ISSN 2464-4617 (print)
ISSN 2464-4625 (online)
Computer Science Research Notes - CSRN 3401
http://www.wscg.eu
WSCG 2024 Proceedings

Texture reading optimization for two-dimensional filter based graphics algorithms

Konstantin Zubatov
Faculty of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
Moscow, 119991, Russia
konstantin.zubatov@graphics.cs.msu.ru

Alexandr Shcherbakov
Faculty of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
Moscow, 119991, Russia
alex.shcherbakov@graphics.cs.msu.ru

ABSTRACT

There are many graphics algorithms that require reading textures in window 3x3, 5x5, etc. Both large window sizes and high rendering resolutions decrease performance. We propose an algorithm-independent method for reducing the number of texture samples implemented in a fragment shader for a local 2x2 area and based on the architectural features of the graphics pipeline — transferring data between fragments via difference derivatives.

Keywords
fragment shader, compute shader, difference derivatives, filters, texture sampling

1 INTRODUCTION

Figure 1: Filter illustration

Image filtering is widely used in computer graphics and image processing. Filters are required both as independent operations and for image processing before or after the main algorithms. Formally, we have an input image $I_{in} \in [0,1]^{w \times h \times c}$ and want to get an output image $I_{out} \in [0,1]^{w \times h \times c}$ by applying a two-dimensional filter function $f : I_{in} \mapsto I_{out}$. Usually it takes a local area (negative indices are used for convenience)

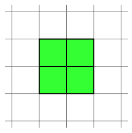
$$A(x,y) = (I_{in}(x+i, y+j))_{-r_w \le i \le r_w, -r_h \le j \le r_h}, \; r_w, r_h \in \mathbb{N}$$

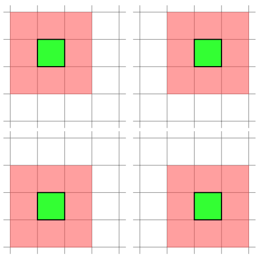
of each texel $I_{in}(x,y)$ and produces single fragment $I_{out}(x,y)$, where $(x,y) \in [0, w-1] \times [0, h-1]$. In the following, only local areas of size $(2r+1) \times (2r+1)$, $r \in \mathbb{N}$, will be considered for two-dimensional filters. As the image or local area size increases, the performance decreases due to the larger number of texture samples required.

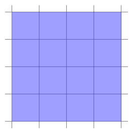
Filtering is usually done on the GPU because it is well parallelized and the result is needed for other render targets (in the case of real-time computer graphics). There are two suitable types of shaders — compute and fragment shaders.

A compute shader allows to reduce bandwidth requirements by using shared memory [Kil12]. In this case, each invocation in the workgroup reads several texels and writes them to fast shared memory located on the chip (see Figure 2), from where each invocation then reads the required texels. Practice shows that this method is not always applicable for mobile devices. For example, Arm GPUs do not implement dedicated on-chip shared memory for compute shaders. The shared memory that is available to use is system RAM that is backed up by the load-store cache [Arm].

A fragment shader does not have shared memory, but is used in rendering, so the filter can be combined with additional operations in the fragment shader. Regardless of the type of shader, another approach to optimizing memory consumption is used — two-pass rendering for separable filters. Separable filters are filters that can be written as the product of two one-dimensional


(a) 2x2 workgroup, green squares are threads


(b) Each thread runs 3x3 filter (local areas are red), invocations are separated for better readability


(c) We can allocate 4x4 array in shared memory (it is blue) and fill it with necessary texels

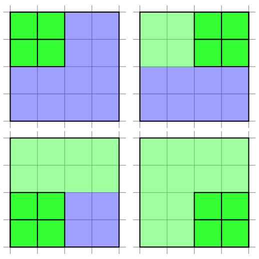

(d) Reading texels in four steps. The bright green squares are threads and the pale green squares are read texels

Figure 2: Example for 2x2 workgroup and 3x3 filter in compute shader. Each thread performs only 4 reads. Without shared memory, this number will increase to 9. In practice, larger workgroups (e.g. 16x16) and other sampling strategies (e.g. linear) are used.

filters. Using this feature, we can compute the same result with $4r+2$ texture samples instead of $(2r+1)^2$, but this would require an intermediate image of the same resolution and additional render pass.

2 RELATED WORKS

2.1 Commonly used filters

Discrete convolutional filters (1) are often used.

$$f(I_{in}(x,y)) = \frac{\sum_{i=-r}^{r}\sum_{j=-r}^{r} g(i,j) I_{in}(x-i,y-i)}{\sum_{i=-r}^{r}\sum_{j=-r}^{r} g(i,j)} \quad (1)$$

Moving from the classical definition of convolution to implementation in a shader, we rewrite the formula for a convolution filter

$$f(I_{in}(x,y)) = \frac{\sum_{i=-r}^{r}\sum_{j=-r}^{r} g(i,j) I_{in}(x+i,y+i)}{\sum_{i=-r}^{r}\sum_{j=-r}^{r} g(i,j)} \quad (2)$$

Hereafter in the text the filter is understood as a function (2) with different kernels $g(i,j)$, and r everywhere means the radius of the local area (hence its size is $(2r+1)\times(2r+1)$) of the filter. In this paper we will discuss four discrete convolutional filters: Gaussian filter, bilateral filter, tent filter and variance clipping for temporal anti-aliasing (TAA).

2.1.1 *Gaussian filter*

A Gaussian filter is a separable filter that is used to produce post effects (like bloom or depth-of-field), as part of Canny edge detection [Can86], and in variance shadow maps [Don06]. Filter's function is described using kernel (3) (factor $\frac{1}{2\pi\sigma^2}$ is reduced)

$$g(i,j) = \exp\left(-\frac{i^2+j^2}{2\sigma^2}\right) \quad (3)$$

where σ is the standard deviation of the Gaussian distribution (greater value — stronger blur).

