared memory.

Using algorithm 1 for all filters, we fuse the last stage of F with the first stage of R. We also fuse the last stage of $F^{\mathcal{T}}$ with the first stage of $R^{\mathcal{T}}$. Combined, the fusion of causal and anticausal processing lowers the required bandwidth to $(10 + 16r/b)hw$. Row and column processing are also fused. The last stage of R and the first stage of $F^{\mathcal{T}}$ are combined to further reduce bandwidth usage to $(9 + 16r/b)hw$.

The resulting algorithm is presented below:

Algorithm 2

2.1 In parallel for all m and n, compute and store the $P_{m,n}(\bar{Y})$.

2.2 Sequentially for each m, but in parallel for each n, compute and store the $P_{m,n}(Y)$ according to (24) and using the previously computed $P_{m,n}(\bar{Y})$.

2.3 In parallel for all m and n, compute and store $B_{m,n}(Y)$ using the previously computed $P_{m-1,n}(Y)$. Then immediately compute and store the $E_{m,n}(\bar{Z})$.

2.4 Sequentially for each m, but in parallel for each n, compute and store the $E_{m,n}(Z)$ according to (27) and using the previously computed $E_{m,n}(\bar{Z})$.

2.5 In parallel for all m and n, compute and store $B_{m,n}(Z)$ using the previously computed $E_{m+1,n}(Z)$. Then immediately compute and store the $P^{\mathcal{T}}_{m,n}(\bar{U})$.

2.6 Sequentially for each n, but in parallel for each m, compute and store the $P^{\mathcal{T}}_{m,n}(U)$.

2.7 In parallel for all m and n, compute and store $B_{m,n}(U)$ using the previously computed $P^{\mathcal{T}}_{m,n-1}(U)$. Then compute and store the $E^{\mathcal{T}}_{m,n}(\bar{V})$.

2.8 Sequentially for each n, but in parallel for each m, compute and store the $E^{\mathcal{T}}_{m,n}(V)$.

2.9 In parallel for all m and n, compute and store $B_{m,n}(V)$ using the previously computed $E^{\mathcal{T}}_{m,n+1}(V)$.

The combination of coalesced memory accesses, increased parallelism, and kernel fusion make algorithm 2 substantially faster than algorithm RT (see section 8).

We could aggressively reduce I/O even further by refraining from storing any of the intermediary results Y, Z, and U at stages 2.3, 2.5, and 2.7, respectively. These can be recomputed from X and the appropriate prologues/epilogues when needed. Bandwidth would decrease to $(6 + 22r/b)hw$. (Notice that this is even less than the sequential algorithm RT!) Unfortunately, the repeated computation of Y, Z, and U increases the total number of steps to $O\big(\frac{hw}{cp}(14r + 4\frac{1}{b}(r^2+r)\big)$, and offsets the gains from the bandwidth reduction. Even though the tradeoff is not advantageous for our test hardware, future GPU generations may tip the balance in the other direction.

Conceptually, buffers $P_{m,n}(\bar{Y})$, $E_{m,n}(\bar{Z})$, $P^{\mathcal{T}}_{m,n}(\bar{U})$, $E^{\mathcal{T}}_{m,n}(\bar{V})$ can be thought of as narrow bands around the perimeter of each block $B_{m,n}(X)$. However, these buffers are stored compactly in separate data arrays to maintain efficient memory access.

5 Overlapping

Our main contribution is the development of an *overlapping* technique which achieves further bandwidth reductions *without* any substantial increase in computational cost.

5.1 Causal-anticausal overlapping

One source of inefficiency in algorithm 2 is that we wait for the complete causal output block $B_{m,n}(Y)$ in stage 2.3 before we obtain the incomplete anticausal epilogues $E_{m,n}(\bar{Z})$. The important insight is that it is possible to work instead with *twice-incomplete* anticausal epilogues $E_{m,n}(\hat{Z})$, computed directly from the incomplete causal output block $B_{m,n}(\bar{Y})$. Even though these are obtained before $B_{m,n}(Y)$ are available, we are still able to compute the complete epilogues $E_{m,n}(Z)$ from them. We name this strategy for obtaining prologues $P_{m,n}(Y)$ and epilogues $E_{m,n}(Z)$ with one fewer pass over the input *causal-anticausal overlapping*.

The trick is to apply properties (9) and (10) to split the dependency chains of anticausal epilogues. Proceeding in one dimension for simplicity:

$$E_m(\boldsymbol{z}) = H\big(R\big(B_m(\boldsymbol{y}), E_{m+1}(\boldsymbol{z})\big)\big) \tag{28}$$

$$= H\big(R\big(F(P_{m-1}(\boldsymbol{y}), B_m(\boldsymbol{x})), E_{m+1}(\boldsymbol{z})\big)\big) \tag{29}$$

$$\begin{aligned} &= H\big(R\big(\mathbf{0}, E_{m+1}(\boldsymbol{z})\big)\big) \\ &\quad + H\big(R\big(F(P_{m-1}(\boldsymbol{y}), B_m(\boldsymbol{x})), \mathbf{0}\big)\big) \end{aligned} \tag{30}$$

$$\begin{aligned} &= H\big(R\big(\mathbf{0}, E_{m+1}(\boldsymbol{z})\big)\big) \\ &\quad + H\big(R\big(F(P_{m-1}(\boldsymbol{y}), \mathbf{0}), \mathbf{0}\big)\big) \\ &\quad + H\big(R\big(F(\mathbf{0}, B_m(\boldsymbol{x})), \mathbf{0}\big)\big). \end{aligned} \tag{31}$$

We can simplify further using (11), (13) and (14) to reach

$$\begin{aligned} E_m(\boldsymbol{z}) &= H\big(R(\mathbf{0}, I_r)\, E_{m+1}(\boldsymbol{z})\big) \\ &\quad + H\big(R(I_b, \mathbf{0})\, F(I_r, \mathbf{0})\, P_{m-1}(\boldsymbol{y})\big) \\ &\quad + H\big(R\big(F(\mathbf{0}, B_m(\boldsymbol{x})), \mathbf{0}\big)\big) \end{aligned} \tag{32}$$

$$\begin{aligned} &= H\big(R(\mathbf{0}, I_r)\big)\, E_{m+1}(\boldsymbol{z}) \\ &\quad + H\big(R(I_b, \mathbf{0})\big)\, F(I_r, \mathbf{0})\, P_{m-1}(\boldsymbol{y}) \\ &\quad + H\big(R\big(F(\mathbf{0}, B_m(\boldsymbol{x})), \mathbf{0}\big)\big) \end{aligned} \tag{33}$$

$$\begin{aligned} &= A_R^b\, E_{m+1}(\boldsymbol{z}) \\ &\quad + \big(H(A_{RB})\, A_{FP}\big)\, P_{m-1}(\boldsymbol{y}) \\ &\quad + E_m(\hat{\boldsymbol{z}}) \end{aligned} \tag{34}$$

