FAST & ROBUST DENOISING USING FEATURE AND COLOR INFORMATION

Alexandre Poirrier, Alexandre Binninger, Félicité Lordon–de Bonniol du Trémont, Nino Scherrer
{apoirrier, binninga, ftremont, ninos}@student.ethz.ch

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Denoising has established itself as an important technique for the efficient synthesis of realistic, high resolution rendered images. We present a fast and adaptive implementation of a joint filtering algorithm based on color and feature information to denoise Monte-Carlo renderings. Based on a detailed performance analysis of a straightforward implementation, we introduce an efficient, restructured implementation including vectorization. By introducing box filtering, eliminating long latency dependencies and applying general optimization techniques we report a vastly improved runtime by a factor of $27.5\times$ (scalar) / $62.5\times$ (vectorized). In addition we provide a continuous profiling / roofline analysis for all relevant steps of optimization.

1. INTRODUCTION

Rendering is the process of creating a photorealistic image from a given 3D scene. This problem can be formulated using the rendering equation [1] as a multidimensional integral that models the physical process of light transport from all light sources to the image sensor. Monte-Carlo (MC) methods [2] are widely used to estimate this complex integral through random light path samples. Such methods form consistent estimators of the correct solution but unfortunately converge only at a rate of $\mathcal{O}(1/\sqrt{n})$. Meaning we get an approximation error of the integral for a low number of samples which produces noise in the final image. While the emergence of sophisticated sampling strategies (e.g. stratified sampling, importance sampling) could reduce the noise, it is still almost impossible to obtain noise-free renderings in acceptable time due to the high computation cost of MC sampling. The aforementioned limitation led to the intensified research of a-posteriori denoising techniques for MC renderings. They are capable of producing high-quality, nearly noise-free renderings at negligible cost compared to an increased number of samples.

Motivation. The algorithm presented in this paper aims at removing the noise from MC renderings. Such renderings are heavily used in the film, video game and special effect industries. As animated movies can be composed of thousands of renderings, fast denoising is of crucial importance as a lot of data needs to be processed. For modern-day applications like video gaming, fast denoising is even more important as denoising needs to happen in realtime.

Related work. Due to the importance of denoising MC renderings, a lot of algorithms have been published in the literature. Some papers such as [3] use general image denoising algorithms to denoise MC renderings. Most of them are based on image space filtering techniques adapted to MC renderings, such as [4] or [5]. Those techniques make mostly use of simple filters (such as Gaussian kernels) for denoising. There are also more complex approaches such as [6], which computes wavelets from the image variance to get a precise and efficient denoising. [7] goes even further and gets additional quality improvements by using inexpensive by-products (such as albedo, depth and normal buffers) of the rendering process in a joint filtering scheme. For a more detailed review we refer to [8]. While the mentioned algorithms date back to 2014 and earlier, follow-up work mostly focused on first order regression models [9] and learning-based techniques using convolutional neural networks [10, 11].

Contribution. This paper presents a fast and adaptive implementation of the MC denoising algorithm offered in [7]. While this algorithm offers robust denoising, it uses additional feature buffers, which makes fast implementations hard to get. As there are numerous data buffers, data will most likely not hold in cache memory as the algorithm processes all information. Moreover, the algorithm uses long-latency operations. It also requires a lot of computations with linear dependency. This makes it difficult to achieve an efficient scheduling of operations without bubbles in the CPU pipeline. The remaining of this paper presents in stepwise manner how the algorithm has been improved to get a more efficient denoising by introducing box-filtering, eliminating linear dependencies and applying general optimization techniques (incl. `AVX2` vectorization). The detailed analysis presents how the optimizations make better use of the memory hierarchy and the CPU pipeline. The final outlook provides an insight into further possible optimizations.

2. ROBUST DENOISING USING COLOR AND FEATURE INFORMATION

This section formally introduces the MC denoising algorithm which serves as a foundation for the following optimizations. It is based on the work of [7] and is modified with some small simplifying assumptions. It is completed with a subsequent cost analysis of a straightforward implementation of the given algorithm.

2.1. Denoising algorithm

The main idea of the presented denoising algorithm, illustrated in Figure 1, is the construction of a filter by blending both color and feature information. In order to achieve this, three different candidate filters (denoted as FIRST, SECOND and THIRD) are constructed using the same filtering mechanism but with different sensitivities to color and feature information. While the FIRST filter is most sensitive to details in the color buffer, it is also most sensitive to its noise. The SECOND filter is constructed in such a way that it treats color and feature information in a balanced way. The THIRD filter only uses features information, therefore it is not affected by the noise of the render but does not use the details produced by MC rendering. The candidate filters are finally averaged to a final filter using a Stein's unbiased risk estimate (SURE) based per-pixel error estimate.