2.1.2 *Bilateral filter*

A bilateral filter is described using kernel (4)

$$g(i,j) = \exp\left(-\frac{i^2+j^2}{2\sigma_d^2} - \frac{\|I_{in}(x+i,y+j)-I_{in}(x,y)\|^2}{2\sigma_r^2}\right) \quad (4)$$

where σ_d and σ_r are smoothing parameters. It is a nonlinear, edge-preserving filter. One of its uses is to prevent incorrect diffusion in skin rendering [Buc16]. Another application is image upsampling [Kop07].

2.1.3 *Tent filter*

A tent filter is described using kernel (5)

$$g(i,j) = \max(k-b|i|,\ 0)\max(k-b|j|,\ 0) \quad (5)$$

where k and b are parameters. It is used in temporal upsampling for pre-processing stage [Yan09], in image downscaling and upscaling for video encoding [Bri99], and as individual filter for blur in image processing [App]. In our solution we modified tent filter and made it nonseparable, with kernel rank 2. It is almost equal to the original filter and has kernel (6)

$$g(i,j) = k - b(|i|+|j|) \quad (6)$$

where k and b are same. This downgrade was done to test the applicability of the proposed method to the filter with rank 2.

2.1.4 *Variance clipping, TAA*

Temporal Antialiasing (also known as Temporal AA, or TAA) is a family of techniques that perform spatial antialiasing using data gathered across multiple frames. One of the important part of any TAA algorithm is history rectification — a process of adapting a color from previous frame(s). Variance clipping [Sal16] addresses

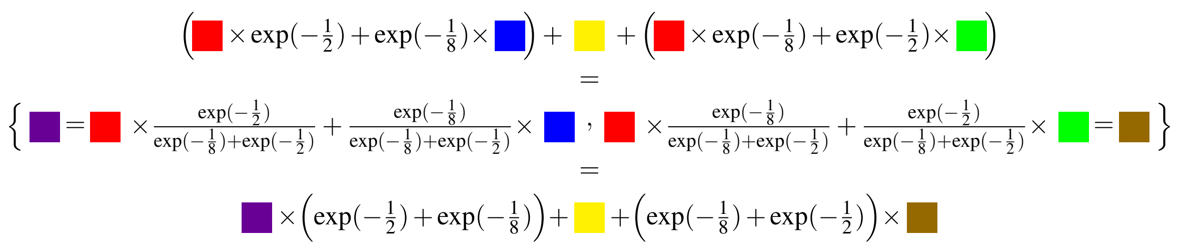


Figure 3: [Rak10] method, $\sigma = 2$, 5x5 local area, horizontal pass, colored squares are read texels. The result in the third row is the same as in the first row, but requires three texels. The purple and brown texels in the third row are linear interpolation (see second row). The readout time of a texel for a linear sampler is independent of whether it is a real texel or a bilinear interpolation of four real texels

outliers by using the local color mean and standard deviation to center and size the color extents used for rectification:

$$C_{min} = \mu - \gamma\sigma \qquad C_{max} = \mu + \gamma\sigma$$

where μ and σ are the mean and standard deviation of the color samples in the local area, and γ is a scalar parameter, $\gamma \in [0.75, 1.25]$. Calculation of μ and γ can be written as filters (7) and (8):

$$\mu = \frac{\sum_{i=-r}^{r}\sum_{j=-r}^{r} I_{in}(x+i, y+j)}{(2r+1)^2} \tag{7}$$

$$\sigma = \sqrt{\frac{\sum_{i=-r}^{r}\sum_{j=-r}^{r} I_{in}^2(x+i, y+j)}{(2r+1)^2} - \mu^2} \tag{8}$$

2.2 Memory optimization work

[Rak10] proposed an improvement for separable Gaussian filter based on a linear texture sampler that returns a bilinear interpolation of the four nearest pixels instead of nearest pixel color for the linear sampler. Let us describe a one-dimensional filter for the first (along x-axis) pass:

$$f(I_{in}(x,y)) = \frac{\sum_{i=-r}^{r} \exp(-\frac{i^2}{2\sigma^2}) I_{in}(x+i, y)}{\sum_{i=-r}^{r} \exp(-\frac{i^2}{2\sigma^2})}$$

We can describe the contribution of two neighboring pixels $I_{in}(x+j, y)$ and $I_{in}(x+j+1, y)$ as:

$$\frac{I_{in}(x+j,y)w_1 + I_{in}(x+j+1,y)w_2}{\sum_{i=-r}^{r} exp(-\frac{i^2}{2\sigma^2})} =$$

$$\frac{\{I_{in}(x+j,y)\frac{w_1}{w_1+w_2} + I_{in}(x+j+1,y)\frac{w_2}{w_1+w_2}\}(w_1+w_2)}{\sum_{i=-r}^{r} exp(-\frac{i^2}{2\sigma^2})}$$

$$= \frac{I_{in}(\frac{(x+j)w_1+(x+j+1)w_2}{w_1+w_2}, y)(w_1+w_2)}{\sum_{i=-r}^{r} exp(-\frac{i^2}{2\sigma^2})}$$

where $w_1 = \exp(-\frac{j^2}{2\sigma^2})$, $w_2 = \exp(-\frac{(j+1)^2}{2\sigma^2})$. The expression demonstrates linear interpolation multiplied by the sum of weights. This method reduces the number of texture samples from $4r+2$ to $2r+2$ in two passes (see example in Figure 3). This is a very efficient method, but it's only applicable to separable filters and requires intermediate image to store the output of the first pass.

3 PROPOSED SOLUTION

Our solution is based on the fact that the fragment shader always works for 2x2 groups of fragments. It is useful for calculating the MIP-level for a texture and provides differential derivatives along the x and y axes. They are typically used to calculate texture MIP-level (separate from texture reading), flat normals, or edge detection. We will use fine differential derivatives along the x and y axes ($dFdxFine$ and $dFdyFine$ in GLSL) and rename them $ddx(p)$ and $ddy(p)$, respectively, where p is a scalar or vector. $ddx(p)$ and $ddy(p)$ calculate derivatives using local differencing based on the value of p for the current fragment and it's immediate neighbor (see Figure 4).