where the twice-incomplete $\hat{\boldsymbol{z}}$ is such that

$$B_m(\hat{\boldsymbol{z}}) = R\big(F(\mathbf{0}, B_m(\boldsymbol{x})), \mathbf{0}\big) \tag{35}$$

$$= R(\bar{\boldsymbol{y}}, \mathbf{0}). \tag{36}$$

Notice that the $r \times r$ matrix $H(A_{RB})\, A_{FP}$ used in (34) can be precomputed. Furthermore, each twice-incomplete epilogue $E_m(\hat{\boldsymbol{z}})$ depends only on the corresponding input block $B_m(\boldsymbol{x})$ and therefore they can all be computed in parallel already in the first pass. As a byproduct of that same pass, we can compute and store the $P_m(\bar{\boldsymbol{y}})$ that will be needed to obtain $P_m(\boldsymbol{y})$. With $P_m(\boldsymbol{y})$, we can compute all $E_m(\boldsymbol{z})$ sequentially with equation (34).

The resulting one-dimensional algorithm is as follows:

Algorithm 3

3.1 In parallel for all m, compute and store $P_m(\bar{\boldsymbol{y}})$ and $E_m(\hat{\boldsymbol{z}})$.

3.2 Sequentially for each m, compute and store the $P_m(\boldsymbol{y})$ according to (24) and using the previously computed $P_m(\bar{\boldsymbol{y}})$.

3.3 Sequentially for each m, compute and store $E_m(\boldsymbol{z})$ according to (34) using the previously computed $P_{m-1}(\boldsymbol{y})$ and $E_m(\hat{\boldsymbol{z}})$.

3.4 In parallel for all m, compute each causal output block $B_m(\boldsymbol{y})$ using (15) and the previously computed $P_{m-1}(\boldsymbol{y})$. Then compute and store each anticausal output block $B_m(\boldsymbol{z})$ using (25) and the previously computed $E_{m+1}(\boldsymbol{z})$.

Using algorithm 3 for both row and column processing and fusing the two stages leads to:

Algorithm 4

4.1 In parallel for all m and n, compute and store the $P_{m,n}(\bar{Y})$ and $E_{m,n}(\hat{Z})$.

4.2 Sequentially for each m, but in parallel for each n, compute and store the $P_{m,n}(Y)$ according to (24) and using the previously computed $P_{m,n}(\bar{Y})$.

4.3 Sequentially for each m, but in parallel for each n, compute and store the $E_{m,n}(Z)$ according to (34) using the previously computed $P_{m-1,n}(Y)$ and $E_{m,n}(\hat{Z})$.

4.4 In parallel for all m and n, compute $B_{m,n}(Y)$ using the previously computed $P_{m-1,n}(Y)$. Then compute and store the $B_{m,n}(Z)$ using the previously computed $E_{m+1,n}(Z)$. Finally, compute and store both $P^{\mathcal{T}}_{m,n}(\bar{U})$ and $E^{\mathcal{T}}_{m,n}(\hat{V})$.

4.5 Sequentially for each n, but in parallel for each m, compute and store the $P^{\mathcal{T}}_{m,n}(U)$ from $P^{\mathcal{T}}_{m,n}(\bar{U})$.

4.6 Sequentially for each n, but in parallel for each m, compute and store each $E^{\mathcal{T}}_{m,n}(V)$ using the previously computed $P^{\mathcal{T}}_{m,n-1}(U)$ and $E^{\mathcal{T}}_{m,n}(\hat{V})$.

4.7 In parallel for all m and n, compute $B_{m,n}(V)$ using the previously computed $P^{\mathcal{T}}_{m,n-1}(V)$ and $B_{m,n}(Z)$. Then compute and store the $B_{m,n}(U)$ using the previously computed $E_{m,n+1}(U)$.

As expected, algorithm 4 is faster than algorithm 2 (see section 8). The advantage stems from a similar $O\big(\frac{hw}{cp}(8r+6\frac{1}{b}(r^2+r)\big)$ number of steps combined with lower $(5+18r/b)hw$ bandwidth.

5.2 Row-column causal-anticausal overlapping

There is still one source of inefficiency: we wait until the complete block $B_{m,n}(Z)$ is available in stage 4.4 before computing incomplete $P^{\mathcal{T}}_{m,n}(\bar{U})$ and twice-incomplete $E^{\mathcal{T}}_{m,n}(\hat{V})$. Fortunately, we can overlap row and column computations and work with *thrice-incomplete* transposed prologues and *four-times-incomplete* transposed epilogues obtained directly during stage 4.1. From these, we can compute the complete $P^{\mathcal{T}}_{m,n}(U)$ and $E^{\mathcal{T}}_{m,n}(V)$ without going over the input again.

The derivation of the procedure for completing the transposed prologues and epilogues is somewhat tedious, and can be found in full in appendix B. Below is the formula for completing thrice-incomplete transposed prologues:

$$\begin{aligned} P^{\mathcal{T}}_{m,n}(U) &= P^{\mathcal{T}}_{m,n-1}(U)\,(A_F^b)^T \\ &\quad + A_{RE}\,\big(E_{m+1,n}(Z)\,\big(T(A_{FB})\big)^T\big) \\ &\quad + (A_{RB}\,A_{FP})\,\big(P_{m-1,n}(Y)\,\big(T(A_{FB})\big)^T\big) \\ &\quad + P^{\mathcal{T}}_{m,n}(\check{U}), \end{aligned} \tag{37}$$

where $b\times r$ matrix A_{RE} as well as $b\times r$ matrices $A_{RB}\,A_{FP}$ and $T(A_{FB})$ can all be precomputed. The thrice-incomplete $\check{U}$ satisfies

$$B_{m,n}(\check{U}) = F^{\mathcal{T}}\big(\mathbf{0}, B_{m,n}(\hat{Z})\big). \tag{38}$$

To complete the four-times-incomplete transposed epilogues of V:

$$\begin{aligned} E^{\mathcal{T}}_{m,n}(V) &= E^{\mathcal{T}}_{m,n+1}(V)\,(A_R^b)^T \\ &\quad + P^{\mathcal{T}}_{m,n-1}(U)\,\big(H(A_{RB})\,A_{FP}\big)^T \\ &\quad + A_{RE}\,\big(E_{m+1,n}(Z)\,\big(H(A_{RB})\,A_{FB}\big)^T\big) \\ &\quad + (A_{RB}\,A_{FP})\big(P_{m-1,n}(Y)\,\big(H(A_{RB})\,A_{FB}\big)^T\big) \\ &\quad + E^{\mathcal{T}}_{m,n}(\tilde{V}). \end{aligned} \tag{39}$$

Here, the $r\times r$ matrix $H(A_{RB})\,A_{FP}$ is the same as in (34), and the $b\times r$ matrix $A_{RB}\,A_{FP}$ is the same as in (37). The new $r\times b$ matrix $H(A_{RB})\,A_{FB}$ is also precomputed. Finally, the four-times-incomplete $\tilde{V}$ is

$$B_{m,n}(\tilde{V}) = R^{\mathcal{T}}\big(B_{m,n}(\check{U}), \mathbf{0}\big). \tag{40}$$

The fully overlapped algorithm is:

Algorithm 5

5.1 In parallel for all m and n, compute and store each $P_{m,n}(\bar{Y})$, $E_{m,n}(\hat{Z})$, $P^{\mathcal{T}}_{m,n}(\check{U})$, and $E^{\mathcal{T}}_{m,n}(\tilde{V})$.