General Input. The algorithm takes as input a color buffer c: the image to denoise, feature buffers (albedo, normal and depth) f and their variance buffers c_{var} and f_{var}. In addition, it takes a neighborhood parameter R. While c is the general output of a MC rendering system, f and the variances c_{var}, f_{var} are inexpensive by-products which can be extracted from the rendering process.

Filtering Mechanism. The main idea of the filtering mechanism is to compute a weighted average of neighboring pixels. The filtered color values $F(p)$ of a pixel p in a color image $u(p)$ is computed as:

$$F(p) = \frac{1}{C(p)} \sum_{q \in N(p)} u(q) w(p,q)$$
$$\iff \quad \begin{bmatrix} F_1(p) \\ F_2(p) \\ F_3(p) \end{bmatrix} = \frac{1}{C(p)} \sum_{q \in N(p)} \begin{bmatrix} u_1(q) \\ u_2(q) \\ u_3(q) \end{bmatrix} w(p,q) \tag{1}$$

where $N(p)$ is a square neighborhood of size $(2r+1) \times (2r+1)$ centered on p and $C(p) = \sum_{q \in N(p)} w(p,q)$ denotes a weight normalization factor. The main challenge of the algorithm is to determine suitable weights and compute them efficiently. Weights are separated into two components: color weights w_c and feature weights w_f, where the final weight is defined as $w(p,q) = min(w_c(p,q), w_f(p,q))$.

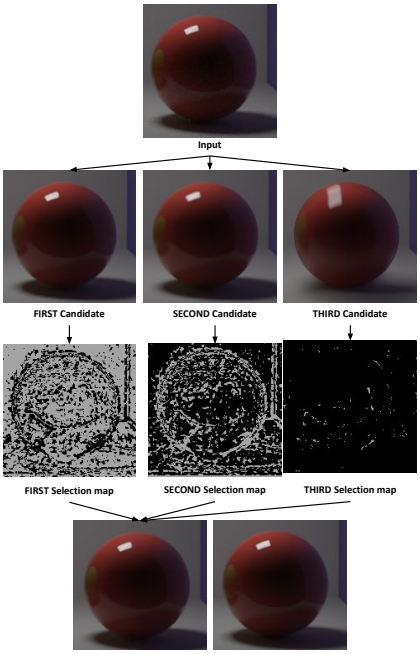

Fig. 1. Illustration of the blending of candidate filters with different sensitivities to color and feature information using SURE-based selection maps

Color Weight Computation. Color weights are based on Non-Local means (NL-means) filtering [12]. The weight $w(p,q)$ for a pixel p and its neighbor q is calculated based on a distance function $d_c^2(P(p), P(q))$ which is defined as:

$$d_c^2(P(p), P(q)) = \frac{1}{3(2f+1)^2} \sum_{i=1}^{3} \sum_{n \in P_f} \Delta_i^2(p+n, q+n)$$

where $P(p)$ and $P(q)$ define a small patch of size $(2f+1) \times (2f+1)$ centered at p and q. The offset to each neighbor within a patch is denoted as $n \in P_f$ and $\Delta_i^2(p+n, q+n)$ denotes the per-pixel difference in color channel i and is defined as:

$$\Delta_i^2(p,q) = \frac{(u_i(p) - u_i(q))^2 - (\mathrm{Var}_i[p] + \mathrm{Var}_i[q,p])}{\varepsilon + k_c^2 (\mathrm{Var}_i[p] + \mathrm{Var}_i[q])}$$

where k_c is a variable to control the sensitivity of the filter to color differences. $\mathrm{Var}_i[p]$ denotes the variance estimate for the sample mean in pixel p and $\mathrm{Var}_i[p,q]$ is defined as $\mathrm{Var}_i[p,q] = \min(\mathrm{Var}_i[q], \mathrm{Var}_i[p])$. Finally the color weight is computed using an exponential kernel as:

$$w_c(p,q) = \exp^{-\max\left(0, d_c^2(P(p), P(q))\right)}$$

Feature Weight Computation. Feature weights are computed in a similar fashion as color weights but by using bilateral weights. We define a per-pixel distance $\Phi_j^2(p,q)$ for

feature j between pixel p and q:

$$\Phi_j^2(p,q) = \frac{(f_j(p) - f_j(q))^2 - (\mathrm{Var}_j[p] + \mathrm{Var}_j[p,q])}{k_f^2 \max\left(\tau, \max\left(\mathrm{Var}_j[p], \|\mathrm{Grad}_j[p]\|^2\right)\right)}$$

where k_f is a variable to control the sensitivity of the filter to features differences and τ denotes a threshold value. The overall feature distance $d_f^2(p,q)$ function and the final feature weight $w_f(p,q)$ are defined as:

