White Paper

SSG

Brijender Bharti

IIR Gaussian Blur Filter Implementation using Intel® Advanced Vector Extensions

June 2010

Intel Corporation

Contents

Introduction

This white paper proposes an implementation for the Infinite Impulse Response (IIR) Gaussian blur filter [1] [2] [3] using Intel® Advanced Vector Extensions (Intel® AVX) instructions.

IIR Gaussian filter

The Gaussian filter is widely used in image processing for noise reduction, blurring, and edge detection. It is a low-pass filter and attenuates the high-frequency noise in the image. The one-dimensional Gaussian function is defined as:

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$$

where σ is the standard deviation of the Gaussian distribution.

The Fourier transform of Gaussian function is also a Gaussian:

$$G(\omega) = e^{-\frac{\omega^2\sigma^2}{2}}$$

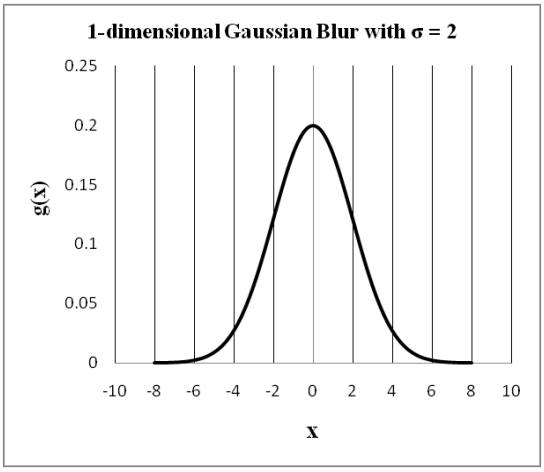


Figure 1a: 1D Gaussian Blur with α=2

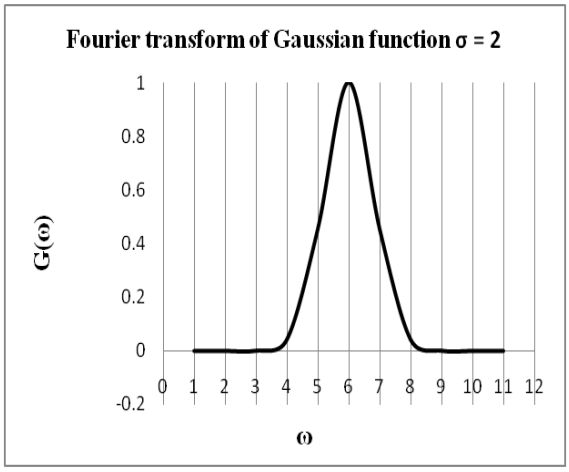


Figure 1b: Fourier transform with α=2

The two-dimensional Gaussian function is defined as:

$$g(x,y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

The Gaussian filter is very compute-intensive, as the number of operations per output pixel grows proportionally with σ. However, the IIR Gaussian filter and its derivatives [1][2][3] recursively solve a difference equation which is independent of σ, so t he number of operations per output pixel are fixed and not related to σ. The equation used in this white paper is:

$$y[n] = y'[n] + a0 * x[n] + a1 * x[n-1] - b1 * y[n-1] - b2 * y[n-2] \quad (1)$$

where $y[n]$, $y[n-1]$, $y[n-2]$ are nth, n-1th and n-2th outputs.
$y'[n]$ is nth output from previous pass.
$x[n]$, $x[n-1]$ are nth and n-1th input.
$a0$, $a1$, $b1$, $and\ b2$ are IIR Gaussian coefficients.

The IIR Gaussian filter processes each pixel horizontally and vertically. It is a separable filter; that means the filter can be applied in any order, i.e., horizontally first or vertically first.

Original Image | Blurred Image (σ =10) | Blurred Image (σ =20) | Blurred Image (σ =40)

Figure 2: IIR Gaussian Blur filter output for different σ values for 2048x2048 image

Intel® Advanced Vector Extensions

Intel® Advanced Vector Extensions (Intel® AVX) extends the capabilities of the Intel® Streaming SIMD Extensions (Intel® SSE) instruction set to new heights and dramatically increases the performance of software applications through its rich set of new functionalities. Intel® AVX supports the following:

- 256-bit wide SIMD registers. It allows processing eight single-precision or four double-precision floating-point (FP) elements in parallel, as compared to four single-precision or two double-precision FP elements in Intel® SSE. It means increased compute performance and greater energy efficiency.
- 256-bit wide AVX load allows loading and processing of 256-bit data.
- Efficient instruction encoding scheme supports three- and four-operand instruction syntax. Most legacy 128-bit SIMD instructions are also enhanced to support new instruction encoding.
- Efficient, compact and better code generation with three- and four-operand instruction syntax.
- Numerous new instructions, e.g., *broadcast* and *permute*, to manage and rearrange data.