5.2 In parallel for all n, sequentially for each m, compute and store the $P_{m,n}(Y)$ according to (24) and using the previously computed $P_{m-1,n}(\bar{Y})$.

5.3 In parallel for all n, sequentially for each m, compute and store $E_{m,n}(Z)$ according to (34) and using the previously computed $P_{m-1,n}(Y)$ and $E_{m+1,n}(\hat{Z})$.

5.4 In parallel for all m, sequentially for each n, compute and store $P^{\mathcal{T}}_{m,n}(U)$ according to (37) and using the previously computed $P^{\mathcal{T}}_{m,n}(\check{U})$, $P_{m-1,n}(Y)$, and $E_{m+1,n}(Z)$.

5.5 In parallel for all m, sequentially for each n, compute and store $E^{\mathcal{T}}_{m,n}(V)$ according to (39), using the previously computed $E^{\mathcal{T}}_{m,n}(\tilde{V})$, $P^{\mathcal{T}}_{m,n-1}(U)$, $P_{m-1,n}(Y)$, and $E_{m+1,n}(Z)$.

5.6 In parallel for all m and n, successively compute $B_{m,n}(Y)$, $B_{m,n}(Z)$, $B_{m,n}(U)$, and $B_{m,n}(V)$ according to (8) and using the previously computed $P_{m-1,n}(Y)$, $E_{m+1,n}(Z)$, $P^{\mathcal{T}}_{m,n-1}(U)$, and $E^{\mathcal{T}}_{m,n+1}(V)$. Store $B_{m,n}(V)$.

The number of steps required by the fully overlapped algorithm 5 is only $O\big(\frac{hw}{cp}(8r+\frac{1}{b}(18r^2+10r))\big)$. In our usage scenario, the filter order is much smaller than the block size. The step complexity is therefore about twice that of the sequential algorithm. On the other hand, algorithm 5 uses only $(3+22r/b)hw$ bandwidth. This is less than half of what is needed by the sequential algorithm. In fact, considering that the lower bound on the required amount of traffic to global memory is $2hw$ (assuming the entire image fits in shared memory), this is a remarkable result. It is not surprising that algorithm 5 is our fastest, at least in first-order filters (see section 8).

6 Overlapped summed-area tables

A summed-area table is obtained using prefix sums over columns and rows, and the prefix-sum filter S is a special case of first-order causal recursive filter (with feedback weight $a_1=-1$). We can therefore directly apply the idea of overlapping to optimize the computation of summed-area tables.

In blocked form, the problem is to obtain output V from input X where the blocks satisfy the relations

$$\begin{aligned} B_{m,n}(V) &= S^{\mathcal{T}}\big(P^{\mathcal{T}}_{m,n-1}(V), B_{m,n}(Y)\big) \quad \text{and} \\ B_{m,n}(Y) &= S\big(P_{m-1,n}(Y), B_{m,n}(X)\big). \end{aligned} \tag{41}$$

With the framework we developed, overlapping the computation of S and $S^{\mathcal{T}}$ is easy. In the first stage, we compute incomplete output blocks $B_{m,n}(\bar{Y})$ and $B_{m,n}(\hat{V})$ directly from the input:

$$B_{m,n}(\bar{Y}) = S\big(\mathbf{0}, B_{m,n}(X)\big) \quad \text{and} \tag{42}$$

$$B_{m,n}(\hat{V}) = S^{\mathcal{T}}\big(\mathbf{0}, B_{m,n}(\bar{Y})\big). \tag{43}$$

We store only the incomplete prologues $P_{m,n}(\bar{Y})$ and $P^{\mathcal{T}}_{m,n}(\hat{V})$. Then we complete them using:

$$P_{m,n}(Y) = P_{m-1,n}(Y) + P_{m,n}(\bar{Y}) \quad \text{and} \tag{44}$$

$$P^{\mathcal{T}}_{m,n}(V) = P^{\mathcal{T}}_{m,n-1}(V) + s\big(P_{m-1,n}(Y)\big) + P^{\mathcal{T}}_{m,n}(\hat{V}). \tag{45}$$

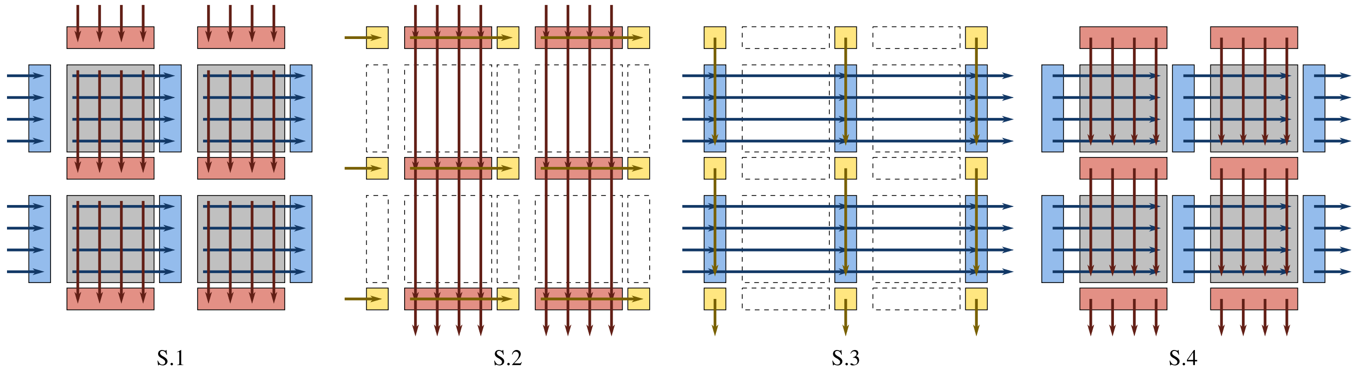


Figure 3: *Overlapped summed-area table computation according to algorithm SAT. Stage S.1 reads the input (in gray) then computes and stores incomplete prologues $P_{m,n}(\bar{Y})$ (in red) and $P^{\mathcal{T}}_{m,n}(\hat{V})$ (in blue). Stage S.2 completes prologues $P_{m,n}(Y)$ and computes scalars $s\big(P_{m-1,n}(Y)\big)$ (in yellow). Stage S.3 completes prologues $P^{\mathcal{T}}_{m,n}(V)$. Finally, stage S.4 reads the input and completed prologues, then computes and stores the final summed-area table.*