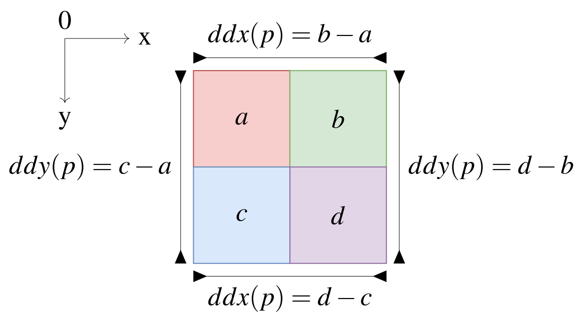


Figure 4: $ddx(p)$ and $ddy(p)$ for 2x2 group of fragments. a, b, c, d are the value of p in the corresponding fragment. Fragments in the column {row} have the same $ddy(p)$ $\{ddx(p)\}$. The x and y axes show the coordinate directions.

It is required to define t_x, t_y

$$t_x = \begin{cases} -1 & \text{if fragment in odd col} \\ 1 & \text{if fragment in even col} \end{cases}$$

$$t_y = \begin{cases} -1 & \text{if fragment in odd row} \\ 1 & \text{if fragment in even row} \end{cases}$$

so that the fragments can exchange data with their immediate neighbors:

$$p_{recvd} = p_{our} + t_x \cdot ddx(p_{our}), \text{ swap along x-axis} \quad (9)$$

$$p_{recvd} = p_{our} + t_y \cdot ddy(p_{our}), \text{ swap along y-axis} \quad (10)$$

Using data transfer, we can describe a naive algorithm that exchanges texels between fragments and reduces the number of texture reads from $(2r+1)^2$ to $(r+1)^2$ for each fragment (see Figure 5):

1. Each fragment reads texels $I_{in}(x-r+2i, y-r+2j)$, where $0 \leq i, j \leq r$. Now each fragment has $r+1$ rows with $r+1$ texels in each (see Figure 5b).
2. We can select r texels from each non-empty row and do swaps along x-axis (see Figure 5c).
3. Now we have $r+1$ full rows of texels. r rows from them are needed for swaps along y-axis (see Figure 5d).
4. After swaps along y-axis each fragment has all texels (see Figure 5e) and computes the filter function.

Unfortunately, use of fine difference derivatives for all texels transfer ($3r^2+2r$ times) works slower than $(2r+1)^2$ texture readings. To improve performance, the shader should perform multiple partial sum transfers computed with its own texels but kernel values required by its neighbours. Let us describe this method with partial sums for the four filters mentioned earlier.

3.1 Gaussian filter

The Gaussian filter kernel (3) has rank 1, which means that all rows (columns) in kernel are a linear combination of one row (column). Therefore, it is enough to:

1. For each $i \in T$, $T = \{-r+2k \mid 0 \leq k \leq r\}$:
   (a) Read texels $I_{in}(x+j, y+i)$, $j \in T$
   (b) initialize the partial sum p_i
   $$p_i := \sum_{j \in T} \exp(-\frac{i^2+j^2}{2\sigma^2}) I_{in}(x+j, y+i)$$
   (c) Select texels for the neighbour on the x-axis. If the fragment (which is executing the shader, not the neighbour!) has even column number, it should select all texels in row except the first one, $\Omega = T \setminus \{-r\}$. Otherwise — except the last one, $\Omega = T \setminus \{r\}$.

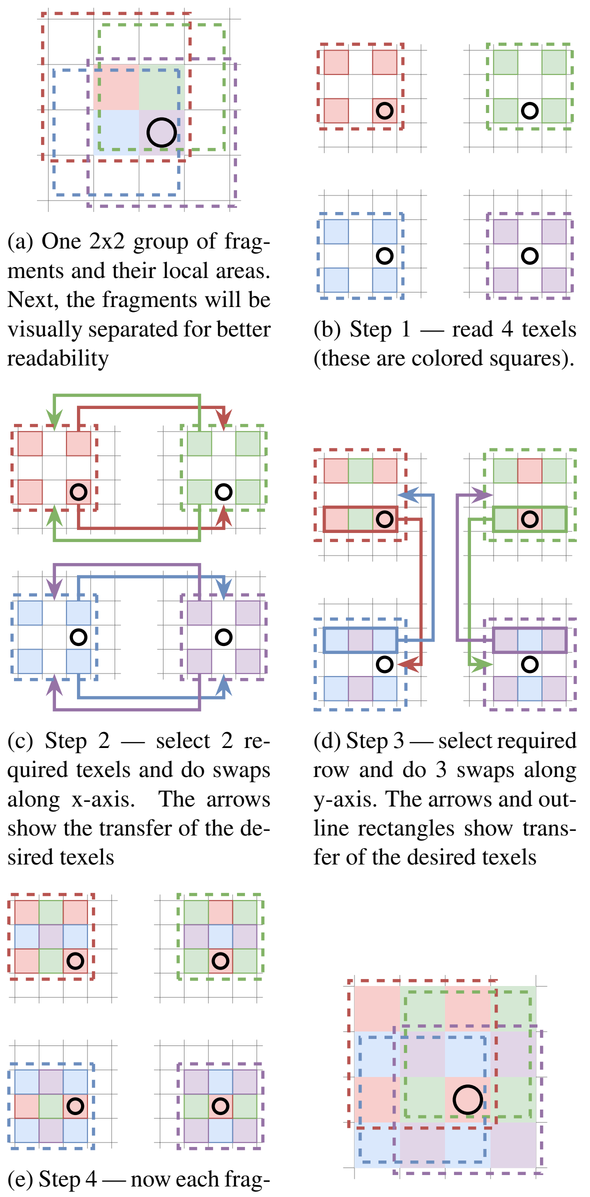

(a) One 2x2 group of fragments and their local areas. Next, the fragments will be visually separated for better readability
(b) Step 1 — read 4 texels (these are colored squares).
(c) Step 2 — select 2 required texels and do swaps along x-axis. The arrows show the transfer of the desired texels
(d) Step 3 — select required row and do 3 swaps along y-axis. The arrows and outline rectangles show transfer of the desired texels
(e) Step 4 — now each fragment contains all required texels
(f) Visual merging of fragments