Intel® AVX is best suited for FP-intensive computation in image processing, video processing, audio processing, scientific applications, and financial applications.

The IIR Gaussian blur filter is a compute-intensive filter. The floating point implementation of this filter produces a high-quality blurred image, which makes Intel® AVX the right candidate to implement this filter to get the best quality and performance.

IIR Gaussian Blur Implementation using Intel® AVX instructions

The IIR Gaussian blur filter applies equation (1) on each pixel through two sequential passes:

- The horizontal pass: This pass processes the input image left-to-right (row-wise), then right-to-left. The output of the left-to-right pass is added to the right-to-left pass.
- The vertical pass: Usually, the vertical pass processes the output from the horizontal pass top-to-bottom (column-wise), and then bottom-to-top. Accessing the input column-wise leads to a lot of cache blocks and impacts the performance of the filter. To avoid this, the horizontal pass transposes the output before writing to the output buffer. It makes the vertical pass similar to the horizontal pass and processes the intermediate output left-to-right, then right-to-left. The vertical pass again transposes the final output before writing the blurred image.

The proposed implementation assumes that input is 24-bit RGB packed (each color channel is represented by an 8-bit integer). For simplicity, the filter takes a symmetric image as input (height == width, e.g., 1024x1024). Since the filter processes multiple rows together and input image is symmetric, it increases the chances of bank conflicts. To avoid these bank conflicts, the filter pads each row with two cache lines. However, the filter does not process these extra cache lines, and works on original image width. The filter does not pack and unpack intermediate outputs (to 24-bit RGB pixel format) from vertical and horizontal passes, as rounding errors will add up on each pass and will produce a low-quality image. By avoiding these intermediate packs/unpacks, the filter produces a high-quality blurred image. The filter also adds an 8-bit alpha channel to the input image to make each pixel 32-bit packed (RGBA pixel format). By doing this, it is easier to align four (32-bit) pixel components (RGBA) in an AVX register. Otherwise, extra instructions (shuffles) are needed to realign the input data. The filter is written using Intel® C/C++ compiler intrinsics.

The Horizontal Pass

The wider Intel® AVX instructions allow processing of four input elements in parallel. The filter unpacks four input packed pixels from four adjacent rows in two 256-bit wide registers.

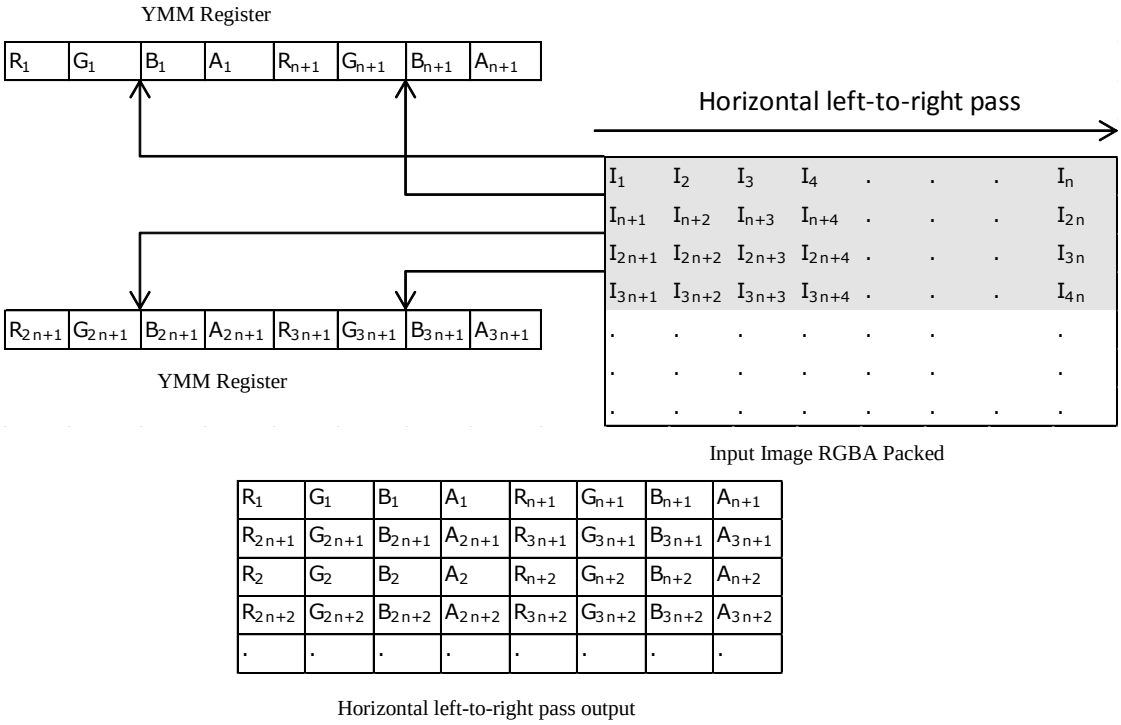


Figure 3: Horizontal left-to-right pass

The first input pixel is unpacked (using *vpmovzxbd*) to the lower 128 bits of the YMM register. The second unpacked pixel is inserted into the higher lane (upper 128 bits) of the YMM register by the new Intel® AVX instruction *vinsertf128*.