Scalar $s\big(P_{m-1,n}(Y)\big)$ in (45) denotes the sum of all entries in vector $P_{m-1,n}(Y)$. The simplified notation means the scalar should be added to *all* entries of $P^{\mathcal{T}}_{m,n}(V)$. The resulting algorithm is depicted in figure 3 and described below:

Algorithm SAT

- S.1 In parallel for all m and n, compute and store the $P_{m,n}(\bar{Y})$ and $P^{\mathcal{T}}_{m,n}(\hat{V})$ according to (42) and (43).
- S.2 Sequentially for each m, but in parallel for each n, compute and store the $P_{m,n}(Y)$ according to (44) and using the previously computed $P_{m,n}(\bar{Y})$. Compute and store $s\big(P_{m,n}(Y)\big)$.
- S.3 Sequentially for each n, but in parallel for each m, compute and store the $P^{\mathcal{T}}_{m,n}(V)$ according to (45) using the previously computed $P_{m-1,n}(Y)$, $P^{\mathcal{T}}_{m,n}(\hat{V})$ and $s\big(P_{m,n}(Y)\big)$.
- S.4 In parallel for all m and n, compute $B_{m,n}(Y)$ then compute and store $B_{m,n}(V)$ according to (41) and using the previously computed $P_{m,n}(Y)$ and $P^{\mathcal{T}}_{m,n}(V)$.

Our overlapped algorithm SAT uses only $(3 + 8/b + 2/b^2)hw$ of bandwidth and requires $O\big(\frac{hw}{cp}(4 + 3/b)\big)$ steps for completion.[2] In contrast, using a stock implementation of prefix scan to process rows, followed by matrix transposition, followed by column processing (as in [Harris et al. 2008]) requires at least $8hw$ bandwidth. Even our transposition-free, fused, bandwidth-saving variant of the multi-wave approach of Hensley [2010] requires at least $(4 + 9/b)hw$ bandwidth. As shown in section 8, the bandwidth reduction leads to the higher performance of the overlapped algorithm SAT.

7 Additional considerations

Feedforward in recursive filters A more general formulation for causal recursive filters would include an additional a_0 coefficient as well as a *feedforward* filter component with coefficients b_k:

$$\boldsymbol{y}_i = \frac{1}{a_0}\left(\sum_{k=0}^{q} b_k \boldsymbol{x}_{i-k} - \sum_{k=1}^{r} a_k \boldsymbol{y}_{i-k}\right). \tag{46}$$

The added complication due to the feedforward coefficients is small compared to the basic challenge of the feedback coefficients, because the feedforward computation does not involve any dependency chains. One implementation approach is to simply pre-convolve the input with the feedforward component and then apply the appropriate feedback-only filter. To reduce memory bandwidth, it is best to fuse the feedforward computation with the parallelized feedback filter. This is relatively straightforward.

Flexibility with higher-order filters Higher-order causal or anti-causal recursive filters can be implemented using different schemes: direct, cascaded, or in parallel. A direct implementation uses equations (1) or (3). Alternatively, we can decompose a higher-order filter into an equivalent set of first- and second-order filters, to be applied in series one after the other (i.e., cascaded) or independently and then added together (i.e., in parallel). See [Oppenheim and Schafer 2010, section 6.3] for an extensive discussion.

Although the different schemes are mathematically equivalent, in practice we may prefer one to the other due to bandwidth-to-FLOP ratio, implementation simplicity, opportunities for code reuse, and numerical precision. (Precision differences are less of a concern in floating-point [Oppenheim and Schafer 2010, section 6.9].)

Our overlapping strategy is compatible with all these schemes. The tools we provide allow the implementation of overlapped recursive filters of any order, in a way that maps well to modern GPUs. Overlapping can be used to directly implement a high-order filter, or alternatively to implement the first- and second-order filters used as building blocks for a cascaded or parallel realization.

Hierarchical reduction of prologues and epilogues In our algorithms, incomplete prologues and epilogues are completed sequentially (i.e., without inter-block parallelism). It is possible to perform such computations in a hierarchical fashion, thereby increasing parallelism and potentially speeding up the process. Completion equations such as (24) have the same structure as a recursive filter. This is clear when we consider the formulation in (52) of appendix A, where we process the input in groups of r elements at a time. Taking stage 5.2 as an example, we could group b prologues $P_{ib,n}(\bar{Y})$ to $P_{(i+1)b,n}(\bar{Y})$ into input blocks and recursively apply a variant of algorithm 1 to compute all $P_{m,n}(Y)$. Step 2 of this new computation is b times smaller still, consisting of only w/b^2 prologues. We can repeat the recursion until we reach a problem size small enough to be solved sequentially, the results of which can be propagated back up to complete the original filtering operation. We skipped this optimization in our work due to diminishing returns. For example, the amount of time spent by algorithm 5 on stages 5.2–5.5 for a 4096^2 image amounts to only about 17% of the total. This means that, even if we could run these stages instantaneously, performance would improve by only 20%.

Recursive doubling for intra-block parallelism In our image-processing applications, the processing of each two-dimensional block exposes sufficient parallelism to occupy all cores in each SM. When processing one-dimensional input with algorithms 1 or 3, however, it may be necessary to increase intra-block parallelism.

[2] Actually, $O\big(\frac{hw}{cp}(4 + (b+1)/b^2 + 2/b)\big)$, but $(b+1)/b \approx 1$.

Table 1: *Properties of the presented algorithms, for row and column processing of an $h \times w$ image with causal and anticausal recursive filters of order r, assuming block-size b, and p SMs with c cores each. For each algorithm, we show an estimate of the number of steps required, the maximum number of parallel independent threads, and the required memory bandwidth.*

Alg.	Step complexity	Max. # of threads	Bandwidth
RT	$\frac{hw}{cp}4r$	h, w	$8hw$
2	$\frac{hw}{cp}\left(8r + 4\frac{1}{b}(r^2+r)\right)$	$\frac{1}{b}hw$	$(9 + 16\frac{r}{b})hw$
4	$\frac{hw}{cp}\left(8r + 6\frac{1}{b}(r^2+r)\right)$	$\frac{1}{b}hw$	$(5 + 18\frac{r}{b})hw$
5	$\frac{hw}{cp}\left(8r + \frac{1}{b}(18r^2+10r)\right)$	$\frac{1}{b}hw$	$(3 + 22\frac{r}{b})hw$
SAT	$\frac{hw}{cp}(8 + \frac{3}{b})$	$\frac{1}{b}hw$	$(3 + \frac{8}{b} + \frac{2}{b^2})hw$

Recursive doubling [Stone 1973] is a well known strategy for first-order recursive filter parallelization we can use to perform intra-block computations. The idea maps well to GPU architectures, and is related to the tree-reduction optimization employed by efficient one-dimensional parallel scan algorithms [Sengupta et al. 2007; Dotsenko et al. 2008; Merrill and Grimshaw 2009]. Using a block size b that matches the number of processing cores c, the idea is to break the computation into steps in which each entry is modified by a different core. Using recursive doubling, computation of b elements completes in $O(\log_2 b)$ steps.