Figure 5: Naive algorithm illustration for 3x3 filter. Texel with black circle is the same for all fragments. Dashed rectangles cover local areas for each fragment

   (d) Compute partial sum p'_i for the neighbour on the x-axis (consider the bias with t_x) (see Figure 6a)
   $$p'_i := \sum_{j \in \Omega} \exp(-\frac{i^2+(j-t_x)^2}{2\sigma^2}) I_{in}(x+j, y+i)$$
   (e) Make a swap along the x-axis and complete the partial sum p_i
   $$p_i^{final} := p_i + p'_i + t_x \cdot ddx(p'_i),$$
   (f) p_i^{final} equals $\sum_{j=-r}^{r} \exp(-\frac{i^2+j^2}{2\sigma^2}) I_{in}(x+j, y+i)$

2. initialize the partial sum of the filter

$$S := \sum_{i \in T} p_i^{final}$$

3. Choose the partial sums p_i^{final} required for the neighbour on the y-axis. If the fragment has even row number, it should select all sums p_i^{final} except the first one, $\Omega = T \setminus \{-r\}$. Otherwise — except the last one, $\Omega = T \setminus \{r\}$.

4. It was mentioned that each row of the kernel matrix (3) can be expressed as a linear combination. We can interpolate partial sums for rows that our neighbour on the y-axis does not know about by multiplying them by certain exponents. If p_i^{final} has a common factor $\exp(-\frac{i^2}{2\sigma^2})$ in this fragment, the neighbor on the y-axis has a common factor $\exp(-\frac{(i-t_y)^2}{2\sigma^2})$. Therefore, we should compute partial sum S' for the neighbor on the y-axis (see Figure 6b)

$$S' := \sum_{i \in \Omega} \exp(\frac{2it_y - 1}{2\sigma^2}) p_i$$

5. Make a swap along the y-axis and complete the sum

$$S^{final} := S + S' + t_y \cdot ddy(S')$$

6. S^{final} equals $\sum\limits_{i=-r}^{r} \sum\limits_{j=-r}^{r} \exp(-\frac{i^2+j^2}{2\sigma^2}) I_{in}(x+i, y+j)$. Normalize S^{final} by the sum of the kernel.

3.2 Tent filter

The kernel (5) is suitable for the previous method. The kernel of modified tent filter (6) has rank 2. Therefore, we need to modify the previous algorithm:

1. For each $i \in T$, $T = \{-r + 2k \mid 0 \le k \le r\}$:

   (a) Read texels $I_{in}(x + j, y + i)$, $j \in T$

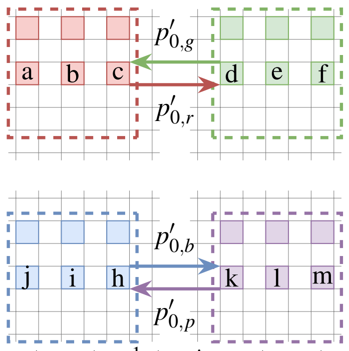
   (b) initialize the partial sum p_i

   $$p_i := \sum_{j \in T} (k - b(|i| + |j|)) I_{in}(x + j, y + i)$$

   and additional partial sum u_i (required in step 4)

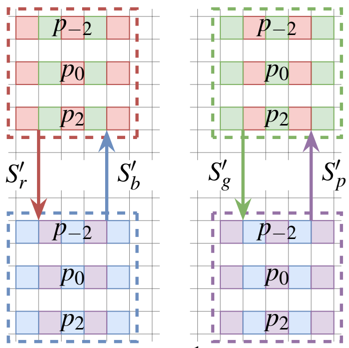
   $$u_i := \sum_{j \in T} I_{in}(x + j, y + i)$$

   (c) Select texels for the neighbour on the x-axis. If the fragment (which is executing the shader, not the neighbour!) has even column number, then it should select all texels in row except the first one, $\Omega = T \setminus \{-r\}$. Otherwise — except the last one, $\Omega = T \setminus \{r\}$.



$$p'_{0,r} = (b + c)\exp(-\tfrac{1}{2\sigma^2}),\ p'_{0,g} = (d + e)\exp(-\tfrac{1}{2\sigma^2})$$
$$p'_{0,b} = (i + h)\exp(-\tfrac{1}{2\sigma^2}),\ p'_{0,p} = (k + l)\exp(-\tfrac{1}{2\sigma^2})$$

(a) Data swap along x-axis in step 1 when $i = 0$. The letters in the squares represent texels



$$S'_r = S'_g = p_0 \times \exp(-\tfrac{1}{2\sigma^2}) + p_2 \times \exp(\tfrac{3}{2\sigma^2})$$
$$S'_b = S'_p = p_0 \times \exp(-\tfrac{1}{2\sigma^2}) + p_{-2} \times \exp(\tfrac{3}{2\sigma^2})$$

(b) Steps 4 and 5. p_{-2}, p_0, p_2 are different in each fragment

Figure 6: Proposed method for 5x5 Gaussian filter. The arrows indicate the transfer of the respective sums

   (d) Compute partial sum p'_i for the neighbour on the x-axis (consider the bias with t_x)

   $$p'_i := \sum_{j \in \Omega} (k - b(|i| + |j - t_x|)) I_{in}(x + j, y + i)$$

   $$u'_i := \sum_{j \in \Omega} I_{in}(x + j, y + i)$$

   (e) Make a swap along the x-axis and complete the partial sums p_i and u_i

   $$p_i^{final} := p_i + p'_i + t_x \cdot ddx(p'_i),$$
   $$u_i^{final} := u_i + u'_i + t_x \cdot ddx(u'_i),$$

2. initialize the partial sum of the filter

$$S := \sum_{i \in T} p_i^{final}$$

3. Choose the partial sums p_i^{final} and u_i^{final} required for the neighbor on the y-axis. If the fragment has even row number, it should select all sums except p_{-r}^{final} and u_{-r}^{final}, $\Omega = T \setminus \{-r\}$. Otherwise — $\Omega = T \setminus \{r\}$.