The new *vbroadcastss* instruction introduced by Intel® AVX helps to broadcast a single-precision element from memory to eight single-precision locations in the YMM register. The IIR Gaussian blur filter uses *vbroadcastss* to load IIR Gaussian coefficients into the YMM registers. Intel® AVX 256-bit add and multiply instructions allow the adding and multiplying of eight single precision floating point values (two unpacked pixels) in parallel.

The Intel® AVX instructions help to minimize cache evictions and cache blocks by processing four input pixels in parallel and producing 64 bytes (one cache line) of output. Also, the output of the left-to-right horizontal pass is stored as unpacked (floats) in a small temp buffer (width*pixels/loop*channels*float size: where pixels/loop = 4, color channels =4) instead of the full image buffer, as there are chances (depending on image size) that this temp buffer will be in the CPU cache when used in the right-to-left pass. The Intel® AVX 256-bit store instruction is used to write output data to the temp buffer.

The horizontal pass does not process extra padded cache lines in the input image. It iterates only on the actual width of the image.

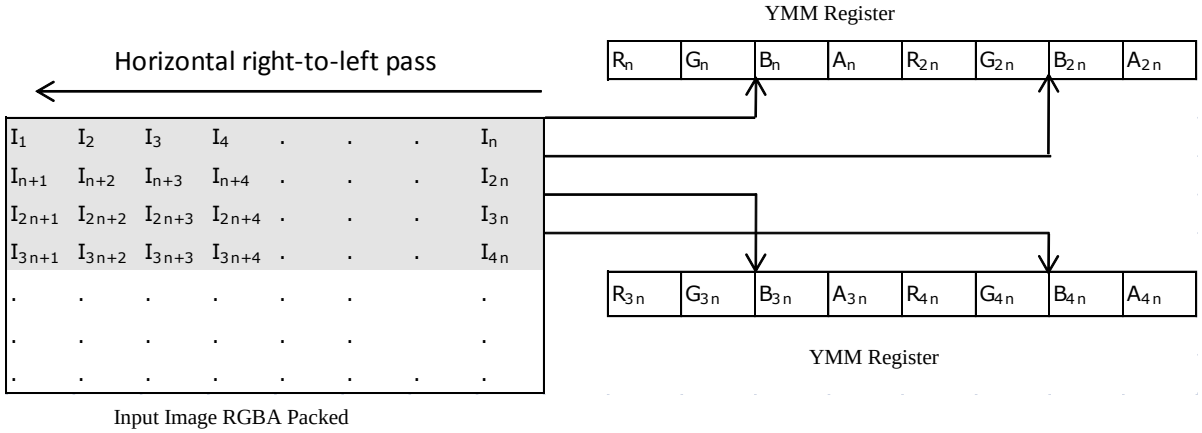
```
// width – actual input image width
// Nwidth - width after padding input image
// id – unsigned int pointer to the input image
// oTemp – temporary small ouput buffer between left-to-right and right-to-left pass
for (int y = 0; y < width; y++) {
 // Load 2 input pixels from adjacent rows
 currIn  = _mm256_castsi256_ps(_mm256_castsi128_si256(_mm_cvtepu8_epi32(*(__m128i *) (id))));
 currIn = _mm256_insertf128_ps(currIn, _mm_castsi128_ps(_mm_cvtepu8_epi32((*(__m128i *)(id+Nwidth)))), 0x1);
 currIn = _mm256_cvtepi32_ps(_mm256_castps_si256(currIn));
// Load 2 more input pixels from next adjacent rows
 currInN  =_mm256_castsi256_ps(_mm256_castsi128_si256(_mm_cvtepu8_epi32(*(__m128i *) (id+2*Nwidth))));
 currInN = _mm256_insertf128_ps(currInN, _mm_castsi128_ps(_mm_cvtepu8_epi32((*(__m128i *)(id+3*Nwidth)))),
0x1);
 currInN = _mm256_cvtepi32_ps(_mm256_castps_si256(currInN));

 currComp = _mm256_mul_ps(currIn, _mm256_broadcast_ss((float *)a0));
 currCompN = _mm256_mul_ps(currInN, _mm256_broadcast_ss((float *)a0));
 temp1 = _mm256_mul_ps(prevIn, _mm256_broadcast_ss((float *)a1));
 temp1N = _mm256_mul_ps(prevInN, _mm256_broadcast_ss((float *)a1));
 temp2 = _mm256_mul_ps(prevOut, _mm256_broadcast_ss((float *)b1));
 temp2N = _mm256_mul_ps(prevOutN, _mm256_broadcast_ss((float *)b1));
 temp3 = _mm256_mul_ps(prev2Out, _mm256_broadcast_ss((float *)b2));
 temp3N = _mm256_mul_ps(prev2OutN, _mm256_broadcast_ss((float *)b2));
 currComp = _mm256_add_ps(currComp, temp1);
 currCompN = _mm256_add_ps(currCompN, temp1N);
 temp2 = _mm256_add_ps(temp2, temp3);
 temp2N = _mm256_add_ps(temp2N, temp3N);
 prev2Out = prevOut;
 prevOut= _mm256_sub_ps(currComp, temp2);
 prev2OutN = prevOutN;
 prevOutN= _mm256_sub_ps(currCompN, temp2N);
 prevIn = currIn;
 prevInN = currInN;
  //store output data
 _mm256_store_ps((float *)(oTemp), prevOut);
 _mm256_store_ps((float *)(oTemp+8), prevOutN);

 id += 1 ; oTemp += 16;    // move to next element
}
```