$$d_f^2(p,q) = \underset{j \in [1 \ldots M]}{\mathrm{argmax}}\ \Phi_j^2(p,q)$$

$$w_f(p,q) = \exp^{-d_f^2(p,q)}$$

SURE Error Estimate. Let us denote the output of the filter at pixel p for a given noisy value u_i as $F_{u_i}(p)$. The SURE error at pixel p is defined as:

$$\mathrm{SURE}(p) = \sum_{i=1}^{3} \|F_{u_i}(p) - u_i\|^2 - \sigma_i^2$$

where σ_i^2 denotes the true variance of the pixel mean of color channel i . In contrast to the work of [7], we eliminate the derivative term of the filtered value to save an additional computation routine and data buffer. We are aware that this may lead to weaker denoising performance.

Overall Algorithm. Based on the introduced filtering mechanism, its corresponding weight computations and the SURE error estimate, we assemble the overall algorithm which is illustrated in Algorithm 1. For better accessibility to the filtering mechanism and inspired by [7], we formally define a filtering functions as:

$$out = \mathrm{flt}(in, u, var\_u, f, var\_f, p)$$

A given input in is filtered according to equation 1 by constructing color weights from a color buffer u and its corresponding variance buffer var_u, and feature weights from a given feature buffer f with its corresponding variance buffer var_f. The filtering parameters (i.e. k_c, k_f, f, r and τ) are encapsulated in a parameter structure p. In a similar fashion we define the SURE error estimator function as:

$$out\_sure = SURE(u, var\_u, F)$$

which computes the error estimate based on a noisy buffer u, its corresponding sample variance var_u and its filtered version F.

In contrast to [7] the sample variance estimation/scaling step is eliminated. Since our input renderings rely on random sampling, the variances $\mathrm{Var_i}$ of sample means can be therefore estimated through sample variance within each pixel. For more in-depth information about the robust MC denoising algorithm we refer the interested reader to [7].

Algorithm 1: Robust Denoising

Input: Noisy color buffer c with its sample variance var_c, noisy feature buffer f with its sample variance var_f and window radius R
Output: Denoised image out

```
   /* (1) Feature Prefiltering                          */
 1 p = { k_c = 1, k_f = ∞, f = 3, r = 5 }
 2 flt_f = flt(f, f, var_f, nil, nil, p)
 3 flt_var_f = flt(var_f, f, var_f, nil, nil, p)
   /* (2) Candidate Filters                             */
 4 p = { k_c = 0.45, k_f = 0.6, f = 1, r = R, τ = 0.001 }
 5 r = flt(c, c, var_c, flt_f, flt_var_f, p)
 6 p = { k_c = 0.45, k_f = 0.6, f = 3, r = R, τ = 0.001 }
 7 g = flt(c, c, var_c, flt_f, flt_var_f, p)
 8 p = { k_c = ∞, k_f = 0.6, f = 1, r = R, τ = 0.0001 }
 9 b = flt(c, c, var_c, flt_f, flt_var_f, p)
   /* (3) Filtered SURE Error Estimates                 */
10 p = { k_c = 1.0, k_f = ∞, f = 1, r = 1, τ = 0.001 }
11 e_r = flt(SURE(c, var_c, r), c, var_c, nil, nil, p)
12 e_g = flt(SURE(c, var_c, g), c, var_c, nil, nil, p)
13 e_b = flt(SURE(c, var_c, b), c, var_c, nil, nil, p)
   /* (4) Binary Selection Maps                         */
14 sel_r = e_r < e_g && e_r < e_b ? 1 : 0
15 sel_g = e_g < e_r && e_g < e_b ? 1 : 0
16 sel_b = e_b < e_r && e_g < e_b ? 1 : 0
   /* (5) Filter Selection Maps                         */
17 p = { k_c = 1.0, k_f = ∞, f = 1, r = 5, τ = 0.001 }
18 sel_r = flt(sel_r, c, var_c, nil, nil, p)
19 sel_g = flt(sel_g, c, var_c, nil, nil, p)
20 sel_b = flt(sel_b, c, var_c, nil, nil, p)
   /* (5) Candidate Filter Averaging                    */
21 out = r * sel_r + b * sel_b + g * sel_g
```