4. Compute partial sum for the neighbor on the y-axis

$$S' := \sum_{i\in\Omega} p_i^{final} + b(|i| - |i - t_y|)u_i^{final}$$

5. Make a swap along the y-axis and complete the partial sum S

$$S^{final} := S + S' + t_y \cdot ddy(S')$$

6. S^{final} equals $\sum\limits_{i=-r}^{r}\sum\limits_{j=-r}^{r}(k - b(|i| + |j|))I_{in}(x+i, y+j)$. Normalize S^{final} by the sum of the kernel.

3.3 TAA, variance clipping

The kernels for filters (7) and (8) (matrices of ones) have rank 1. Therefore, for 3x3 local area we can apply the same algorithm as for the Gaussian filter, but twice. For larger local areas this is not optimal in terms of the number of derivatives. Instead of rows interpolation, we can directly calculate partial sums for other three fragments in the constant derivatives number. This method uses nonuniform texture sampling (see Figure 7) for efficient choose of required texels (see Figure 8).

$$T = \{-r + 2k \mid 0 \le k \le r\},\ Q = T \setminus \{-r\}$$

1. Read texels in nonuniform mode:

$$I_{in}(x + it_x, y + jt_y),\ i, j \in T$$

2. Compute the partial sums for the neighbour on the y-axis (see Figure 8a):

$$\mu'_y := \sum_{i\in T}\sum_{j\in Q} I_{in}(x + it_x, y + jt_y)$$

$$\sigma'_y := \sum_{i\in T}\sum_{j\in Q} I^2_{in}(x + it_x, y + jt_y)$$

3. Make a swap along the x-axis and contribute a part in diagonal neighbour's partial sums (see Figure 8b):

$$\mu'_d := \mu'_y + t_x \cdot ddx(\mu'_y) + \sum_{i\in Q}\sum_{j\in Q} I_{in}(x + it_x, y + jt_y)$$

$$\sigma'_d := \sigma'_y + t_x \cdot ddx(\sigma'_y) + \sum_{i\in Q}\sum_{j\in Q} I^2_{in}(x + it_x, y + jt_y)$$

4. Make a swap along the y-axis and contribute a part in the partial sums for the neighbour along the x-axis (see Figure 8c):

$$\mu'_x := \mu'_d + t_y \cdot ddy(\mu'_d) + \sum_{i\in Q}\sum_{j\in T} I_{in}(x + it_x, y + jt_y)$$

$$\sigma'_x := \sigma'_d + t_y \cdot ddy(\sigma'_d) + \sum_{i\in Q}\sum_{j\in T} I^2_{in}(x + it_x, y + jt_y)$$

5. Make a swap along the x-axis and complete sums (see Figure 8d):

$$\mu'_{final} := \mu'_x + t_x \cdot ddx(\mu'_x) + \sum_{i\in T}\sum_{j\in T} I_{in}(x + it_x, y + jt_y)$$

$$\sigma'_{final} := \sigma'_x + t_x \cdot ddx(\sigma'_x) + \sum_{i\in T}\sum_{j\in T} I^2_{in}(x + it_x, y + jt_y)$$

6. Complete the calculations of filters (7) and (8):

$$\mu := \frac{\mu'_{final}}{(2r+1)^2}$$

$$\sigma := \sqrt{\frac{\sigma'_{final}}{(2r+1)^2} - \mu^2}$$

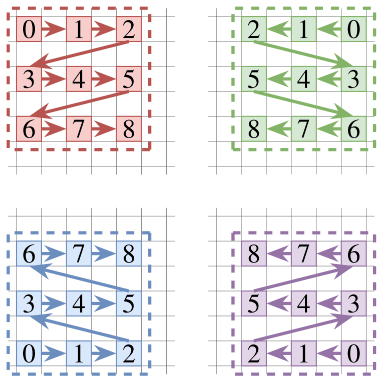
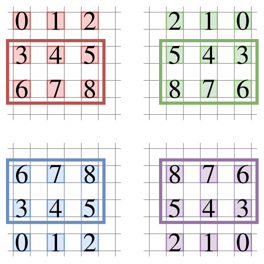
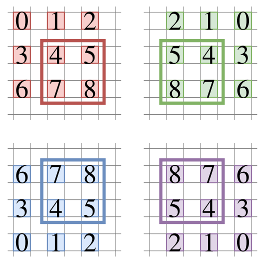


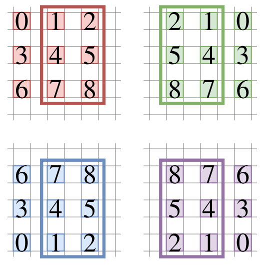
Figure 7: Nonuniform texture sampling for 2x2 group, 5x5 local area. The numbers represent the texel indices in the array. Arrows indicate the direction of sampling.



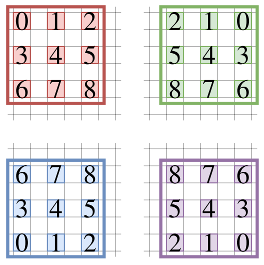
(a) For neighbour on y-axis



(b) For diagonal neighbour



(c) For neighbour on x-axis



(d) For fragment itself

Figure 8: Proposed method, 5x5. The rectangles in each subfigure cover the texels required for that step.