Table 1: The horizontal pass left-to-right loop

The horizontal right-to-left pass is similar to the left-to-right pass, and processes the image from right to left. It unpacks four input pixels from adjacent rows using Intel® AVX instructions. The Intel® AVX 256-bit load instruction is used to load the output from the previous pass. The output of this pass is added to the output of the left-to-right pass. The final output is stored in a full image size buffer (input to the vertical pass). The output is written in transposed form and is not packed to maintain the high quality of the output image. The Intel® AVX 256-bit *vmovntps* instruction helps in streaming out the output data to memory without caching, since this output is not used in the horizontal pass anymore.



R_1	G_1	B_1	A_1	R_{n+1}	G_{n+1}	B_{n+1}	A_{n+1}	.	.	.	.
R_2	G_2	B_2	A_2	R_{n+2}	G_{n+2}	B_{n+2}	A_{n+2}	.	.	.	.
R_3	G_3	B_3	A_3	R_{n+3}	G_{n+3}	B_{n+3}	A_{n+3}	.	.	.	.
R_4	G_4	B_4	A_4	R_{n+4}	G_{n+4}	B_{n+4}	A_{n+4}	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.

Horizontal right-to-left pass output (transposed form)

Figure 4: The horizontal right-to-left pass

```
// width – actual input image width
// Nwidth - width after padding input image
// id – unsigned int pointer to the input image
// od  - float pointer to the output buffer  - input to the vertical pass
// oTemp – temporary small ouput buffer between left-to-right and right-to-left pass



for (int y = width-1; y >= 0; y--) {
 // load inputs
 inNext  = _mm256_castsi256_ps(_mm256_castsi128_si256(_mm_cvtepu8_epi32(*(__m128i *) (id))));
 inNext = _mm256_insertf128_ps(inNext, _mm_castsi128_ps(_mm_cvtepu8_epi32((*(__m128i *)(id+Nwidth)))), 0x1);
 inNext = _mm256_cvtepi32_ps(_mm256_castps_si256(inNext));

 inNextN  = _mm256_castsi256_ps(_mm256_castsi128_si256(_mm_cvtepu8_epi32(*(__m128i *) (id+2*Nwidth))));
 inNextN = _mm256_insertf128_ps(inNextN, _mm_castsi128_ps(_mm_cvtepu8_epi32((*(__m128i *)(id+3*Nwidth)))),
0x01);
 inNextN = _mm256_cvtepi32_ps(_mm256_castps_si256(inNextN));
//get the current output from temp buffer
 output = _mm256_load_ps((float *) (oTemp));
 outputN = _mm256_load_ps((float *) (oTemp+8));

 currComp = _mm256_mul_ps(currIn, _mm256_broadcast_ss((float *)a2));
 temp1 = _mm256_mul_ps(prevIn, _mm256_broadcast_ss((float *)a3));
 temp2 = _mm256_mul_ps(prevOut, _mm256_broadcast_ss((float *)b1));
 temp3 = _mm256_mul_ps(prev2Out, _mm256_broadcast_ss((float *)b2));
 currComp = _mm256_add_ps(currComp, temp1);
 temp2 = _mm256_add_ps(temp2, temp3);
 prev2Out = prevOut;
 prevOut= _mm256_sub_ps(currComp, temp2);
 prevIn = currIn;
 currIn = inNext;
```

```
 // add previous pass output
 output = _mm256_add_ps(output, prevOut);

 //next set
 currComp = _mm256_mul_ps(currInN, _mm256_broadcast_ss((float *)a2));
 temp1 = _mm256_mul_ps(prevInN, _mm256_broadcast_ss((float *)a3));
 temp2 = _mm256_mul_ps(prevOutN, _mm256_broadcast_ss((float *)b1));
 temp3 = _mm256_mul_ps(prev2OutN, _mm256_broadcast_ss((float *)b2));
 currComp = _mm256_add_ps(currComp, temp1);
 temp2 = _mm256_add_ps(temp2, temp3);
 prev2OutN = prevOutN;
 prevOutN = _mm256_sub_ps(currComp, temp2);
 prevInN = currInN;
 currInN = inNextN;
 // add previous pass output
 outputN = _mm256_add_ps(outputN, prevOutN);

 _mm256_stream_ps((float *)(od), output);
 _mm256_stream_ps((float *)(od+8), outputN);

 id -= 1; od -= 4*height;
 oTemp -=16;
}
```

