Performance and error analysis of recursive edge-aware Gaussian filters on GPUs

Hermes H. Ferreira

Institute of Mathematics and Statistics
UFRGS
Porto Alegre, Brazil
hermes.hofmeister@ufrgs.br

Eduardo S. L. Gastal

Institute of Informatics
UFRGS
Porto Alegre, Brazil
eslgastal@inf.ufrgs.br

Lucas M. Schnorr

Institute of Informatics
UFRGS
Porto Alegre, Brazil
schnorr@inf.ufrgs.br

Philippe O. A. Navaux

Institute of Informatics
UFRGS
Porto Alegre, Brazil
navaux@inf.ufrgs.br

Abstract—We present a schematic for image edge-aware Gaussian GPU filtering which has linear complexity on the number of pixels of the image. It allows us to reduce the execution time as we increase the number of Streaming Multiprocessors (SMs) on the GPU. We make use of a domain transformation and use a complex-valued recursive formulation of the Gaussian filter. The algorithm partitions the image in disjoint regions, where we compute in parallel the filtering operations, avoiding communication between regions. Our implementation leads to a real-time solution using a modern GPU. With the RTX 2080 Ti, we achieved an execution time of less than 10 milliseconds for 2 filtering iterations on high-resolution RGB images of dimensions 2048×2048.

I. INTRODUCTION

Digital filtering is a fundamental operation on image processing. When implemented naively, a digital filter can be costly since it might involve 2D convolution with quadratic worst-case $O(N^2)$ complexity. The problem becomes more sophisticated when we also desire filtering to preserve edges on the image (Figure 1). It is advantageous to reduce the convolution operation to computing a linear recursive equation as this is guaranteed to have $O(N)$ complexity on the number of pixels. In this paper, we do so by making use of the domain transform algorithm [4]. We restrict ourselves to Gaussian edge-aware filtering, introduced as the bilateral filter [11], which is a popular filter that has often been an object of study. For example, Jiawen et al. [1] and Qingxiong et al. [12] present techniques to implement the bilateral filter that only handle grayscale images. We use the more general techniques presented by Gastal and Oliveira [5].

In this paper, we introduce an algorithm designed for edge-aware recursive GPU filtering in linear time $O(N)$. We analyse its behavior in a series of experiments considering both performance and error bounds. The main contribution is that the algorithm developed in this work is easy to implement, makes efficient use of the GPU resources, and has its numerical efficiency analytically justified. For instance, by using our approximation scheme we achieved a speedup of 100% for filtering 2048×2048 images on the RTX 2080 Ti due to better resource utilization.

We point out that Nehab et al. [8] present an algorithm designed to efficiently compute recursive filters on the GPU. It requires communication between neighbouring blocks but requires only two image reads and filters in every direction in

Fig. 1: (Left) Original photograph with dimensions 768×1024. (Right) Output of the edge-aware Gaussian filter with $\sigma = 50$ and $\sigma_r = 51$, computed using one of the algorithms proposed in this paper (Algorithm A). Note how small details of the image are smoothed but important edges are preserved.

only one kernel. As described in [8], the algorithm is designed for spatially-invariant filtering and relies on representing the filter on each block by a single matrix. As such, their algorithm cannot be directly applied to edge-aware recursive filters, since the edge-aware transform removes the spatial invariability of the filter (which can be seen as a non-uniform sampling). Thus, to the best of our knowledge, our work describes the first parallel GPU implementation of an edge-aware recursive Gaussian filter.

In Section II we provide the basic concepts on edge-aware filtering and GPU programming. In Section III we present a theoretical analysis of filtering approximation errors. Section IV describes first our standard approach to filtering, and then we state our proposal to better scale this approach in more powerful GPUs. In Section V we state what are the relevant variables to the experiments and show our experimental results for both quality of filtering and processing time in two GPU cards: GTX 1050 Ti and the RTX 2080 Ti. We provide source code for all of our algorithms and experiments in the supplementary materials.

II. BACKGROUND AND BASIC CONCEPTS

Intuitively [4], an RGB image can be considered as a bidimensional surface residing within a 5-dimensional space. Each pair $p = (x, y) \in \Omega$ has its RGB value $I(p) = (R(p), G(p), B(p))$, and so the image surface is given by $\hat{I} : \Omega \to \mathbb{R}^5$ where $\hat{I}(p) := (p, I(p))$. An edge-aware filter is such that the amount of information exchanged between two pixels (points on the surface) is inversely proportional to their distance in $\mathbb{R}^5$. This filter requires a choice of a metric for the space $\mathbb{R}^5$, we shall use the geodesic distance (other metrics are also viable).

Instead of working with the 5-dimensional underlying space, we would rather flatten this surface into $\mathbb{R}^2$ while preserving the distance between points. Unfortunately, this is not generally possible (for more details see [3]). It is, however, possible to isometrically flatten a curve within the surface because this is a one-dimensional object. Since linear recursive filters are computationally more efficient and are defined for one-dimensional signals, once we do such transformation we have substantial performance gains. The idea is to flatten first the rows of the image and apply recursive filtering, and then flatten the columns and filter them recursively. It should be noted, however, that the filtering must include more vertical and horizontal passes. This is required because a 2D edge-aware filter is not separable, as it cannot be written as a composition of two 1D edge-aware filters.