3.4 Bilateral filter

Computing the partial sums of other fragments with nonuniform sampling (see Figure 7) is the only way, since bilateral filter is nonlinear. Since its kernel (4) depends on the central texel $I_{in}(x,y)$ (each fragment has a unique central texel), we must also pass it between fragments. At the beginning, we have one of two situations:

1. The local area radius is $r = 2k$, $k \in \mathbb{N}$ and central texel $I_{in}(x,y)$ is sampled (3rd case in Figure 9).
   (a) In this case, we swap the central texel with our neighbour along the x-axis and proceed to the 4th case.
   (b) In the 4th case, we compute the partial sums for the numerator and denominator of the filter and proceed to the 1st case by swapping partial sums and central texel with the neighbour along the y-axis.
   (c) In the 1st case, we compute the partial sums for the numerator and denominator of the filter and proceed to the 2nd case by swapping partial sums and central texel with the neighbour along the x-axis.
   (d) In the 2nd case, we compute the partial sums for the numerator and denominator of the filter and proceed to the 3rd case by swapping partial sums with the neighbour along the y-axis.
   (e) In the 3rd case, we use our own central texel and complete the sums.
2. The local area radius is $r = 2k-1$, $k \in \mathbb{N}$ and central texel $I_{in}(x,y)$ is sampled by our diagonal neighbor (1st case in Figure 9)
   (a) In this case, we compute the partial sums for the numerator and denominator of the filter and proceed to the 2nd case by swapping partial sums and central texel with the neighbour along the x-axis.
   (b) In the 2nd case, we compute the partial sums for the numerator and denominator of the filter and proceed to the 3rd case by swapping partial sums with the neighbour along the y-axis.
   (c) In the 3rd case, we compute the partial sums for the numerator and denominator of the filter and proceed to the 4th case by swapping partial sums with the neighbour along the x-axis.
   (d) In the 4th case, we compute the partial sums for the numerator and denominator of the filter and proceed to the 3rd case by swapping partial sums with the neighbor along the x-axis. After that we have completed sums.

In the 1st situation at the beginning (and if $r = 1$), 6 partial derivatives are required. In the 2 situation (except $r = 1$), 7 partial derivatives are required (the numerator is a three-dimensional vector and the denominator is a scalar, so they were combined into one four-dimensional vector). Let us describe the partial sums for numerator and denominator in each case (number corresponds to the case number on Figure 9):

$$T = \{-r+2k \mid 0 \le k \le r\},\ Q = T \setminus \{-r\}$$

1.
$$c := \sum_{i\in Q}\sum_{j\in Q} I_{in}(x+it_x, y+jt_y)g(it_x, jt_y)$$
$$w := \sum_{i\in Q}\sum_{j\in Q} g(it_x, jt_y)$$

2.
$$c := \sum_{i\in T}\sum_{j\in Q} I_{in}(x+it_x, y+jt_y)g(it_x, jt_y)$$
$$w := \sum_{i\in T}\sum_{j\in Q} g(it_x, jt_y)$$

3.
$$c := \sum_{i\in T}\sum_{j\in T} I_{in}(x+it_x, y+jt_y)g(it_x, jt_y)$$
$$w := \sum_{i\in T}\sum_{j\in T} g(it_x, jt_y)$$

4.
$$c := \sum_{i\in Q}\sum_{j\in T} I_{in}(x+it_x, y+jt_y)g(it_x, jt_y)$$
$$w := \sum_{i\in Q}\sum_{j\in T} g(it_x, jt_y)$$

where c is the partial sum of the numerator, w is the partial sum of the denominator. Swaps are performed according to the formulas (9) and (10).

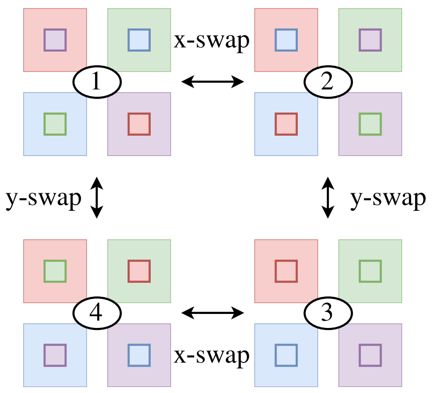


Figure 9: Four cases for the bilateral filter. The big squares are fragments. The small squares are the central texels for the corresponding fragments. From each case, you can move to one of the two neighbouring cases through swaps along the x or y axis.

4 EXPERIMENTAL RESULTS

We have used the Khronos framework [Khr19] for the Vulkan API to run our GLSL shaders on PC and Android. The comparison includes the following types of shaders (see Figures 11 and 12):

- Default version — filters are implemented in fragment shader and read $(2r+1)^2$ texels
- Compute version — filters are implemented in compute shader using shared memory. Gaussian filter also uses two-pass rendering with 128x1 workgroups. Other compute shaders use 16x16 workgroups
- Proposed version — filters are implemented in fragment shader and use derivatives
- Linear version — Gaussian filter is implemented in fragment shader and uses two-pass rendering with a linear sampler [Rak10]

Adreno 650 shows 5-100% performance degradation depending on filter and area size when using shared memory. It can be concluded that Adreno has a high cost of fine derivatives. For 3x3 filters, the performance of our method is worse by at least 56% compared to a default fragment shaders (or does not change for variance clipping). However, for large areas, our method outperforms default fragment shaders by at least 15% (except 7x7 tent filter). Also worth mentioning is the method with the linear sampler — at least 37% better performance than competitors.

RTX 2070 has relatively cheap derivatives, due to which we get a 25-50% improvement over a default fragment shader and compute shader. For this video card, our method bypasses the fragment shader with linear sampler by 42% and 24% for 3x3 and 5x5 area respectively. However, for the 7x7 region, our method is 13% slower.