Table 2: The horizontal right-to-left pass loop

The vertical pass

The vertical pass possesses more challenges with regard to memory and cache issues since the image is traversed vertically. However, the IIR Gaussian blur filter has minimized these issues by transposing the horizontal pass output (input to the vertical pass). It makes the vertical pass the same as the horizontal pass and processes the transposed input left to right first, then right to left.

The vertical left-to-right pass processes four unpacked input pixels from consecutive rows using Intel® AVX instructions. The 128-bit AVX load instruction is used to insert the first pixel into the lower half of the YMM register. The *vinsertf128* instruction is used to insert the second pixel from the next row into the upper half of the YMM register.

The Intel® AVX *vbroadcastss* instruction is used to load IIR Gaussian blur coefficients from memory. The new Intel® AVX add and multiply instructions are used to produce four unpacked pixels (16 SP elements) of output.

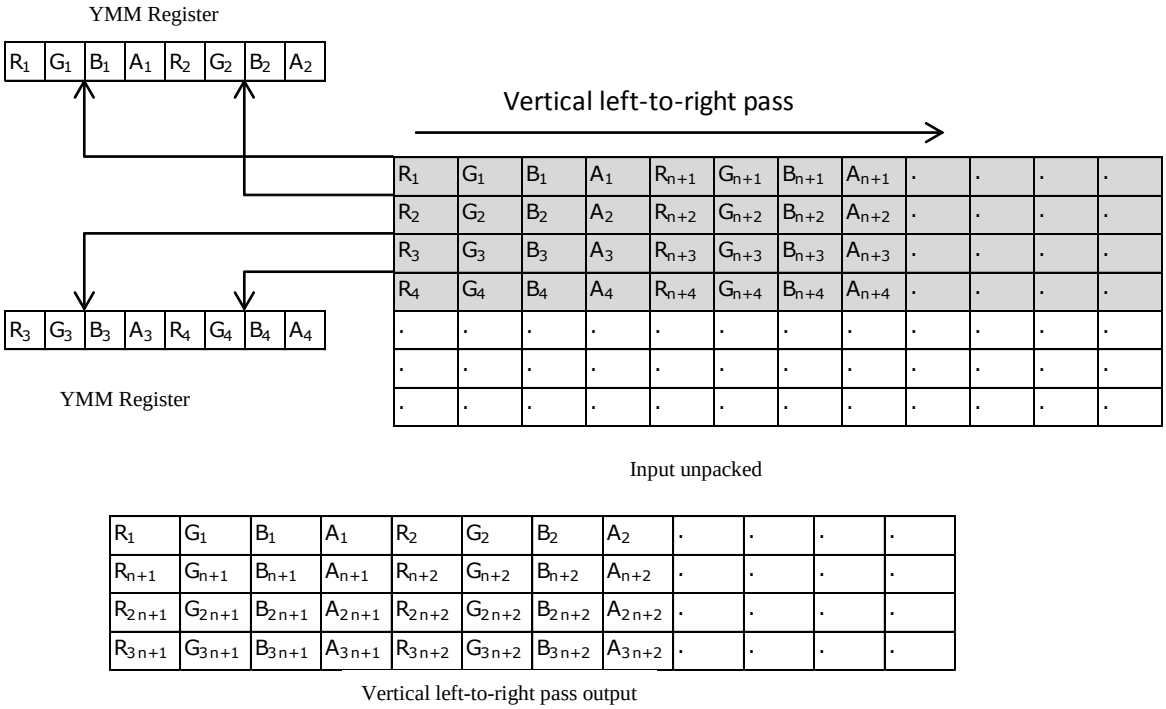


Figure 5: The vertical left-to-right pass

The vertical left-to-right pass output is stored in a small temp buffer using the Intel® AVX 256-bit store instruction. The output is transposed again (image form); that helps the vertical right-to-left pass, since this pass is going to load, process and store the output in image form.

```
// height –input image width
// id – float  pointer to the input (output from the horizontal pass)
// oTemp – temporary small ouput buffer between left-to-right and right-to-left pass

for (int y = 0; y < height; y++) {
 // load 2 input pixels
 currIn  = _mm256_castps128_ps256(_mm_loadu_ps((float *) id));
 currIn = _mm256_insertf128_ps(currIn, _mm_loadu_ps((float *)(id+4*height)), 0x1);

  // load 2 more input pixels
 currInN  = _mm256_castps128_ps256(_mm_loadu_ps((float *) (id+8*height)));
 currInN = _mm256_insertf128_ps(currInN, _mm_loadu_ps((float *)(id+12*height)), 0x1);
 //Process first 2 pixels
 currComp = _mm256_mul_ps(currIn, _mm256_broadcast_ss((float *)a0));
 temp1 = _mm256_mul_ps(prevIn, _mm256_broadcast_ss((float *)a1));
 temp2 = _mm256_mul_ps(prevOut, _mm256_broadcast_ss((float *)b1));
 temp3 = _mm256_mul_ps(prev2Out, _mm256_broadcast_ss((float *)b2));
 currComp = _mm256_add_ps(currComp, temp1);
 temp2 = _mm256_add_ps(temp2, temp3);
 prev2Out = prevOut;
 prevOut= _mm256_sub_ps(currComp, temp2);
 prevIn = currIn;
 // Store output in temp buffer
 _mm256_storeu_ps((float *)(oTemp), prevOut);

 //Process next 2 pixels
 currComp = _mm256_mul_ps(currInN, _mm256_broadcast_ss((float *)a0));
 temp1 = _mm256_mul_ps(prevInN, _mm256_broadcast_ss((float *)a1));
 temp2 = _mm256_mul_ps(prevOutN, _mm256_broadcast_ss((float *)b1));
 temp3 = _mm256_mul_ps(prev2OutN, _mm256_broadcast_ss((float *)b2));
 currComp = _mm256_add_ps(currComp, temp1);
 temp2 = _mm256_add_ps(temp2, temp3);
```

```
 prev2OutN = prevOutN;
 prevOutN= _mm256_sub_ps(currComp, temp2);
 prevInN = currInN;
 _mm256_storeu_ps((float *)(oTemp+8), prevOutN);

 id += 4; oTemp += 16;    // move to next row
} //height = j loop
```

Table 3: The vertical left-to-right pass loop

The vertical right-to-left pass is similar to the left-to-right pass. It processes the horizontal pass output from the last pixel of each row towards the first pixel of that row. The 256-bit load instruction is used to load the output of the previous pass. The output computed in this pass is added to the output of the left-to-right vertical pass. The final output is packed to 32-bit RGBA form. The vertical right-to-left pass writes four packed pixels per iteration.