A. Expressions for the domain transform and one-dimensional Gaussian edge-aware filtering

Each pixel in a particular flattened row (or column) of the image is given by $f[k]$ where $k \in \{0, 1, .., L\}$ are the sample's indices. In particular, with RGB images the signal is of the form $f[k] = (f_1[k], f_2[k], f_3[k])$, with a number of $d = 3$ channels. In the isometrically-flattened curve (the transformed domain), the new sample positions are given by [4]:

$$t(k) = \sum_{i=0}^{k} \sqrt{1 + \frac{\sigma_s^2}{\sigma_r^2} \sum_{c=0}^{d} (f_c[i] - f_c[i-1])^2} \,. \quad (1)$$

The new signal is essentially the same, although with the underlying abstraction that the samples have non-uniform spacing given by the values $t(k) - t(k-1)$. Here σ_s controls how the mixing occurs based on the pixel distance in the 2D image space, and σ_r controls it based on the color distances in the 3D RGB space. We will be interested in the values

$$\Delta_{t_k} := t(k) - t(k-1) = \sqrt{1 + \frac{\sigma_s^2}{\sigma_r^2} \sum_{c=0}^{d} (f_c[k] - f_c[k-1])^2} \,. \quad (2)$$

In this work, we will implement a Gaussian filter in the transformed domain (described by the non-uniform spacings Δ_{t_k}). Deriche [2] gives the following approximation for the positive region $(x > 0)$ of the Gaussian distribution with deviation σ and unit-height (unnormalized):

$$u^+(x) = \mathrm{Re} \left\{ \alpha_0 \exp \left(-\frac{\lambda_0}{\sigma} x \right) + \alpha_1 \exp \left(-\frac{\lambda_1}{\sigma} x \right) \right\}. \quad (3)$$

Choose $\sigma = \sigma_s$. Using the procedure for computing a recursive filter in a non-uniform domain described in [5], one obtains the filter $g[k] = \sum_{i=0}^{1} \mathrm{Re} \left\{ g_i^+[k] + g_i^-[k] \right\}$, which has a causal component and an anticausal component (one that propagates forwards, and one that propagates backwards):

$$g_i^+[k] = a_i f[k] + b_i^{\Delta_{t_k}} g_i^+[k-1] + \Phi_{k-1,k}^i(\Delta_{t_k}), \quad (4)$$

$$g_i^-[k] = a_i b_i^{\Delta_{t_{k+1}}} f[k+1] + b_i^{\Delta_{t_{k+1}}} g_i^+[k+1] + \Phi_{k+1,k}^i(\Delta_{t_{k+1}}),$$

$$\Phi_{j,k}^i(\delta) = \left(\frac{b_i^\delta - 1}{r_{i,0}\delta} - r_{i,1} b_i \right) f[k] - \left(\frac{b_i^\delta - 1}{r_{i,0}\delta} - r_{i,1} b_i^\delta \right) f[j].$$

The constants that should be replaced into the above equations in order to implement an edge-aware Gaussian filter, extracted from Eq. (3), are given in the appendix. To filter a 2D image, we filter in parallel the causal and anticausal components of each horizontal/vertical pass. We compute first the horizontal pass and then the vertical pass on the output. (It is also possible to interchange the vertical filtering with the horizontal filtering). This sequential vertical/horizontal filtering is best used with low-pass filters, such as the Gaussian filter.

B. On GPU architecture

Modern GPUs are composed of Streaming Multiprocessors (SMs), equipped with computational cores. The GTX 1050 Ti used in our experiments has 6 SMs, each with 128 cores.

In an SM, every 32 cores are grouped into a so-called *warp*. Within the same warp, cores execute the same instruction at the same time. This means that diverging the execution patterns of cores within the same warp might significantly reduce performance.

When launching a program on the GPU, also known as a *kernel*, we assign it two values: the number of *blocks* it will execute and the number of threads executed per block. Blocks are assigned to SMs and threads are assigned to cores within the SM. It is possible to assign more blocks than SMs available and more threads than cores, and this might even increase performance in some cases, since it gives more leeway to the job scheduler.

Each SM has fast access to on-chip shared memory. Because it is faster, we desire to make more use of it when performing the filtering operations. The GPU memory hierarchy also includes the device global memory, which can be accessed by every SM, and provides superior bandwidth vs. accessing host (CPU) memory.

To access global memory, the best approach is to extract consecutive addresses of 4 bytes, since this can be done in a single instruction by a warp. This means that if V is a vector of 4-byte values in global memory, a warp should access in the pattern: $V[\text{offset} + \text{threadId}] \xrightarrow{\text{copy to}}$ on-chip shared memory.

Above, threadId is the index of the core in the SM card, in our scenario, this value is an integer between 0 and 127 since the SMs in the GTX 1050 Ti have 128 cores each. The use of shared memory imposes a practical challenge, as it might introduce *bank conflicts* – that is, access patterns that become
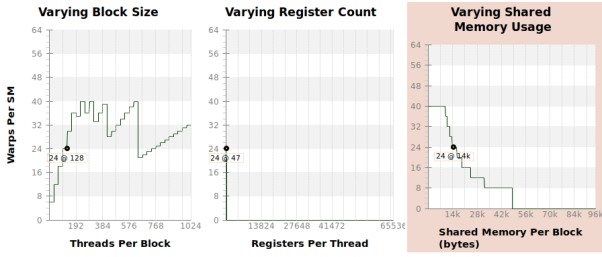
Fig. 2: Number of simultaneous warps available for execution in the SM of the GTX 1050 Ti as a function of: the block size (left), the number of registers used in the execution (center), and the employed amount of shared memory (right). These graphs were obtained through Nvidia's visual profiler for our horizontal causal filtering kernel.

serialized and thus do not fully utilize the bandwidth of the shared memory.