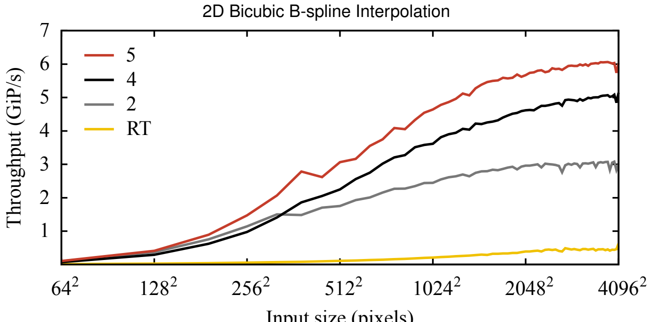
The extension of recursive doubling to higher-order recursive filters has been described by Kooge and Stone [1973]. The key idea is to group input and output elements into r-vectors and consider equation (1) in the matrix form of (52) in appendix A. Since the algebraic structure of this form is the same as that of a first-order filter, the same recursive doubling structure can be reused.

8 Results

Table 1 summarizes the main characteristics of all the algorithms that we evaluated, in terms of number of required steps, the progression in parallelism, and the reduction in memory bandwidth.

Our test hardware consisted of an NVIDIA GTX 480 with 1.5GB of RAM (480 CUDA cores, $p = 15$ SMs, $c = 32$ cores/SM). All algorithms were implemented in *C for CUDA*, under CUDA 4.0. All our experiments ran on *single-channel* 32-bit floating-point images. Image sizes ranged from 64^2 to 4096^2 pixels, in 64-pixel increments. Measurements were repeated 100 times to reduce variation. Note that small images are solved in sequence, *not* in batches that could be processed independently for added parallelism and performance.

First-order filters As an example of combined row-column causal-anticausal first-order filter, we solve the bicubic B-spline interpolation problem (see also figure 7(top)). Algorithm RT is the original implementation by Ruijters and Thévenaz [2010] (available on-line). Algorithm 2 adds blocking for memory coalescing, inter-block parallelism, and kernel fusion. (Algorithms 1 and 3 work on 1D input and are omitted from our tests.) Algorithm 4 employs overlapped causal-anticausal processing and fused row-column processing. Finally, algorithm 5 is fully overlapped. Performance numbers in figure 4(top) show the progression in throughput described throughout the text. As an example, the throughput of algorithm 5 solving the bicubic B-spline interpolation problem for 1024^2 images is 4.7GiP/s (gibi-pixels per second). Each image is transformed in just 0.21ms or equivalently at more than 4800fps. The algorithm appears to be compute-bound. Eliminating computation and keeping data movements, algorithm 5 attains 13GiP/s (152GB/s) on large images, whereas with computation it reaches 6GiP/s (72GB/s).



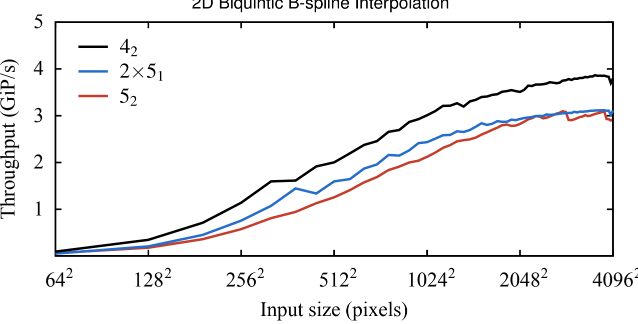


Figure 4: *Throughput of the various algorithms for row-column causal-anticausal recursive filtering. (Top plot) First-order filter (e.g. bicubic B-spline interpolation). (Bottom plot) Second-order filter (e.g. biquintic B-spline interpolation).*

Second-order filters The second-order, causal-anticausal, row-column separable recursive filter used in our tests solves the biquintic B-spline interpolation problem. Figure 4(bottom) compares three alternative structures: $2\times5_1$ is a cascaded implementation using two fused fully-overlapped passes of first-order algorithm 5, 4_2 is a direct second-order implementation of algorithm 4, and 5_2 is a direct fully-overlapped second-order implementation of algorithm 5. Our implementation of 4_2 is the fastest, despite using more bandwidth than 5_2. The higher complexity of second-order equations slows down stages 5.4 and 5.5 substantially. We believe this is an optimization issue that may be resolved with a future hardware, compiler, or implementation. Until then, the best alternative is to use the simpler and faster 4_2. It runs at 3.1GiP/s for 1024^2 images, processing each image in less than 0.32ms, or equivalently at more than 3200fps.

Precision A useful measure of numerical precision in the solution of a linear system such as the bicubic B-spline interpolation problem is the *relative residual*. Using random input images with entries in the interval $[0, 1]$, the relative residual was less than 2×10^{-7} for all algorithms and for all image sizes.

Summed-area tables Our overlapped summed-area table algorithm was compared with the algorithm of Harris et al. [2008] implemented in the CUDPP library [2011], and with the multi-wave method of Hensley [2010]. We also compare against a version of Hensley's method improved by two new optimizations: fusion of processing across rows and columns, and storage of just "carries" (e.g. $P_{m,n}(\bar{Y})$) between intermediate stages to reduce bandwidth. As expected, the results in figure 5 confirm that our specialized overlapped summed-area table algorithm outperforms the others.