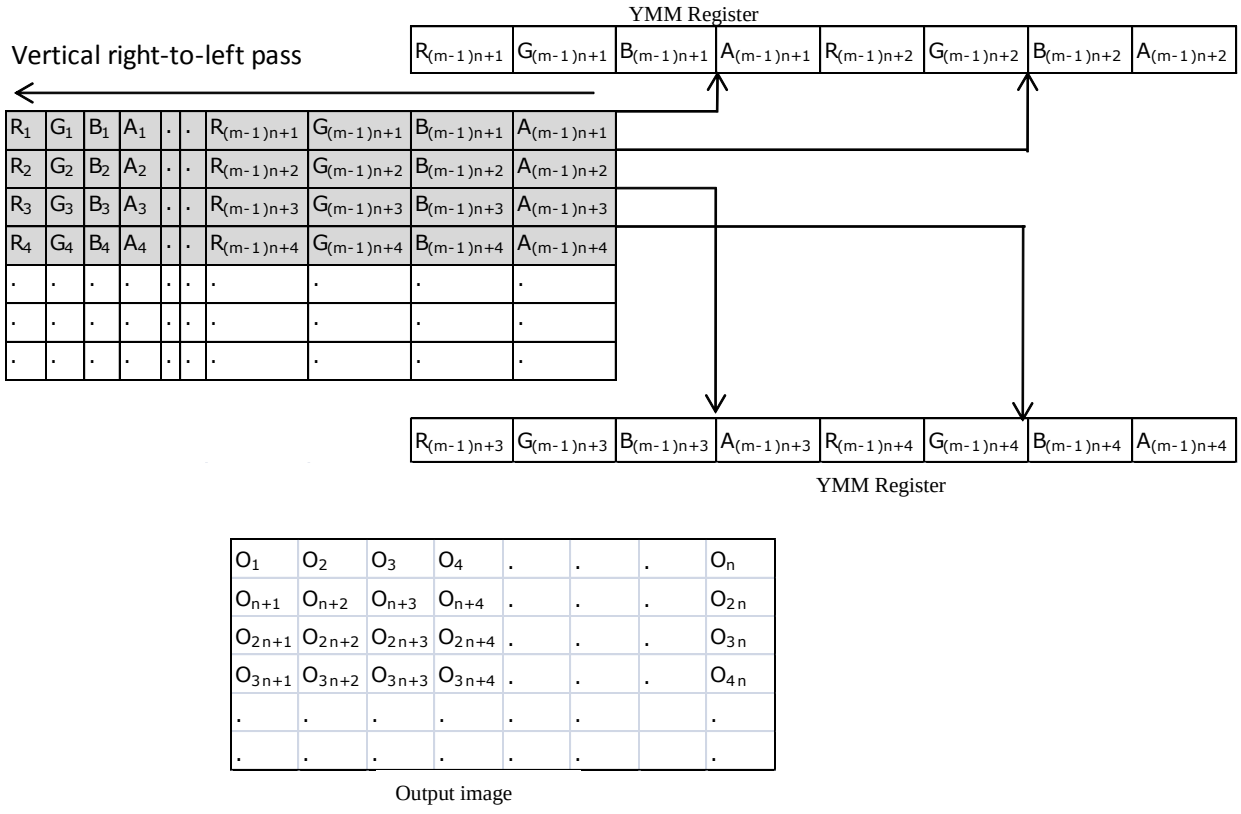


Figure 6: The vertical right-to-left pass

```
// height –input image width
// id – float  pointer to the input (output from the horizontal pass)
// od  - float pointer to the output buffer  - input to the vertical pass
// oTemp – temporary small ouput buffer between left-to-right and right-to-left pass

for (int y = height-1; y >= 0; y--) {
 // load 2 input pixel
 inNext  = _mm256_castps128_ps256(_mm_loadu_ps((float *) id));
 inNext = _mm256_insertf128_ps(inNext, _mm_loadu_ps((float *)(id+4*height)), 0x1);

 // load 2 more input pixel
 inNextN  = _mm256_castps128_ps256(_mm_loadu_ps((float *) (id+8*height)));
 inNextN = _mm256_insertf128_ps(inNextN, _mm_loadu_ps((float *)(id+12*height)), 0x1);

 //get output from left to right pass
 output = _mm256_load_ps((float *) (oTemp));

 currComp = _mm256_mul_ps(currIn, _mm256_broadcast_ss((float *)a2));
 temp1 = _mm256_mul_ps(prevIn, _mm256_broadcast_ss((float *)a3));
 temp2 = _mm256_mul_ps(prevOut, _mm256_broadcast_ss((float *)b1));
 temp3 = _mm256_mul_ps(prev2Out, _mm256_broadcast_ss((float *)b2));
 currComp = _mm256_add_ps(currComp, temp1);
 temp2 = _mm256_add_ps(temp2, temp3);
 prev2Out = prevOut;
 prevOut= _mm256_sub_ps(currComp, temp2);
 prevIn = currIn;
 currIn = inNext;

 //add currently computed output to left-to-right pass output
 output = _mm256_add_ps(output, prevOut);
 output  = _mm256_castsi256_ps(_mm256_cvttps_epi32(output));