Shared memory is divided into separate banks in a round-robin fashion as illustrated below:
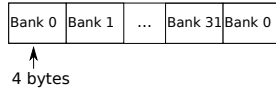
Therefore, when accessing shared memory, we should take caution that two different threads in the same warp are not accessing the same banks at the same time, as this is a conflict. If we are striding the shared memory by a multiple of 32, this is not a problem, since each thread will access only one bank. If a thread is accessing consecutive addresses in shared memory, it will access more than one bank. In this scenario, it compensates to introduce padding, as it can potentially reduce conflicts. For example, if S is a shared memory vector of consecutive 4-byte values we could access in the pattern $S[\text{threadId} \cdot (\text{width} + \text{padsize}) + j]$ while varying j. In this case, the padsize should be chosen so that width + padsize becomes a prime number. In some cases, this reduces the conflicts to zero. One exception to this discussion is that, when all threads within a warp require access to a single value, it can be broadcasted in a single instruction.

Finally, it is important to note that each thread has access to a maximum number of registers and that making use of an excessive number of these registers might reduce performance (Figure 2). A good basic reference for GPU programming is the book by Sanders and Kandrot [10].

III. APPROXIMATION SCHEME AND THEORETICAL RESULTS

In this section we present our chosen approximation scheme (used to parallelize the computation into independent blocks) and its theoretical implication on filtering quality.

We partition the image into identical disjoint blocks of pixels and compute the recursive causal and anticausal filtering components in each of these blocks in parallel.

Suppose a one-dimensional block for simplicity, denoted as $\{f[0], f[1], ..., f[B]\}$ where B is the block length. Our causal filter is of the form

$$g[k] = af[k] + b^{\Delta_{t_k}} g[k-1] + \Phi_{k-1,k}(\Delta_{t_k}). \quad (5)$$

Here the indices are local to the block partitions, so that $f[-1]$ might be a value inside the image, but outside the current block. To compute $g[0]$ (the first output value of the block), one would need to know the output $g[-1]$ of the previous block. This creates a data dependency between blocks, severely impacting performance and how much the algorithm can be parallelized. To avoid this, our idea is to extrapolate the block to obtain an approximation to the value $g[0]$ without depending on the previous block's output. We approximate by considering an extended block $\{f[-L], ..., f[0], ..., f[B]\}$, for some L, and then computing the appropriate filter's boundary conditions (which take into account its edge-aware characteristics). We should give a reasonable value of L such as to not compromise the performance of the algorithm. In the next subsections we will analyse the error given by this block partition.

A. Theoretical error analysis

We can describe the causal filter by the general equation

$$g_n = u_n f_n + v_{n-1} f_{n-1} + w_{n-1} g_{n-1}, \quad (6)$$

where the input is given by f. Unrolling the recurrence we obtain the expression for the error $\varepsilon(i,n)$ in g_n produced by assuming $g_{n-i} = 0$, for a given i: $\varepsilon_\sigma(i,n) = \left(\prod_{j=1}^{i} w_{n-j}\right) g_{n-i}$. As expected, the error depends on the true value of g_{n-i}, and also on the weights $w_{n-i}, \ldots, w_{n-1}$. Using that $w_{n-j} = b^{\Delta_{n-j+1}}$ (see Eq. (5)) we obtain

$$\varepsilon_\sigma(i,n) = b^{\left(\sum_{j=0}^{i-1} \Delta_{n-j}\right)} = b^{d(t_n, t_{n-i})} \cdot g_{n-i}, \quad (7)$$

where $d(t_i, t_j)$ is the distance between the i-th and j-th samples in the transformed (non-uniform) domain. Now using that $b = e^{\frac{-\lambda}{\sigma}}$ we conclude $\varepsilon_\sigma(i,n) = e^{\frac{-\lambda}{\sigma} d(t_n, t_{n-i})} \cdot g_{n-i}$.

Since the filter is normalized and the 8-bit image colors are integers between 0 and 255, we have $|g_{n-i}| \leq 255$. Therefore, taking the smallest of the lambdas listed in the Appendix (which generates the worst-case error), we obtain $|\varepsilon_\sigma(i,n)| \leq 255 \cdot e^{\frac{-1.72}{\sigma} d(t_n, t_{n-i})}$. So by extending the block from 0 to $-L$ the error at index n of the output is bounded by

$$|\varepsilon_\sigma(n+L, n)| \leq 255 \cdot e^{\frac{-1.72}{\sigma}(d(t_n, t_0) + d(t_0, t_{-L}))}. \quad (8)$$

Choosing L such that $d(t_0, t_{-L}) = \sigma$ we obtain

$$|\varepsilon_\sigma(n+L, n)| \leq 255 \cdot e^{-1.72} \cdot e^{\frac{-1.72}{\sigma} \cdot n} \leq 46 \cdot e^{\frac{-1.72}{\sigma} \cdot n}. \quad (9)$$

This is a crude estimate which we can improve upon by making assumptions on the initial conditions of the block. In the next section, we will explore possible heuristics for this situation.