2.2. Cost Analysis

For a computational cost analysis of the given denoising algorithm, we define the cost measure as:

$$\begin{aligned} C(n) = {} & C_{\mathrm{ADD}} * N_{\mathrm{ADD}}(n) + C_{\mathrm{MUL}} * N_{\mathrm{MUL}}(n) \\ & + C_{\mathrm{DIV}} * N_{\mathrm{DIV}}(n) + C_{\mathrm{EXP}} * N_{\mathrm{EXP}}(n) \\ & + C_{\mathrm{MIN/MAX}} * N_{\mathrm{MIN/MAX}}(n) \end{aligned}$$

where n denotes the height/width of the input image. As simplifying assumption we only consider images of square size. The exact execution number of the individual operations N_{OP} was counted based on the straightforward implementation which is further described in section 3. Due to the complex nature of the exact operation counts we provide an illustration in figure 2 for a fixed window radius of $R = 10$ and general filtering parameters as chosen in Algorithm 1. For completeness we provide the exact formulas in the supplementary section 8.1, as well as measurements performed for varying R justifying our choice of $R = 10$ in section 8.4. As one can see in figure 2, additions ($\approx 50\%$) and multiplications ($\approx 25\%$) are the predominant opera-

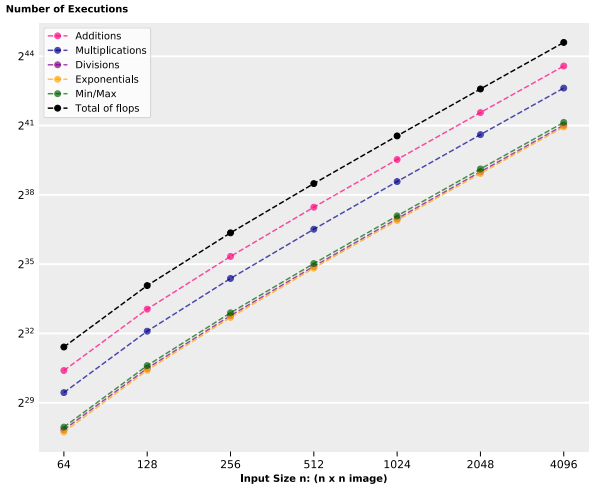

Fig. 2. Illustration of the execution count of relevant operations for a fixed window radius of $R = 10$

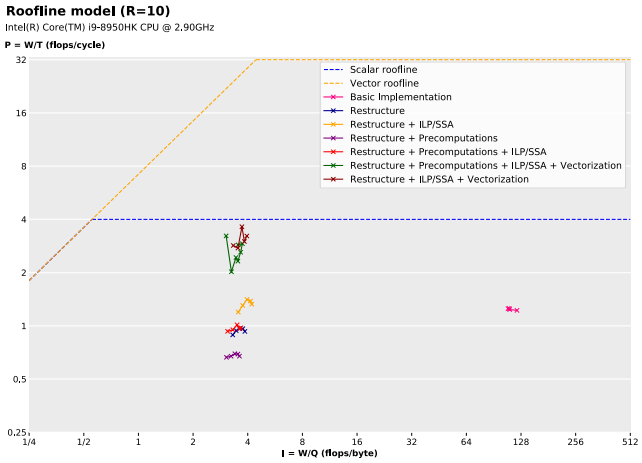

Fig. 3. Roofline model for a fixed window radius of $R = 10$. Data movement was measured with Valgrind, Work was calculated by in-code flop counting. Each point represents a different input size ranging from 64×64 to 512×512. Note that the roofline has been plotted using the specifications given by the Intel manual. However, the actual maximum performance peak is most likely smaller due to the instruction mix, as well as the actual achievable bandwidth.

tions with respect to the number of executions. Unfortunately there is also a considerably fraction of long-latency operations such as divisions ($\approx 8.5\%$) and exponentiations ($\approx 8\%$). It is known that those operations take considerably longer to compute than floating point addition and multiplication and should be avoided as much as possible. Based on recent measurements of Agner Fog [13] and the computational complexity of the individual operations, we can state:

$$C_{\texttt{EXP}} \gg C_{\texttt{DIV}} > C_{\texttt{MIN/MAX}} > C_{\texttt{MUL}} > C_{\texttt{ADD}}$$

Without taking the exact costs of the operation into account, we can summarize the asymptotic computational complexity as $\mathcal{O}(n^2 * R^2)$.

3. ALGORITHM RESTRUCTURING / OPTIMIZATIONS

This section introduces all relevant steps of optimization in a step-wise manner. We start by introducing a basic implementation which is following closely the given formulas in section 2. It is succeeded by a restructured implementation which vastly reduces the operation count by rearranging the overall order of the computation in such a way that efficient box-filtering can be leveraged. It further aims to eliminate long latency operations and dependencies to minimize the overall cost. In subsequent steps, we highlight how weight-sharing can be used to to further decrease the computational cost, and how general optimization techniques can be applied to increase instruction level parallelism (ILP). Finally, we present vectorization using `AVX2` and introduce a blocking technique to reduce data movement.

Throughout this section, the reader may refer to figure 3 to follow the continuous roofline analysis for all relevant code versions. It illustrates the different implementations on a roofline plot [14] compared to the scalar and vectorized rooflines. This plot evaluates how a code version performs respectively to the maximum attainable performance on the test machine by also taking data movement into account. It allows to infer memory or compute boundness and to deduce upper bounds on possible performance improvements. Therefore it serves as a guide through the presented steps of optimizations.