 __m128i output2 =  _mm_castps_si128(_mm256_extractf128_ps(output, 1));
 __m128i  output1 = _mm_castps_si128(_mm256_castps256_ps128(output));

 //get the left-to-right pass output
 outputN = _mm256_loadu_ps((float *) (oTemp+8));

 currComp = _mm256_mul_ps(currInN, _mm256_broadcast_ss((float *)a2));
 temp1 = _mm256_mul_ps(prevInN, _mm256_broadcast_ss((float *)a3));
 temp2 = _mm256_mul_ps(prevOutN, _mm256_broadcast_ss((float *)b1));
 temp3 = _mm256_mul_ps(prev2OutN, _mm256_broadcast_ss((float *)b2));
 currComp = _mm256_add_ps(currComp, temp1);
 temp2 = _mm256_add_ps(temp2, temp3);
 prev2OutN = prevOutN;
 prevOutN= _mm256_sub_ps(currComp, temp2);
 prevInN = currInN;
 currInN = inNextN;
 outputN = _mm256_add_ps(outputN, prevOutN);
 outputN  = _mm256_castsi256_ps(_mm256_cvttps_epi32(outputN));

 //convert to RGBA
 __m128i  output4 = _mm_castps_si128(_mm256_extractf128_ps(outputN, 1));
 __m128i  output3 = _mm_castps_si128(_mm256_castps256_ps128(outputN));

 output1 = _mm_packus_epi32(output1, output2);
 output3 = _mm_packus_epi32(output3, output4);
 output1 = _mm_packus_epi16(output1, output3);

 _mm_store_si128((__m128i *)(od), output1);
 id -= 4; od -= width; oTemp -=16; // move to previous row
} //height = j loop
```

Table 4: The vertical right-to-left pass loop

Tools

The IIR Gaussian blur filter is implemented using Intel® C/C++ compiler intrinsics. The Intel® C/C++ compiler intrinsics are listed in the Intel® Advanced Vector Extensions Programming Reference. The filter can be compiled using the Intel® C/C++ Compiler 11.1 or later versions.