B. Estimating the error by using non-zero initial conditions

We explore the case where we approximate g_{-L} instead of assuming $g_{-L} = 0$. We do this by assuming instead that the *input* is constant outside the extended block.

Since we are computing a Gaussian filter in a transformed domain (Eq. (1)), the final filtered value (given by the sum of the subfilters $\{g_0^+, g_1^+\}$ defined by Eq. (4)), is

$$g(-L) = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{-L_t} f(t^{-1}(s)) \, e^{-\frac{(-L_t - s)^2}{2\sigma^2}} \, ds \, . \quad (10)$$

where $-L_t := t^{-1}(-L)$ is a shorthand notation. We now approximate $g(-L)$ by assuming f is constant below $t^{-1}(-L)$. That is, $f(s) = \beta$ for all $s < t^{-1}(L)$. The approximated value is given by $\tilde{g}(-L)$ where:

$$\tilde{g}(-L) = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{-L_t} \beta \cdot e^{-\frac{(-L_t - s)^2}{2\sigma^2}} \, ds = \frac{\beta}{2} \, . \quad (11)$$

As such, the filtering error at the n-th output sample of the block is given by

$$|\varepsilon_\sigma(n + L, n)| \le e^{\frac{-1.72}{\sigma}(d(t_n, t_0) + d(t_0, t_{-L}))} \cdot E(-L) \quad (12)$$

where

$$\begin{aligned} E(-L) &= |\tilde{g}(-L) - g(-L)| \\ &= \left| \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{-L_t} (f(t^{-1}(s)) - \beta) \, e^{-\frac{(-L-s)^2}{2\sigma^2}} \, ds \right|. \end{aligned} \quad (13)$$

Compared to Eq. (8), Eq. (12) is an improved error bound which can guide our selection of the approximation constant β.

Recall that 95% of the mass of the Gaussian in the integral of Eq. (13) resides within two standard deviations of its mean $\mu = -L_t$. This indicates that choosing β to be as similar as possible to the neighbouring samples of $f(-L)$ should reduce the error $E(-L)$, in the process also reducing the filtering error ε_σ for all output samples. Note that this error decays at least exponentially within the block, since neighbouring samples are separated by at least a distance of $\Delta = 1$ transformed-domain units (see Eq. (2)). Most commonly for edge-aware filtering, $\Delta \gg 1$, and the error decays even faster.

C. Selecting approximation parameters to minimize the error

Eqs. (12–13) show that we can decrease the filtering error by *(i)* increasing the block width extension L until $d(t_0, t_L)$ is reasonably large and *(ii) choosing a good value for β.*

A good choice for L is the value that makes $d(t_0, t_L) \ge 2\sigma$. This guarantees a worst-case error $|\varepsilon_\sigma|$ of 9 RGB units, which is just 3.5% of the full 8-bit range $[0, 255]$. Furthermore, if one makes a proper selection of β such that $E(-L) < 30$, the error $|\varepsilon_\sigma|$ becomes smaller than 1 unit. This is a negligible error since it will be rounded-off when quantized to 8-bit pixels.

A good choice for β is $\beta = f[-L]$. This is the best possible choice that can be made considering that each block does not have direct access to input samples $f[n]$ with $n < -L$. This choice results in small filtering errors for two reasons: first, if the image has strong color variations around $f[-L]$, then the distance $d(t_0, t_L)$ is expected to be very large (since

the spacings in Eq. (2) are directly proportional to color variations). This minimizes the error regardless of β, due to the exponentials in Eq. (12). Second, if the image has low color variability around $f[-L]$, then by definition $\beta = f[-L]$ is a good representative value for the colors in this region. As such, $E(-L)$ in Eq. (13) is expected to be small, also reducing the filtering error.

D. Theoretical global error

Suppose the image is fully divided in blocks of $w \times h$ pixels, and that we are performing horizontal filtering (thus, the 1D block sizes are $B = w$). Assume that we extend each block by L samples, such that $d(t_0, t_L) = \kappa \cdot \sigma$ for a fixed κ. Given an upper bound M for the errors $E(-L)$ of all blocks, one can derive the worst-case mean squared error bound when comparing the exact filtered image with the approximated filtered image as:

$$\text{MSE}(w, \kappa, \sigma) \le \frac{M^2 \cdot e^{-\kappa \cdot 3.44}}{w} \cdot \frac{1 - e^{\frac{-3.44 \cdot w}{\sigma}}}{1 - e^{\frac{-3.44}{\sigma}}}. \quad (14)$$

Note that in practice M depends on σ and σ_r, and also on individual features of the image's content (in particular, on the ocurrence of high-contrast edges). However, this expression indicates that such features become less relevant as κ increases. Features of the image will affect the error at most quadratically, while κ reduces it exponentially.

IV. METHODS AND TOOLS

We begin by stating the approach to which we compare our results in the following sections.

A. Standard approach

In the GPU we obtain the best results by reading the image block into the shared memory of an SM. The author has experimented with a GTX 1050 Ti GPU which is equipped with 6 SM cards with 48kb of shared memory each with 128 processing cores. To achieve the best performance we ought to choose to filter the image in blocks with height and width such that we achieve as many active warps as possible while allowing to read the image with most of the available cores.