It can be seen from the graphs that different GPU architectures have different costs of fine derivatives. However, as the region size increases, the proposed method shows an increasingly noticeable difference in performance, although it depends on the filter type.

4.1 Metrics and coarse derivatives

Area size	Gaussian	Tent	Bilateral	VC
3x3	1.62e-6	0	1.89e-6	1.89e-6
5x5	2.71e-6	0	2.98e-6	3.25e-6
7x7	2.98e-6	0	3.52e-6	4.34e-6

Table 1: MSE for default and proposed (with fine derivatives) methods

The proposed method with fine derivatives yields images that are almost identical to those obtained in the default method (see Tables (1) and (2)).

Area size	Gaussian	Tent	Bilateral	VC
3x3	0.999	1.0	0.999	0.999
5x5	0.999	1.0	0.999	0.999
7x7	0.999	1.0	0.999	0.999

Table 2: SSIM for default and proposed (with fine derivatives) methods

Area size	Gaussian	Tent	Bilateral	VC
3x3	3.62	4.34	14.19	685.31
5x5	1.165	1.19	14.51	122.98
7x7	0.513	0.62	21.06	29.99

Table 3: MSE for default and proposed (with coarse derivatives) methods

Area size	Gaussian	Tent	Bilateral	VC
3x3	0.991	0.99	0.988	0.503
5x5	0.995	0.995	0.992	0.854
7x7	0.997	0.997	0.984	0.966

Table 4: SSIM for default and proposed (with coarse derivatives) methods

In the early stages, testing the proposed method with coarse derivatives did not yield any noticeable performance gains for the RTX 2070. Because of this and the metrics (see Tables (3) and (4)), we abandoned the coarse derivatives in performance measurement. While for the Gaussian filter the difference is negligible, for variance clipping the method with coarse derivatives is not applicable at all (see Figure 10).

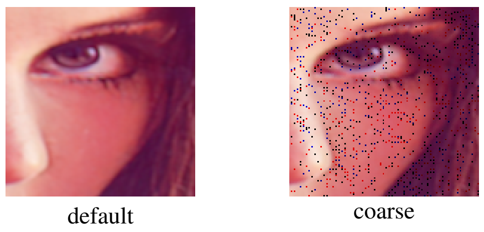


Figure 10: Mean value in variance clipping, 3x3. The default method produces correct results, while proposed method with coarse derivatives produces visual artefacts due to incorrect texels received via swaps

5 CONCLUSION

We have presented a few strategies for memory optimization in two-dimensional filters based graphics algorithms. It does not require intermediate images, gives noticeable performance gains almost everywhere, and can only be implemented by rewriting shaders. Unfortunately, explicit difference derivatives are not always cheap enough to allow blind replacement of shaders with the proposed ones. The source code of the shaders is available on GitHub: `https://github.com/kzubatov/FiltersOptimizationInfo`. Performance tests made by the community on other GPU architectures will also be available there.