The Intel® C/C++ compiler supports automatic vectorization, profile-guided optimization, and multithreaded application support through Intel® Threading Building Blocks and OpenMP*. The OpenMP APIs are used to thread the IIR Gaussian blur filter.
The Intel® AVX website (http://software.intel.com/en-us/avx/) provides links to help, develop, emulate and analyze software written using Intel® AVX instructions. The Intel® Software Development Emulator (Intel® SDE) can be used to emulate the IIR Gaussian blur filter on computers that do not support Intel®AVX (prior to Intel® microarchitecture codename Sandy Bridge).

Quality Analysis

The IIR Gaussian blur filter maintains the highest image quality by avoiding intermediate packing and unpacking of the output between the passes. The packing and unpacking in each pass introduces rounding errors. As measured for a 2048x2048 blurred image for $\sigma = 40$, if the output of each pass is packed (versus not packed), the root mean square error and signal-to-noise ratio are as follows:

	R-Channel	G-Channel	B-Channel
Root Mean Square Error	1.6	1.6	1.6
Signal-to-Noise Ratio (dB)	38.0	36.3	36.5

Performance Analysis

The IIR Gaussian blur filter's performance using Intel® AVX versus Intel® SSE for large numbers of iterations is as follows:

Image Size	Intel® SSE (cycles/pixel)	Intel® AVX (cycles/pixel)	Intel® AVX Performance Gain over Intel® SSE
512x512	7.5	5.8	1.30x
1024x1024	11.5	8.5	1.35x
2048x2048	17.1	8.9	1.92x

The IIR Gaussian blur filter's performance depends on memory bandwidth, memory latency, and cache effects (cache blocks, cache evictions, cache replacements, and bank conflicts). Intel® AVX instructions help to minimize memory and cache effects by processing four pixels in parallel with a minimum set of instructions. The new Intel® AVX 256-bit load instructions help to bring more data to the CPU, and allow processing of one cache line of data (64 bytes – 4 unpacked pixels) at a time which helps to reduce cache blocks.
For small images/frames (512x512), the Intel® AVX performance is 5.8 cycles/pixel, which is 1.3x faster than the Intel® SSE implementation. Since in this case, the whole data (input/intermediate/output) will mostly fit in the CPU cache, the Intel® AVX performance gain is due to wider registers and fewer instructions. For higher resolution images (2048x2048), the Intel® AVX performance is as high as 1.92x as compared with the Intel® SSE implementation.

This performance gain is due to the fact that Intel® AVX implementation is able to load, process and store two times more data (2x wider registers and loads), which leads to fewer cache blocks and better memory bandwidth usage.

Conclusion

The Intel® AVX-based IIR Gaussian blur filter provides better performance as compared with Intel® SSE-based implementations. The key features of Intel® AVX — the 256-bit wider registers, 256-bit loads, fewer instructions (due to three-operand format), and efficient encoding — helped to get better performance. The implementation also shows the ease with which existing floating-point code can be ported to Intel® AVX.

Alternate Implementations

The Intel® AVX based IIR Gaussian filter implementation can be modified to achieve better performance as follows:

Pack/unpack intermediate data

The proposed implementation maintains the highest quality output by not packing/unpacking output between the passes, to avoid the rounding factor. The output between the vertical pass and the horizontal pass is a full-size unpacked image. If the highest quality output is not desired, Intel® AVX instructions can help to achieve better performance by passing compact packed output between the passes. It will add extra processing to the passes, but will give better performance for higher resolution frames/images where memory bandwidth is bottlenecked.

Processing smaller frames/images (512x512)

Since the proposed implementation gives the best performance for small frames/images (512x512), bigger frames/images can be divided into smaller frames/images, and these small frames can be processed in parallel. Each small frame needs a few extra pixels in input due to the IIR nature of the algorithm. It will require additional processing for extra pixels, but that overhead is low.

References

[1] Rachid Deriche – "Recursively implementing the Gaussian and its derivatives", 1993.
[2] Lucas J. van Vliet, Ian T. Young and Piet W. Verbeek – "Recursive Gaussian derivative Filters", 1998
[3] Dave Hale, "Recursive Gaussian Filters", CWP-546
[4] Luis Alvarez, Rachid Deriche, Francisco Santana – "Recursivity and PDE's in Image Processing", 2000