For filtering in horizontal direction, we chose dimensions $h = 128$ and $w = 24$. Further increasing the width reduced the performance, this is due to the hardware as seen in Figure 2.

We will restrict ourselves to discussing the implementation of the horizontal filtering. The case for vertical filtering is analogous. We present the straightforward approach to filtering on GPU when assigning each SM a horizontal stripe of height h allows using all of the SM cards available.

Algorithm A (Figure 3):

- Horizontally partition the image in stripes of height h;
- Each SM processes one horizontal stripe, starting with a window of dimensions $w \times h$ on the left side of the stripe;
- Store the image window inside shared memory, perform the causal filtering operations and write to the output. Move the window to the right by a distance of w. Repeat until the window reaches the right side of the image;
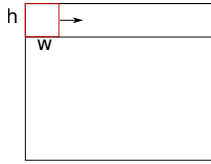
Fig. 3: Illustration of the subdivision used in Algorithm A.

Fig. 4: Example of domain division with 9 SMs. With more SMs, one could place more blocks per stripe similarly.

- Now, traveling the window to the left side of the image, analogously compute the anticausal filtering;
- After the horizontal filtering is done, perform the vertical filtering operations, this time with vertical stripes, and choosing block dimensions accordingly (in our case, $w = 128$ and $h = 24$).

In this standard approach, since each SM is assigned a whole stripe, with a fixed image size we have bounded performance that cannot be improved by increasing the number of available SM cards (the stripes sizes are fixed, and tied to the available shared memory in the SMs). We will address this issue next, justifying our approach with our error analysis.

Before delving deeper into the subject, let us also remark that when doing convolution operations, the filters' variances add up (this is considered in standard probability theory textbooks such as [7]), and as a result, when we decompose the original filter into a sequence of alternated horizontal and vertical passes we should account for this. Therefore we choose variances for each iteration such that their total sum adds up to the original σ_s (an "iteration" of the filter is defined as performing a horizontal pass followed by a vertical pass). As noted in [4], it is desirable as well to decrease the variance on each pass by a constant factor, and we choose it to be halved. This means that the i-th iteration has a deviation of $\sigma_i = \sigma_s \cdot \sqrt{3} \cdot 2^{N-i}/\sqrt{4^N - 1}$, where N is the number of iterations chosen. Now our algorithm is a composition of row passes and column passes with decreasing variance. As observed in [4], 3 iterations suffice, in our experiments we choose the number of iterations to be 2 unless stated otherwise.

B. Improving the algorithm's performance

Our approach is focused at improving performance when the number of SMs exceeds the number of horizontal or vertical stripes. As an example, let us take the number of SM cards as 9 (but note that the algorithm can be easily generalized for more SMs). We divide our image into 9 equal-sized domains, each of which will be attributed to a single SM card (Figure 4). Notice that this domain partition is not directly adequate to filtering in parallel, for example, SM2 should wait for SM1 to compute its boundary conditions so that it could start filtering. Now instead of waiting for the output of the SM1, we follow the scheme of extending the domain size as to obtain approximations for the boundary conditions of SM2 before SM1 computes its output. Notice that then SM2 must extrapolate its boundaries to approximate its left boundary conditions.

Algorithm B is applied to a boundary block to approximate its causal initial (boundary) conditions. In this boundary block,
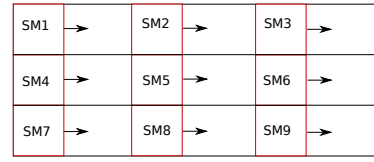
say B, we assume that generically, a thread is assigned points $B_{i,j} = t_j$, and we also choose an error threshold κ as an input.

Algorithm B:
- Starting from the initial point t_0, compute backwards the values Δ_{t_k}, $k = 0, -1, -2, ...$ until $\sum_{i=0}^{L-1} \Delta_{t_{-i}} \geq \kappa \cdot \sigma$;
- Filter the rows starting from t_{-L} until t_0 using the initial condition $g_i^+[-L] = \frac{a_i}{1-b_i} f[-L + 1]$. Output the approximated boundary condition $g_i^+[0]$.

We can now compute the filter in parallel for all the SM domains. On boundary blocks, we apply Algorithm B (and its anticausal/vertical variants) to obtain the initial conditions, and filtering is performed in each domain using Algorithm A. This allows more resource usage of the GPU. Our proposed approach (Algorithm C) thus uses Algorithm B in conjunction with Algorithm A.

Algorithm C:
- Partition the image in SM domains (Figure 4);
- Compute the initial conditions for the boundary blocks with Algorithm B;
- For each SM domain, filter analogously to Algorithm A, using the approximated initial conditions.

C. Practical details of our implementation

Our approach was to read an RGB image by padding it. We stored each pixel as a uchar4, a structure provided by CUDA. By doing this, we can extract groups of pixels in a single instruction. With $w = 24$ and using a padsize of 5 in shared memory, we had no bank conflicts in the algorithm.

Recall that the Gaussian filter we are using requires that we compute the complex powers $b_i^{\Delta_{t_k}}$. To work with complex numbers we used the Thrust library, which allows us to easily express complex arithmetic. However, when computing those powers, we chose not to use Thrust's `pow` function, since it makes use of double-precision values, and we prefer to restrict ourselves to single-precision values as this yields superior performance and is sufficient to capture the necessary precision.