Recursive Gaussian filters As mentioned in section 1, Gaussian filters of wide support are well approximated by recursive filters (see also figure 7(bottom)). To that end, we implemented the third-order

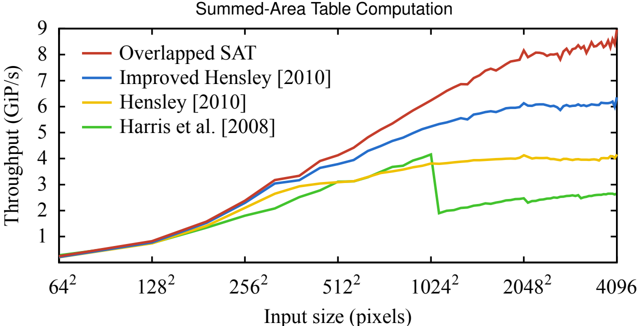


Figure 5: *Throughput of summed-area table computation, using our overlapped algorithm, the multi-wave algorithm of Hensley [2010], a revisited version of it, and the CUDPP code of Harris et al. [2008].*

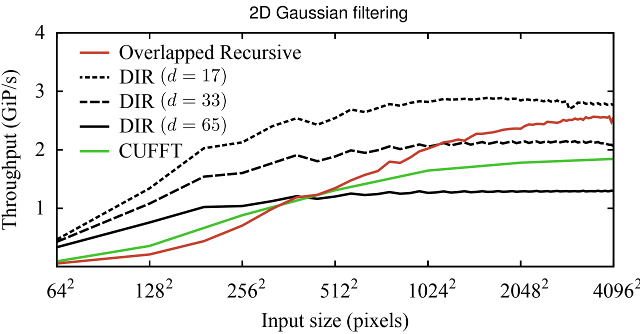


Figure 6: *Throughput of 2D Gaussian filtering, using our overlapped 3rd-order recursive implementation, direct convolution with d taps ($\sigma = d/6$), and the Fast Fourier Transform.*

approximation ($N = 3$) of van Vliet et al. [1998] with cascaded first- and second-order filters. The first-order component uses algorithm 5, and is fused with the second-order component, which uses algorithm 4_2. Figure 6 compares performance of the overlapped recursive implementation with convolution in the frequency domain using the CUFFT library [2007], and with highly optimized direct convolution (separable in shared memory) [Podlozhnyuk 2007]. Because computation in the direct convolution is proportional to the number of filter taps d, we show results for filter kernels with $d = 17, 33, 65$ taps. Although the direct approach is fastest for small filter windows, both the recursive filter and CUFFT are faster for larger windows. For reasonably sized images, overlapped recursive filtering is fastest. Moreover, its advantage over CUFFT should continue to increase on larger images due to its lower, linear-time complexity.

9 Conclusions

We describe an efficient algorithmic framework that reduces memory bandwidth over a sequence of recursive filters. It splits the input into blocks that are processed in parallel by modern GPU architectures, and overlaps the causal, anticausal, row, and column filter processing. The full sequence of filter passes requires reading the input image only twice, and writing only once. The reduced number of accesses to global memory leads to substantial performance gains. We demonstrate practicality in several scenarios: solving bicubic and biquintic B-spline interpolation problems, computing summed-area tables, and performing Gaussian filtering.

Future work Although we have focused on 2D processing, our framework of overlapping successive filters should extend to volumes (or even higher dimensions) using analogous derivations. However, one practical difficulty is the limited shared memory (e.g. 48k bytes) available in current GPUs. Making the volumetric blocks of size b^3 fit within shared memory would require significantly smaller block sizes, and this would decrease the efficiency of the algorithm.

Another area for future work is to adapt our algorithms to modern multicore CPUs (rather than GPUs). Although row and column parallelism is already sufficient for the fewer CPU threads, the memory bandwidth reduction may still be beneficial. In particular, our blocking structure may allow more accesses to be served by the faster L1 cache (or by the L2 cache in the case of volumetric processing). Whether this would lead to performance gains remains to be seen. The reduced bandwidth may also be beneficial on architectures like the Cell microprocessor in which memory transfers must be managed explicitly.

Finally, we would like to investigate whether the idea of overlapped computation can be used to optimize the solution of general narrow-band-diagonal linear systems.

10 Acknowledgements

This work has been funded in part by a post-doctoral scholarship from CNPq and by an INST grant from FAPERJ.

References

BLELLOCH, G. E. 1989. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11).

BLELLOCH, G. E. 1990. Prefix sums and their applications. Technical Report CMU-CS-90-190, Carnegie Mellon University.

BLU, T., THÉNAVAZ, P., and UNSER, M. 1999. Generalized interpolation: Higher quality at no additional cost. In *IEEE International Conference on Image Processing*, volume 3, 667–671.

BLU, T., THÉVENAZ, P., and UNSER, M. 2001. MOMS: Maximal-order interpolation of minimal support. *IEEE Transactions on Image Processing*, 10(7):1069–1080.

CROW, F. 1984. Summed-area tables for texture mapping. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, volume 18, ACM, 207–212.

CUDPP library. 2011. URL `http://code.google.com/p/cudpp/`.

CUFFT library. 2007. URL `http://developer.nvidia.com/cuda-toolkit`. NVIDIA Corporation.

DERICHE, R. 1992. Recursively implementing the Gaussian and its derivatives. In *Proceedings of the 2nd Conference on Image Processing*, 263–267.

DOTSENKO, Y., GOVINDARAJU, N. K., SLOAN, P.-P., BOYD, C., and MANFERDELLI, J. 2008. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, 205–213.

GÖDDEKE, D. and STRZODKA, R. 2011. Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):22–32.

HARRIS, M., SENGUPTA, S., and OWENS, J. D. 2008. Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*, chapter 39.

HENSLEY, J. 2010. Advanced rendering technique with DirectX 11: High-quality depth of field. Gamefest 2010 talk.

HENSLEY, J., SCHEUERMANN, T., COOMBE, G., SINGH, M., and LASTRA, A. 2005. Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24(3):547–555.

HOCKNEY, R. W. and JESSHOPE, C. R. 1981. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger.