Basic Implementation. This straightforward implementation is closely following the formulas in section 2. Input and intermediate buffers are stored within a three-dimensional memory-structure with a [channel, width, height] layout. Since the algorithm is dominated by filtering, we provide an analysis of a filtering procedure where color and feature information are taken into account. For every pixel p in the noisy input image, the filtering mechanism is applied to compute the filtered value $F(p)$. Meaning for each pixel q in the defined neighborhood, the weight $w(p, q)$ is computed based on its color weight $w_c(p, q)$ and feature weight $w_f(p, q)$. Finally the contribution is accumulated to $F(p)$. While the color weight is additionally dependent on its patch region, the feature weight is only dependent on

pixels p and q. After covering all neighboring pixels, the filtered value $F(p)$ is normalized and the procedure moves on to the next pixel as outlined in Algorithm 2. Since neighboring pixels share trivially a big fraction of the neighborhood, the illustrated filtering routine is showing desirable properties in terms of spatial and temporal locality.

The filtering procedure is executed eleven times in the complete denoising algorithm. While multiple filtering calls operate on a small neighborhood ($R \leq 5$), candidate filtering operates on a neighborhood defined by the window size R. Since denoising quality is mainly dependent on the window size, R is likely to be set bigger than 10. Thus, the color weights computation in candidate filtering is clearly dominating the runtime, with a computational complexity of $\mathcal{O}(W * H * (2R+1)^2 * (2f+1)^2)$ (where $f \leq 3$ is the patch size). A hotspot analysis using Intel VTune confirmed this bottleneck.

On the roofline model in figure 3, the basic implementation is characterized by a high operational intensity and is definitely compute bound. Despite the desirable locality properties, the compute boundness and the previous identified bottleneck motivates the restructuring of the algorithm to reduce the computational complexity the filtering mechanism.

Algorithm 2: Basic - Filtering

```
   /* Covering all pixels                                  */
 1 for Pixel p do
       /* Covering neighbourhood N(p)                      */
 2     for q in N(p) do
           /* Covering all color channels                  */
 3         for i = 1 to 3 do
               /* Covering patch P_f                       */
 4             for n in P_f do
 5                 Compute Δ_i(p+n, q+n) ;
 6                 w_c(p,q) += Δ_i(p+n, q+n);
 7         Normalize and exp of w_c(p,q);
 8         Compute w_f(p,q);
 9         w := min(w_c, w_f);
10         F(p) += input(q) * w(p,q);
11     Normalization of F(p);
```

Restructured Implementation. Due to the identified compute boundness and the revealed bottleneck, this implementation aims at reducing the overall computational complexity. Its key idea is to restructure the filtering operation to allow efficient box-filtering over the complete image space for computing color weights. In addition, long latency operations such as division or exponentiation are eliminated on the go as much as possible by mathematical simplifications. Scalar replacement is also directly integrated.

By analyzing the color weight computation of the basic implementation, it becomes apparent that per-pixel differences Δ_i are computed multiple times and it is performing some sort of naive $3D$ box-filtering over the per-pixel differences on local regions. The key insight is that this computation routine can be extended to a box-filtering approach over the complete image space. Therefore we can exploit the kernel separability, as indicated below, and perform two $1D$ convolutions instead of one $3D$ convolution.

$$\frac{1}{3(2f+1)}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix} = \frac{1}{3(2f+1)}\begin{bmatrix}1\\1\\1\end{bmatrix} * \begin{bmatrix}1&1&1\end{bmatrix}$$

In order to allow those efficient operations, the algorithm needs to be restructured heavily. Instead of processing the image pixel by pixel, a neighborhood translation is fixed and the complete image is processed for this specific shift, as illustrated in Algorithm 3. In contrast to the basic implementation, data are traversed multiple times during filtering which results in reduced temporal locality. Although, the general data accesses pattern is very linear which improves spatial locality.

While the basic implementation is computing $\mathcal{O}(W * H * (2R+1)^2 * (2f+1)^2)$ per-pixel differences, this restructured implementation is only computing $\mathcal{O}(W * H * (2R+1)^2)$ per-pixel differences since they can be reused multiple times in the neighborhood. By using the introduced kernel-separability, the overall computational complexity of the filtering can be reduce to $\mathcal{O}(W * H * (2R+1)^2)$. For a fixed window radius of $R = 10$ and default filtering parameters, it results in a floating point operation reduction of roughly 15x. For an illustration of the resulting reduction we refer to the supplementary section 8.3.