So, to compute the complex powers we refer to Moivre's theorem. If z is a complex number and we want to compute z^x where x is real, then we first write z in its polar form as $r \cdot e^{i\theta}$. So $z^x = r^x \cdot e^{ix\theta} = r^x \cdot (\cos(x\theta) + i \cdot \sin(x\theta))$. Now the power r^x (which is a real value) and the trigonometric functions can be computed using CUDA's math library, which only uses single-precision arithmetic.

Also note that filtering uses a lot of complex constants, for instance $b_i, r_{i,j}, a_i$. We precompute those along with the polar form of the values b_i to compute complex powers. These

constants should not be stored as local variables in registers since this will potentially reduce performance. Our approach was to use these constants as global device variables, and since every thread requests these values simultaneously, they can be broadcasted. This approach provided superior performance.

In our implementation, we compute the filter's causal and anticausal components in parallel and then sum them [5]. We filter horizontally and then vertically for better filtering quality.

V. Performance and error evaluation

In this section we show the experimental aspects of our work, presenting and discussing our results.

### A. Experimental Design: Software and Hardware Platform

We developed our code in CUDA/C, using NVIDIA CUDA Compiler version 10.2, and experimented with the GPU cards GTX 1050 Ti and RTX 2080 Ti. The code is available at the companion artifact. We run experiments for both filtering quality and for execution time in the following subsections.

B. Control variables and observed outcomes

Our controllable variables in the experiments are σ, σ_r, κ, the filtering window dimensions $w \times h$, and the number of CUDA blocks per stripe used. The number of blocks controls the number of SM domains chosen as per Figure 4. It is expected that larger values for σ and σ_r should cause executions to last longer and to be more prone to numerical errors due to boundary approximation. In this section we investigate these effects. The variable σ_r is expected to interfere more in the results since it controls the mixing of the colors on the RGB range. Good choices for window dimensions depend on the hardware involved, and we restricted ourselves to values $h = 128$ and $w = 24$ as this was the proper choice for filtering in the GTX 1050 Ti. This choice also performed well for the RTX 2080 Ti. We also investigate how the execution time behaves by changing the filtered image dimensions.

Increasing the number of blocks per stripe should have the following effects: if we have spare SMs to assign to each block we expect to get a speedup. If we have more blocks than SMs, we could have either improved or worse results: it might be the case that a single SM could execute more than one block simultaneously (given it has the necessary memory resources) and this allows better resource utilization. Further, more block granularity might have slower blocks interfere less in the execution. However, having too many blocks might cause the approximation phase of the algorithm to overtake the execution time, since proportionally more blocks would be classified as boundary blocks.

In the experiments, we measured the total execution time of filtering rows and columns twice (two iterations). We do not consider the cost of loading the image into the GPU. We measure the approximation effects by computing the mean squared error between the approximated output and the desired output.
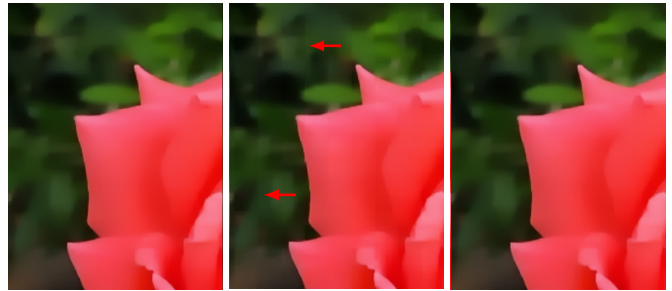
(a) Ground truth   (b) $\kappa = 0$ (c) $\kappa = 1.5$

Fig. 5: Under the same conditions of Figure 1 we highlight with red arrows the artifacts introduced by Algorithm D with $\kappa = 0$ (b), that is, not extending the blocks. Compare to the desired output from Algorithm A (a), which generated the correct filtering output but is slower (less parallelism). Algorithm D with $\kappa = 1.5$ (c) uses our boundary approximation, and with this value for κ the visual artifacts disappear.

C. Approximation of boundary conditions and impact on filtering quality

We explore how our proposed boundary condition approximation impacts the quality of the filter in practice. For this end, we use an implementation of the filter which computes the approximation for each block in the image in every direction (this is, we treat each block as a boundary block in every direction, and approximate every initial condition: upward, downward, forward, backward). Let's call this Algorithm D, which represents the worst-case scenario (in terms of filtering quality) since it requires the most approximations possible. This is only done to better understand the behavior of Algorithm C, which also uses boundary approximation but only in boundary blocks.