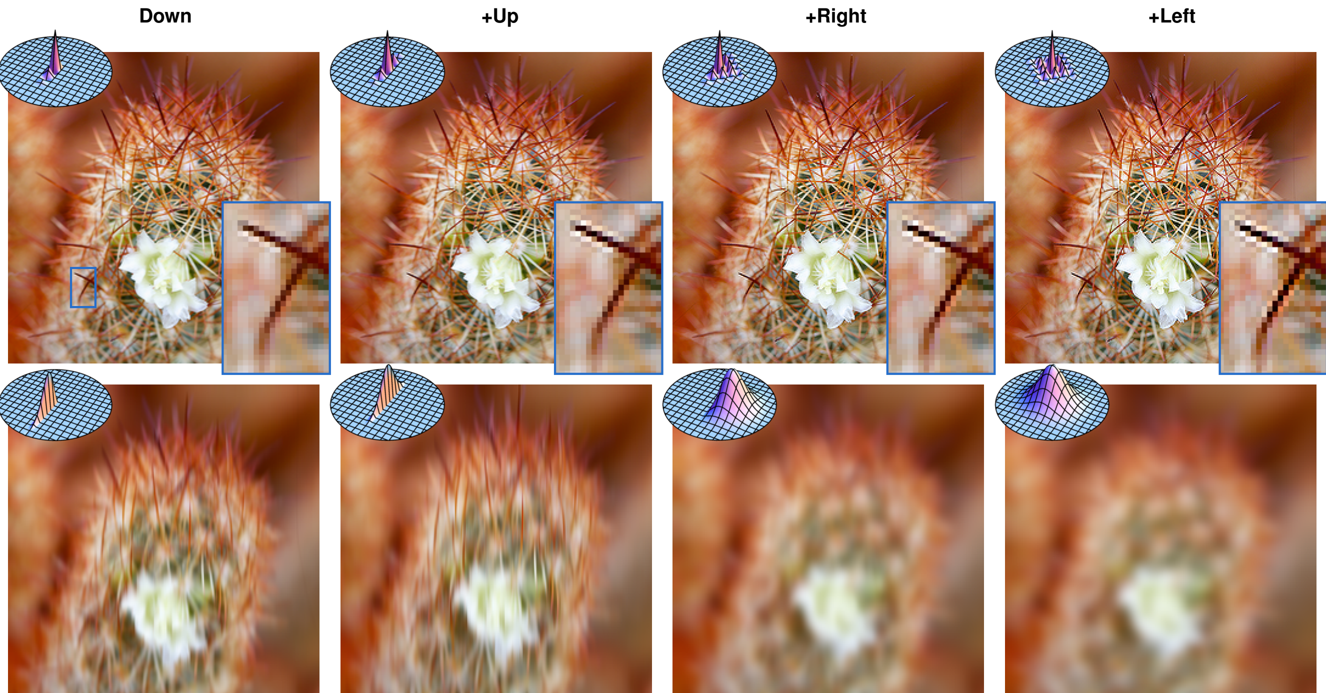


Figure 7: *Effect of successive recursive filtering passes on a test image. (Top row) Cubic B-Spline interpolation filter, which acts as a high-pass. (Bottom row) Gaussian filter with $\sigma = 5$, which acts as a low-pass. Insets show the combined filter impulse responses (not to scale). Our overlapped algorithm computes all four of these successive filtering passes using just two traversals of the input image.*

IVERSON, K. E. 1962. *A Programming Language*. Wiley.

KASS, M., LEFOHN, A., and OWENS, J. D. 2006. Interactive depth of field using simulated diffusion on a GPU. Technical Report #06-01, Pixar Animation Studios.

KIRK, D. B. and HWU, W. W. 2010. *Programming Massively Parallel Processors*. Morgan Kaufmann.

KOOGE, P. M. and STONE, H. S. 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793.

LAMAS-RODRÍGUES, J., HERAS, D. B., BÓO, M., and ARGÜELLO, F. 2011. Tridiagonal solvers internal report. Technical report, University of Santiago de Compostela.

MERRILL, D. and GRIMSHAW, A. 2009. Parallel scan for stream architectures. Technical Report CS2009-14, University of Virginia.

OPPENHEIM, A. V. and SCHAFER, R. W. 2010. *Discrete-Time Signal Processing*. Prentice Hall, 3rd edition.

PARHI, K. and MESSERSCHMITT, D. 1989. Pipeline interleaving and parallelism in recursive digital filters–Part II: Pipelined incremental block filtering. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(7):1099–1117.

PODLOZHNYUK, V. 2007. Image convolution with CUDA. NVIDIA whitepaper.

RUIJTERS, D. and THÉVENAZ, P. 2010. GPU prefilter for accurate cubic B-spline interpolation. *The Computer Journal*.

SENGUPTA, S., HARRIS, M., ZHANG, Y., and OWENS, J. D. 2007. Scan primitives for GPU computing. In *Proceedings of Graphics Hardware*, 97–106.

STONE, H. S. 1971. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–161.

STONE, H. S. 1973. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM*, 20(1):27–38.

SUNG, W. and MITRA, S. 1986. Efficient multi-processor implementation of recursive digital filters. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 257–260.

SUNG, W. and MITRA, S. 1992. Multiprocessor implementation of digital filtering algorithms using a parallel block processing method. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):110–120.

VAN VLIET, L. J., YOUNG, I. T., and VERBEEK, P. W. 1998. Recursive Gaussian derivative filters. In *Proceedings of the 14th International Conference on Pattern Recognition*, 509–514 (v. 1).

ZHANG, Y., COHEN, J., and OWENS, J. D. 2010. Fast tridiagonal solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

A Derivation of $T(A_{FP}) = A_F^b$

In this appendix we derive the $r \times r$-matrix $T(A_{FP})$ that is used to transfer a prologue vector across a zero-valued block in the recursive filter computation of (24).

When dealing with a higher-order recursive filter F, it is convenient to group input and output elements into r-vectors, so that F is expressed in a simplified matrix form. To that end, let

$$\dot{\boldsymbol{y}}_i = \begin{pmatrix} y_{i-r} \\ \vdots \\ y_{i-2} \\ y_{i-1} \end{pmatrix}, \ \dot{\boldsymbol{x}}_i = \begin{pmatrix} x_{i-r} \\ \vdots \\ x_{i-2} \\ x_{i-1} \end{pmatrix} \text{ and } \ddot{\boldsymbol{x}}_i = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ x_{i-1} \end{pmatrix}. \tag{47}$$

It is easy to verify that if

$$\dot{\boldsymbol{y}}_0 = \boldsymbol{p}, \quad \text{then} \qquad (48)$$
$$\dot{\boldsymbol{y}}_i = \ddot{\boldsymbol{x}}_i + A_F \dot{\boldsymbol{y}}_{i-1}, \quad \text{where}$$

$$A_F = \begin{pmatrix} 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \\ -a_r & -a_{r-1} & \cdots & -a_2 & -a_1 \end{pmatrix}. \qquad (49)$$

A nicer expression involves $\dot{\boldsymbol{x}}_i$ rather than $\ddot{\boldsymbol{x}}_i$. Iterating (48):

$$\dot{\boldsymbol{y}}_i = \ddot{\boldsymbol{x}}_i + A_F \dot{\boldsymbol{y}}_{i-1} = \ddot{\boldsymbol{x}}_i + A_F \ddot{\boldsymbol{x}}_{i-1} + A_F^2 \dot{\boldsymbol{y}}_{i-2} = \cdots \qquad (50)$$
$$= \left(\sum_{k=0}^{r-1} A_F^k \ddot{\boldsymbol{x}}_{i-k} \right) + A_F^r \dot{\boldsymbol{y}}_{i-r} \qquad (51)$$
$$= \bar{A}_F \dot{\boldsymbol{x}}_i + A_F^r \dot{\boldsymbol{y}}_{i-r}. \qquad (52)$$

Column k of matrix $\bar{A}_F$ is simply the last column of A_F^{r-k-1}, for $k \in \{0, 1, \ldots, r-1\}$. Under this formulation, r new output elements are produced and r elements of input are consumed. For example, given a second-order filter:

$$A_F = \begin{pmatrix} 0 & 1 \\ -a_2 & -a_1 \end{pmatrix}, \qquad (53)$$

$$\bar{A}_F = \begin{pmatrix} 1 & 0 \\ -a_1 & 1 \end{pmatrix}, \quad \text{and} \quad A_F^2 = \begin{pmatrix} -a_2 & -a_1 \\ a_1 a_2 & a_1^2 - a_2 \end{pmatrix} \qquad (54)$$

Equivalent derivations appear in the works of Kooge and Stone [1973] and Blelloch [1990].