On the roofline model in figure 3 it is clearly visible that the restructuring is reducing the operational intensity. However, it is still compute-bound. In addition, the amount of expensive exponentations is reduced to 1.6% of the total executed operations and profiling revealed a reduction of 50% in the utilization of the divider computation unit.

Precomputations. After the heavy restructuring of the filtering procedure, this code version aims to exploit shared computations for a further reduction of operations.

A detailed analysis of the algorithm revealed, that following computations can be shared: (1) weights for prefiltering the feature buffers and their corresponding variances, (2) per-pixel distances of candidate filters FIRST and SECOND and (3) feature weights of candidate filters FIRST and THIRD.

The exploited precomputations result in an additional floating point operation reduction of roughly 1.7x for a fixed

Algorithm 3: Restructure - Filtering

```
   /* Covering all neighborhood shifts                    */
1  for r_x = −r to r do
2      for r_y = −r to r do
           /* Computing per-pixel diff.                   */
3          for p = (0,0) to (W,H) do
4              q = p + [r_x; r_y];
5              for i = 1 to 3 do
6                  temp(p) += Δ_i(p,q);
           /* Computing feature weights                   */
7          for p = (0,0) to (W,H) do
8              q = p + [r_x; r_y];
9              wf(p) = w_f(p,q);
           /* Compute Patch Contribution                  */
10         wc = conv(temp, box_f);
           /* Add Contribution                            */
11         w(p) = min(wc(p), wf(p));
12         F(p) += input(q) * w(q);
13 Overall Weight Normalization;
```

window size $R = 10$ and default filtering parameters. In addition, the roofline model in figure 3 reveals no significant change of the operational intensity and shows that we are still compute-bound.

Scalar Optimizations. The next step of optimization aims at increasing the CPU port usage to increase performance. More precisely, computations consist of long sequences of operations with linear dependency. This is quite costly, especially for high latency operations such as division and exponentiation.

Linear dependencies are broken by using loop unrolling and accumulators. This technique allows to process independent operations per group, using the full capabilities of the CPU ports, before reducing the final operation. To implement this, as loop bodies operate on independent data, loops are unrolled and accumulators are being used to store simultaneously the results of different independent operations. Moreover, Single Static Assignment (SSA) is used to get further optimizations from the compiler. Since finding correct parameters for accumulator and unrolling factors for increased ILP is quite tedious by hand, an autotuning script has been developed. This script, written in Python, instruments the C code to produce different versions of the chosen loop to unroll. Then it automatically compiles the code and performs different tests on our benchmarking platform to select the best version. Despite this tool is not fully automatized and cannot guarantee the identification of the global optima, it considerably ease the ILP work.

The roofline model on figure 3 shows that while operational intensity remains similar to previous versions, performance does increase. Achieved performance is still far from the maximum peak performance, but as stated in the roofline caption the actual roof may be smaller due to the instruction mix. The program remains compute bound.

Advanced Vector eXtensions (AVX2). As the program is still compute bound, vectorization will enable to process more data using fewer operations. Using AVX2, independent operations are now performed simultaneously on eight floats. The main issue with vectorization is to find the correct data alignment, loading and computation pattern to exploit as much as possible the CPU ports.

To avoid invalid data accesses, computations are performed on the inner part of the image of size $H - 2 * (R + f)$. This prevents from having aligned data accesses, as values of R and f vary across the different filters. Moreover, border patches do not have a size multiple of eight in all cases, therefore more computations than necessary are performed. This leads to question the actual choice of the memory layout. Interesting factors are the number of shuffles across the 128-bit lane as well as the access time for a value in memory. For each memory layout, a specific code with adjusted ILP using the autotuning script has been implemented. More details on memory layout can be found in the following buffer memory layout section. Further, as the algorithm relies on several exponentations which do not belong to standard Intel intrinsics, a slightly modified version of an existing implementation [15] is used.

The roofline model on figure 3 hints for some issues that arise with vectorization. Even if performance does increase by a non negligible factor, it is very far from the expected x8 factor. Indeed, by vectorizing the code, it is now limited by the vectorized roofline. And therefore, an attentive reader will remark that the code is now memory bound which limits the possible performance gain. Measures from Intel VTune confirm the memory boundness, with more than 50% of instructions ending up memory bound and only 25% of instructions ending up being retired.

Blocking. In order to decrease last level cache misses and relieve pressure on the memory channel, blocking is applied on the vectorized version as a next step. Operating on smaller blocks would greatly increase temporal locality since data is traversed multiple times. However blocking introduces a new challenge as neighboring blocks share multiple expensive computations.

Therefore two strategies have been implemented: First, blocking by using a global array to store intermediate computations such that they can reused across blocks. Unfortunately

this solution increases the memory footprint. The second strategy performs naive blocking by doing overhead computations.