Figure 1 shows the correct filtering output. Figure 5(b) shows the result of Algorithm D with $\kappa = 0$ (*i.e.*, no boundary approximation). This causes the filtering to only occur within the blocks and introduces strong visual artifacts. These artifacts reduce as we set κ to larger values such as $\kappa = 1.5$ (Figure 5(c)). Figure 6 plots the filtering error versus the values of κ. We notice that the error decreases rapidly with $\kappa = 0.5$, and increasing κ beyond 2 does not provide better results in general. A good choice of κ depends on the inputs σ_s and σ_r as well. But notice that Algorithm C (our proposed implementation) will have much less visual artifacts than Algorithm D since it only requires approximating the boundary conditions in some regions (boundary blocks). By experimenting with 20 distinct images from Kodak Lossless True Color dataset, we saw little variability on the results, which seem to indicate that $\kappa = 2.0$ is a good choice. Furthermore, Figure 6 shows that there is very low variability on the MSE when $\kappa \geq 0.5$, indicating that individual features of the images contribute less to the overall error as κ increases. To understand this behavior, refer to Eq. (14), where the individual features of the image affect the error at most quadratically, while κ affects it exponentially.
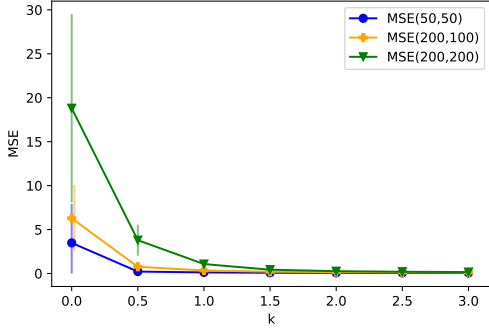
Fig. 6: With a set of 20 images from Kodak, we plot the mean squared error by comparing the approximated image from Algorithm D, and the desired output (Algorithm A) as a function of κ. We also plot the 95% confidence interval for each graph. Each graph in the form $\mathrm{MSE}(a, b)$ is the plot for the MSE when $(\sigma, \sigma_r) = (a, b)$.

Fig. 7: Comparison between original image (left) and filtered image with Algorithm C (right), for $\sigma = 70$ and $\sigma_r = 150$. With two row passes and two column passes. The image has dimensions 1024×768.

We recall that Algorithm A is the standard approach (high filtering quality but low parallelism), while Algorithm C requires approximating the image on SM domain boundaries, but allows for higher parallelism. In the next subsection, we show the gains obtained by allowing the use of more SM cards in the filtering process by making use of Algorithm C.

D. Comparing Algorithm A with Algorithm C

Our performance results can be seen in Figure 8. Our implementation of Algorithm C partitions each stripe (of height 128 pixels) in the image in a given number of sections, each section is filtered by a CUDA block. Comparing Figures 8a and 8c to Figures 8b and 8d we see that larger values for σ and σ_r introduce more variability in the timing results, although Algorithm C still performs better on average. We point out that the filtering parameters $\sigma = 200$ and $\sigma_r = 150$ used in the latter figures are exaggerated in order to stress the computational performance of our boundary approximation.

Notice that with the GTX 1050 Ti, Algorithm C outperforms Algorithm A for a 2048×2048 image. To explain this, observe that since each stripe has height 128, we have more stripes than SM cards (here we have 6 SM cards). Algorithm C performs

better because an SM can run more than one CUDA block if it has the necessary resources, so having more blocks allows better resource management. This also allows slower SMs to not impact so much the execution time.

From Figure 8a we see that having more blocks per row yielded worse results. We reason this behavior might be due to the fact that the increased amount of blocks per stripe causes the boundary approximation phase to be more computationally demanding, and so more blocks per stripe slow down the algorithm. However, this scenario seems to change, on average, in Figure 8b. This behavior remains to be explained. Despite the increased costs in the approximation phase, Algorithm C performed better on average even with 6 blocks per stripe.

The RTX 2080 Ti has 68 SM cards each with 64 processing cores. This makes it clear that increasing the number of blocks per stripe should improve the performance because we have spare SM cards for the filtering process. Because of this, we got much better results for Algorithm C than with Algorithm A, as can be seen from Figure 8c and Figure 8d. We executed the filtering process with the same window dimensions 128×24, which means in this case that we are assigning more lines to each window than the number of processing cores in the SM. To better match the architecture of the RTX 2080 Ti, we experimented with window dimensions 64×64 as well, but this change did not offer considerable improvement. For comparison, we ran a CPU version of filter on a 3.2 GHz i7-8700, with 12 threads, and it took roughly 6 seconds to filter the image with dimensions 2048^2 (versus around 0.01 seconds on the RTX 2080 Ti).

E. Discussion

We discuss scenarios that might arise when attempting an implementation by using the algorithm proposed by Nehab et al. [8]. Their algorithm reads an entire block into shared memory and performs all filtering operations in a single kernel. The best choice is equal block dimensions $B \times B$ for B equal to the number of cores in the SM cards. We note, however, that for some GPUs this choice would not allow us to properly use all cores in the SM. For example, the GTX 1050 Ti has 48Kb of shared memory and 128 cores per SM, so a block with dimensions 128×128 would require 49Kb of memory for an RGB image. This problem becomes worse when dealing with RGBA images. Therefore, either for horizontal or vertical filtering, we would have to use fewer cores than available. A possibility would be to implement their algorithm by using one kernel for horizontal filtering and another for vertical filtering, but this would require more than two image reads. The algorithm and implementation we propose in this paper address these limitations, but we have not compared the two algorithms in practical situations since the algorithm of Nehab et al. would require changes to be applied to edge-aware recursive filters.