Anticausal filters can also be expressed in terms of r-vectors in a form similar to (52), with corresponding matrices $\bar{A}_R$ and A_R.

Recall that

$$T(F(\boldsymbol{p}, \mathbf{0})) = T(F(I_r, \mathbf{0})\, \boldsymbol{p}) = T(A_{FP}\, \boldsymbol{p}) = T(A_{FP})\, \boldsymbol{p}. \qquad (55)$$

When $\boldsymbol{x}_i$ is null, equations (48) reduce to

$$\dot{\boldsymbol{y}}_0 = \boldsymbol{p}, \qquad \dot{\boldsymbol{y}}_i = A_F \dot{\boldsymbol{y}}_{i-1} = A_F^i\, \boldsymbol{p}. \qquad (56)$$

Since the last r output elements of the b-wide block are given by

$$T(F(\boldsymbol{p}, \mathbf{0})) = \dot{\boldsymbol{y}}_b, \qquad (57)$$

we have

$$T(A_{FP})\, \boldsymbol{p} = A_F^b\, \boldsymbol{p} \quad \Rightarrow \quad T(A_{FP}) = A_F^b. \qquad (58)$$

B Derivation of full overlapping

Consider the transposed prologues of U. From the definition and repeated application of properties (9) and (10):

$$\begin{aligned} P_{m,n}^{\mathcal{T}}(U) = {} & T^{\mathcal{T}}\big(F^{\mathcal{T}}(P_{m,n-1}^{\mathcal{T}}(U), \mathbf{0})\big) \\ & + T^{\mathcal{T}}\big(F^{\mathcal{T}}(\mathbf{0}, R(\mathbf{0}, E_{m+1,n}(Z)))\big) \\ & + T^{\mathcal{T}}\big(F^{\mathcal{T}}(\mathbf{0}, R(F(P_{m-1,n}(Y), \mathbf{0}), \mathbf{0}))\big) \\ & + T^{\mathcal{T}}\big(F^{\mathcal{T}}(\mathbf{0}, R(F(\mathbf{0}, B_{m,n}(X)), \mathbf{0}))\big). \end{aligned} \qquad (59)$$

Now applying (11), (13) and (14):

$$\begin{aligned} P_{m,n}^{\mathcal{T}}(U) = {} & P_{m,n-1}^{\mathcal{T}}(U)\, T^{\mathcal{T}}\big(F^{\mathcal{T}}(I_r, \mathbf{0})\big) \\ & + R(\mathbf{0}, I_r)\, E_{m+1,n}(Z)\, T^{\mathcal{T}}\big(F^{\mathcal{T}}(\mathbf{0}, I_b)\big) \\ & + R(I_b, \mathbf{0})\, F(I_r, \mathbf{0})\, P_{m-1,n}(Y)\, T^{\mathcal{T}}\big(F^{\mathcal{T}}(\mathbf{0}, I_b)\big) \\ & + T^{\mathcal{T}}\big(F^{\mathcal{T}}(\mathbf{0}, R(F(\mathbf{0}, B_{m,n}(X)), \mathbf{0}))\big) \end{aligned} \qquad (60)$$

$$\begin{aligned} = {} & P_{m,n-1}^{\mathcal{T}}(U)\, \big(A_F^b\big)^T \\ & + A_{RE}\, \big(E_{m+1,n}(Z)\, \big(T(A_{FB})\big)^T\big) \\ & + (A_{RB}\, A_{FP})\, \big(P_{m-1,n}(Y)\, \big(T(A_{FB})\big)^T\big) \\ & + P_{m,n}^{\mathcal{T}}(\check{U}), \end{aligned} \qquad (61)$$

where $\check{U}$ is such that

$$B_{m,n}(\check{U}) = F^{\mathcal{T}}\big(\mathbf{0}, R\big(F(\mathbf{0}, B_{m,n}(X)), \mathbf{0}\big)\big) \qquad (62)$$
$$= F^{\mathcal{T}}\big(\mathbf{0}, B_{m,n}(\hat{Z})\big). \qquad (63)$$

In (61) we have exploited matrix-product associativity to select the best computation ordering.

Following a similar derivation for the transposed epilogues of V:

$$\begin{aligned} E_{m,n}^{\mathcal{T}}(V) = {} & H^{\mathcal{T}}\big(R^{\mathcal{T}}(\mathbf{0}, E_{m,n+1}^{\mathcal{T}}(V))\big) \\ & + H^{\mathcal{T}}\big(R^{\mathcal{T}}(F^{\mathcal{T}}(P_{m,n-1}^{\mathcal{T}}(U), \mathbf{0}), \mathbf{0})\big) \\ & + H^{\mathcal{T}}\big(R^{\mathcal{T}}(F^{\mathcal{T}}(\mathbf{0}, R(\mathbf{0}, E_{m+1,n}(Z))), \mathbf{0})\big) \\ & + H^{\mathcal{T}}\big(R^{\mathcal{T}}(F^{\mathcal{T}}(\mathbf{0}, R(F(P_{m-1,n}(Y), \mathbf{0}), \mathbf{0})), \mathbf{0})\big) \\ & + H^{\mathcal{T}}\big(R^{\mathcal{T}}(F^{\mathcal{T}}(\mathbf{0}, R(F(\mathbf{0}, B_{m,n}(X)), \mathbf{0})), \mathbf{0})\big) \end{aligned} \qquad (64)$$

$$\begin{aligned} = {} & E_{m,n+1}^{\mathcal{T}}(V)\, \big(A_R^b\big)^T \\ & + P_{m,n-1}^{\mathcal{T}}(U)\, \big(H(A_{RB})\, A_{FP}\big)^T \\ & + A_{RE}\, \big(E_{m+1,n}(Z)\, \big(H(A_{RB})\, A_{FB}\big)^T\big) \\ & + (A_{RB}\, A_{FP})\big(P_{m-1,n}(Y)\, \big(H(A_{RB})\, A_{FB}\big)^T\big) \\ & + E_{m,n}^{\mathcal{T}}(\tilde{V}), \end{aligned} \qquad (65)$$

where

$$B_{m,n}(\tilde{V}) = R^{\mathcal{T}}\big(F^{\mathcal{T}}(\mathbf{0}, R(F(\mathbf{0}, B_{m,n}(X)), \mathbf{0})), \mathbf{0}\big) \qquad (66)$$
$$= R^{\mathcal{T}}\big(B_{m,n}(\check{U}), \mathbf{0}\big). \qquad (67)$$