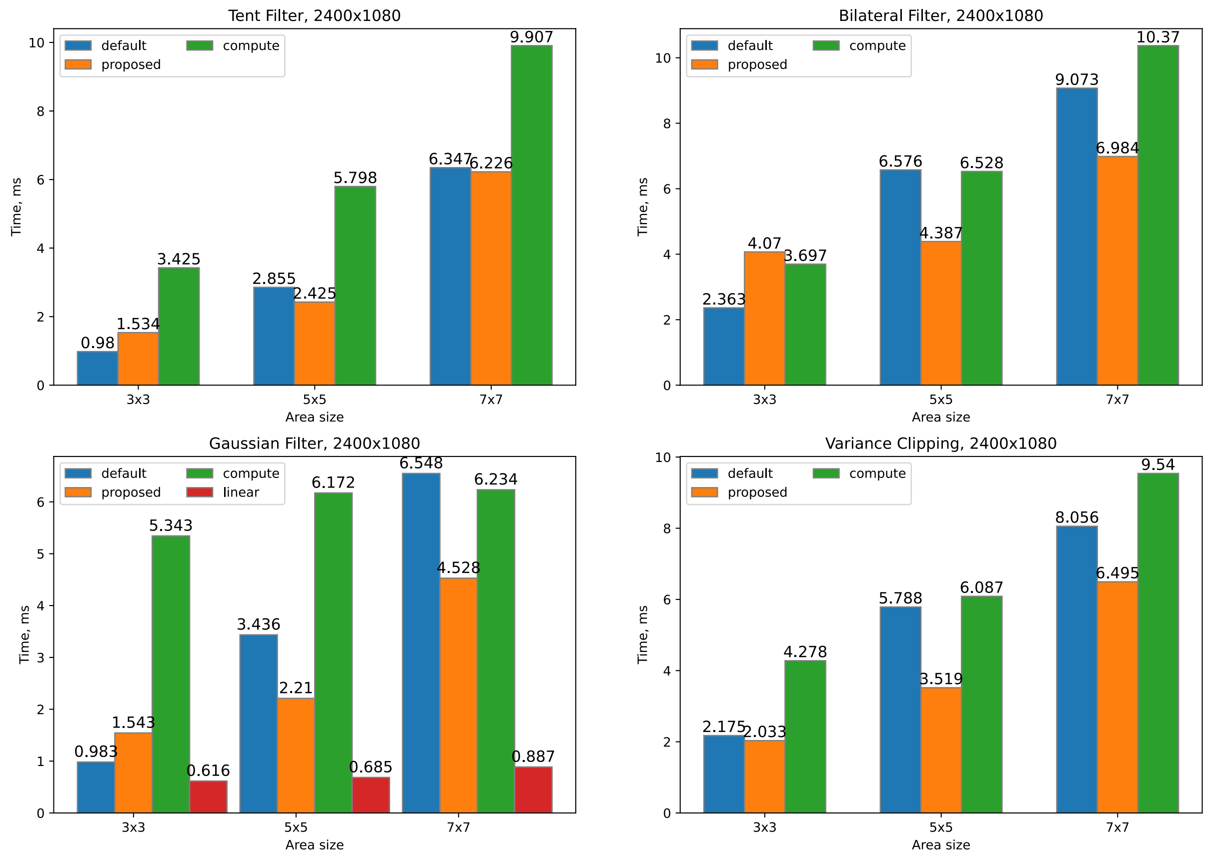


Figure 11: Performance tests, Adreno 650, Android 13

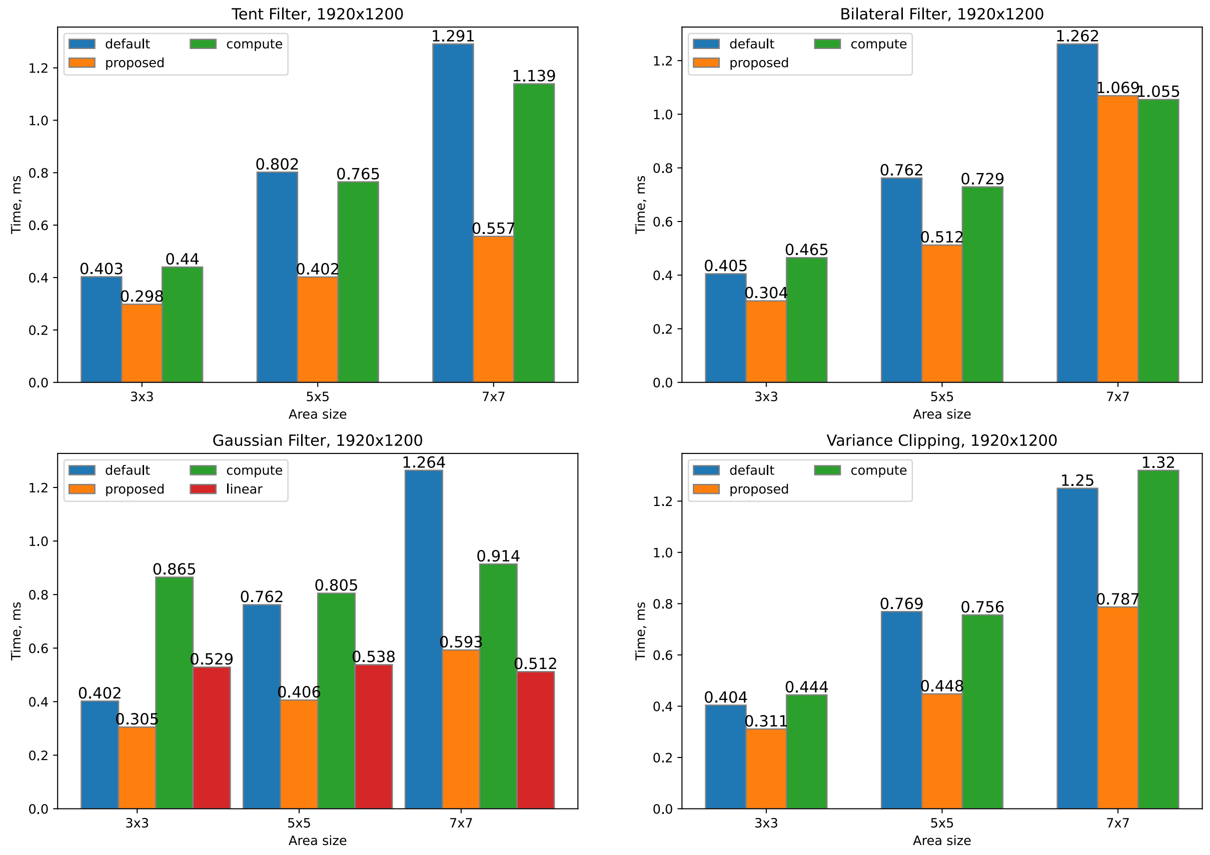


Figure 12: Performance tests, RTX 2070, Linux 5.15

6 REFERENCES

[App] Apple documentation. Blurring an image. https://developer.apple.com/documentation/accelerate/blurring_an_image

[Arm] Arm GPU Best Practices Developer Guide. Compute shading. Shared memory. https://developer.arm.com/documentation/101897/0302/Compute-shading/Shared-memory

[Bri99] Briceno, Hector, Steven Gortler, and Leonard McMillan: NAIVE - Network Aware Internet Video Encoding. Proceedings of he 7th ACM International Multimedia Conference (Part 1), 1999.

[Buc16] R.H. Buch, A.C. Calaf, P.P.V. Alcocer, "Optimized skin rendering for scanned models". In WSCG 2016: short communications proceedings: The 24th international conference in Central Europe on computer graphics, Visualization and Computer Vision 2016, pp. 89-96.

[Can86] J. Canny, "A computational approach to edge detection". IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 8, no. 6, pp. 679-698, 1986.

[Don06] W. Donnelly, A. Lauritzen, "Variance shadow maps". In Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games, I3D'06, pp. 161-165, New York, 2006.

[Khr19] One stop solution for all Vulkan samples. https://github.com/KhronosGroup/Vulkan-Samples

[Kil12] M. Kilgard, "NVIDIA OpenGL in 2012: Version 4.3 is here!" In SIGGRAPH 2012.

[Kop07] J. Kopf, M. F. Cochen, D. Lischinski, M. Uyttendaele, "Joint bilateral upsampling". In ACM Transactions on Graphics, Volume 26, Issue3, 2007.

[Rak10] D. Rakos, "Efficient Gaussian blur with linear sampling", 2010. https://www.rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling

[Sal16] M. Salvi, "An excursion in temporal supersampling". In Game Developers Conference, 2016.

[Yan09] L. Yang, D. Nehab, P. V. Sander, P. Sitthi-Amorn, J. Lawrence, H. Hoppe, "Amortized supersampling". In ACM Trans. Graph. 28, 5 (Dec. 2009), 135:1-135:12, 2009.