VI. CONCLUSION

We presented an implementation for edge-aware Gaussian GPU filtering. Our approach is simple and allows us to properly scale the computations with the number of SM
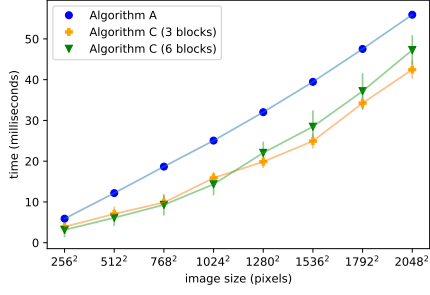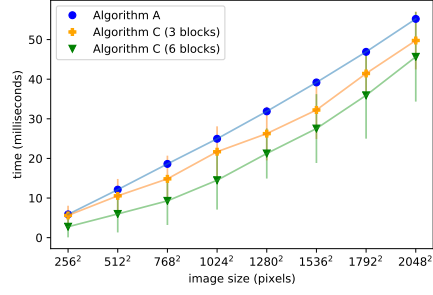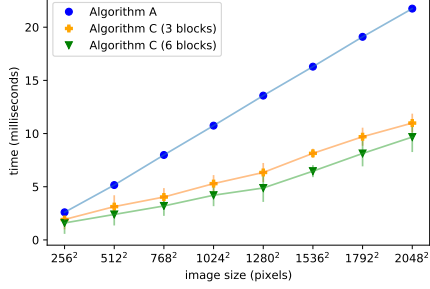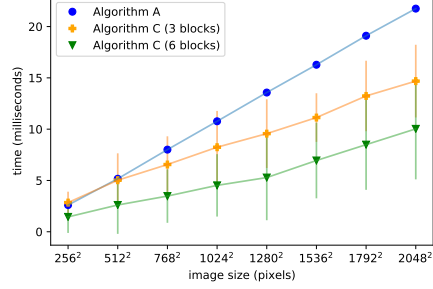
(a) GTX 1050 Ti with $\sigma = 50$ and $\sigma_r = 50$

(b) GTX 1050 Ti with $\sigma = 200$ and $\sigma_r = 150$

(c) RTX 2080 Ti with $\sigma = 50$ and $\sigma_r = 50$

(d) RTX 2080 Ti with $\sigma = 200$ and $\sigma_r = 150$

Fig. 8: Results on the RTX 2080 Ti and the GTX 1050 Ti. Measured by filtering on 5 diferent images, changing the size of the filtered area to dimensions $W \times W$ for $W = 256$ up to $W = 2048$. Each sample is an average of 50 measurements (each of the 5 images was filtered 10 times) with a 99% confidence interval around each average.

cards available on the GPU. When resources are properly available, execution time can be reduced by 50%, as indicated in Section V, Figure 8. And even when resources are not available, we reduced the execution time by the increased granularity in the domain. We managed to get good speedups while bounding the numerical errors by using our proposed Algorithm C. For future work, we believe that a comparison should be done to other approaches adapted for edge-aware recursive filtering [8]. We would also like to explore similar ideas to improve the quality and/or performance of space-frequency analysis methods [6].

SOFTWARE AND DATA AVAILABILITY

We endeavor to make our analysis reproducible. All the code used to generate the results can be found in the link below, including the images used and instructions:

https://github.com/hermes-hf/-edgeaware_gaussian_filter.

REFERENCES

[1] J. Chen, S. Paris, and F. Durand, "Real-time edge-aware image processing with the bilateral grid," *ACM TOG*, vol. 26, p. 103, 07 2007.

[2] R. Deriche, "Recursively implementing the gaussian and its derivatives," in *Proc. Secound Int. Conf. On Image Processing*, 1992, pp. 263–267.

[3] M. P. do Carmo, *Differential geometry of curves and surfaces.* Prentice Hall, 1976.

[4] E. S. L. Gastal and M. M. Oliveira, "Domain transform for edge-aware image and video processing," *ACM TOG*, vol. 30, no. 4, 2011.

[5] ——, "High-order recursive filtering of non-uniformly sampled signals for image and video processing," *CGF*, vol. 34, no. 2, May 2015.

[6] ——, "Spectral remapping for image downscaling," *ACM TOG*, vol. 36, no. 4, 2017.

[7] M. Loeve, *Probability theory*, 4th ed. Springer-Verlag New York, 1977.

[8] D. Nehab, A. Maximo, R. Lima, and H. Hoppe, "GPU-efficient recursive filtering and summed-area tables," *ACM TOG*, vol. 30, no. 6, 2011.

[9] D. Poulton and J. Oksman, "Digital filters for non-uniformly sampled signals," jan 2001.

[10] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley, 2010.

[11] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *ICCV*, 1998, pp. 839–846.

[12] Q. Yang, K.-H. Tan, and N. Ahuja, "Real-time O(1) bilateral filtering," vol. 1, 06 2009, pp. 557–564.

APPENDIX: MATHEMATICAL CONSTANTS

$$\alpha_0 = 1.6800 + 3.7350j, \qquad \lambda_0 = 1.783 + 0.6318j,$$
$$\alpha_1 = -0.6803 + 0.2598j, \qquad \lambda_1 = 1.723 + 1.9970j,$$
$$a_i = \alpha_i / \gamma, \qquad b_i = e^{-\lambda_i / \sigma},$$
$$r_{i,0} = \frac{(b_i - 1)^2}{a_i b_i}, \qquad r_{i,1} = \frac{a_i}{b_i - 1},$$
$$\gamma = \text{Re}\left\{ \alpha_0 \cdot \frac{1 + b_0}{1 - b_0} + \alpha_1 \cdot \frac{1 + b_1}{1 - b_1} \right\}.$$