A profiling analysis confirmed that both blocking attempts reduce the memory boundness and make better use of the available memory bandwidth. In terms of runtime, doing overhead computations is superior, but still slower than the default vectorized approach.

Buffer Memory Layout. In the basic version, buffers were stored on a $(3, W, H)$ layout, with arrays of size H linked by pointers to form 3D arrays. As contiguous arrays are easier to manage and faster to access than pointers, we replaced pointers by a single contiguous memory block for each image. As explained in the section about vectorization, data alignment is not crucial here as the algorithm itself prevents data from being loaded in an aligned fashion. In addition, one needs to choose how data is arranged in this array. Some of the innermost loops of our algorithm are necessarily over the channel counter as they aggregate the values over all channels in a single new channel. Therefore, to provide better spatial locality in those cases, we replaced the memory layout $(3, W, H)$ with $(W, H, 3)$ where values of different channels but same width and height are contiguous.

4. EXPERIMENTAL RESULTS

This section presents the different experimental results we have obtained during the optimization process. It first presents the experimental setup, then highlights the actual results.

Experimental setup. In order to provide reliable, comparable and reproducible results, experiments have been performed on a single machine. Namely on a Intel(R) Core(TM) i9-8950HK CPU @ 2.90GHz with cache sizes of: 32KB L1, 256KB L2 and 12MB L3 and a maximum memory bandwidth of 41.8 GB/s. Apple clang compiler (version 11.0.0) on MacOS Catalina (10.15.4) system.

On this platform a testing infrastructure has been implemented. It is divided into two components: (1) a validation infrastructure and (2) a benchmarking infrastructure. The validation infrastructure ensures all code versions produced during development are coherent and produce the same results. To this end, two measures are used to compare different code versions: (1) the Root Mean Square Error (RMSE) [16] which provides some insight about the algorithm precision and (2) the maximal absolute error difference. As the algorithm is deterministic, results should be exactly the same. This is only true up to a certain floating point precision as floating point arithmetic is different from real arithmetic. On the other hand, the benchmarking infrastructure measures the number of cycles elapsed between two points in the program using the Time Stamp Counters (TSC). Measure is performed only on the relevant piece of code, with a warm cache as several executions for testing validity are performed before benchmarking.

Using a 3D renderer, several testsets have been created to evaluate the performance of the algorithm. Images are of squared size with sizes of 64, 128, 256, 512, 768, 1024, 1536 and 2048, sampled using 100 samples per pixel.

Results. As stated in the previous section, the first main optimization aims at reducing the number of operations. As it makes no sense to compare performance of algorithms with different number of operations, the relevant indicator is the runtime. Runtime of all algorithms are shown on figure 4. On this graph, speedup introduced by restructure is quite clear, with roughly a x13 factor (see also figure S4). Based on this restructuring, another version has been created to include some precomputations. Once again, the number of operations changes, therefore the indicator is runtime, depicted on figure 4. It further improves runtime with a speedup of x16 compared to the basic implementation.

Those two implementations, with and without precomputations, serve as reference for further optimizations. As stated in the previous section, at this point code is mostly compute bound. Therefore the next experiment aims at verifying this compute boundness, as well as testing if improving ILP leads to better performance. The answer lies in the performance plots 6 and 7 which hold performance improvements for the version without precomputations and with precomputations respectively. In both cases, using SSA, unrolling and accumulators give an increase in performance of 1.5x. This leads to a scalar code reaching 40% of the machine's peak performance for the code without precomputation and 25% for the code with precomputations. Note that the actual peak performance for this specific algorithm is probably lower due to the instruction mix. Runtime wise, the amelioration of performance translates directly as a speedup factor, illustrated in figure 4: the version with precomputations is 45x faster than the basic implementation (see figure S4). While the version without precomputations has better performance, it has more operations, which result in worse runtime than the version with precomputations. The former performs unnecessary computations, but utilizes better the CPU resources, as the ratio of expensive operations is lower.

The versions presented in the last paragraph are the best scalar implementations achieved. This paragraph presents the vectorized versions. In particular, results depicted on figures 6 and 7 show that performance is not multiplied by 8 as expected, which proves the memory boundness of the

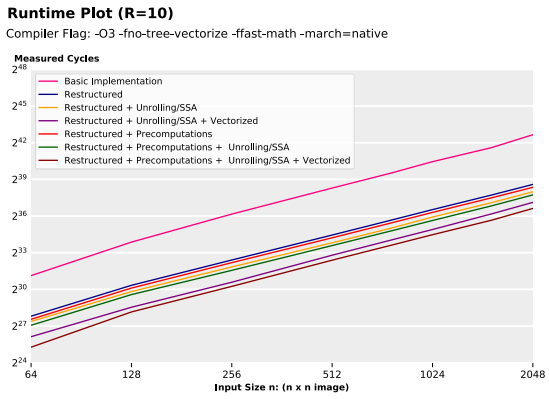

Fig. 4. Runtime Plot in a logarithmic scale - Includes all major implementations

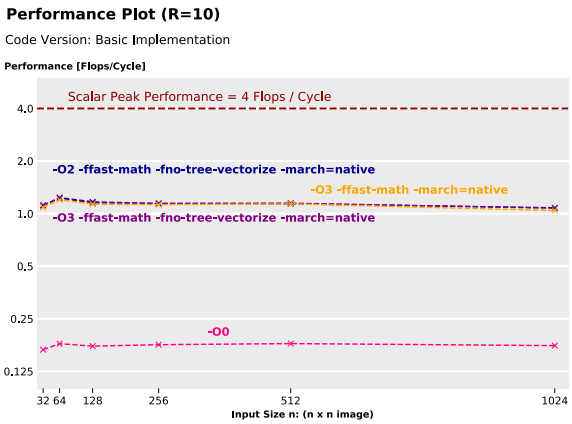

Fig. 5. Performance Plot - Basic Implementation using different compile flags. Flags `-O2` and `-O3` achieved very similar performances.

vectorized code. It does also show the claimed speedup in the abstract, as shown on figures 4 and S4, with the best vectorized version being 62.5 times faster than the basic implementation. For the non-improved runtime results of the blocked versions, we refer to the supplementary section 8.6.

5. CONCLUSIONS

This paper presents an optimized implementation of a denoising algorithm for Monte Carlo renderings. We first restructure the algorithm itself, taking advantage of kernel separability to reduce the asymptotic complexity. Then we take further the operation count reduction by exploiting possible precomputations of weights and per-pixel distances. Those optimizations incur a speedup of x15 compared with the original implementation. The next step was to break linear dependencies using loop unrolling and accumulators. We created an autotuning script to determine automatically the best parameters and provide the best unrolled program performance-wise. After those optimizations, runtime has improved by a factor 27.5 compared to the original implementation, reaching 40% of peak scalar performance. Finally the code offered by this paper has been vectorized using AVX2. Even though AVX-2 allows for simultaneously operating on eight floats, performance has not improved by a factor eight since the program becomes memory-bound. To tackle this issue, blocking has been applied to improve temporal locality, bringing additional challenges. In the end, the vectorized version achieves a speedup of x62.5 as compared to the original implementation, but only achieving 9% of peak vectorized performance due to memory boundness. Further work could improve the blocking strategy to further improve performance, for instance by keeping overhead computations in smaller arrays fitting in cache across iteration of multiple blocks.

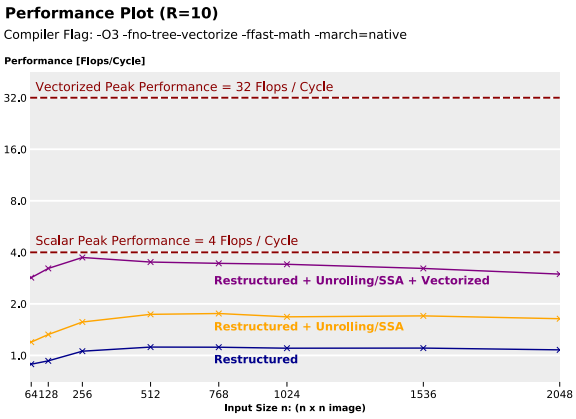

Fig. 6. Performance Plot - Restructured Implementation without Precomputations

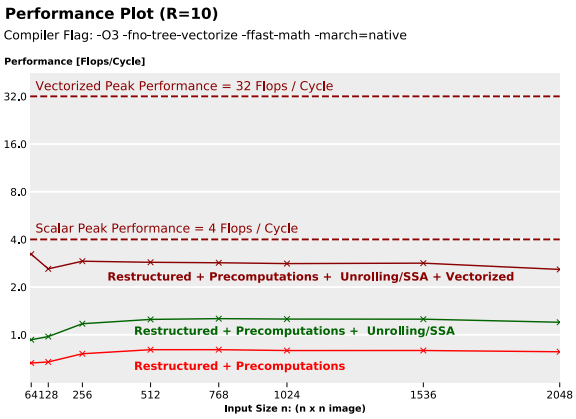

Fig. 7. Performance Plot - Restructured Implementation with Precomputations

6. CONTRIBUTIONS OF TEAM MEMBERS

Alexandre P.. Basic implementation: weights computations + convolution. Worked with Felicite on reducing the opcount with precomputations (not kernel separability / restructure) and with math simplifications. ILP/SSA: worked with Nino on factorizing code to reduce number of div/exp, adding many precomputations. Analysed bottlenecks to find loops where ILP was to be increased, then broke some long dependencies with accumulators. Implemented the autotune script, adapted all the code to use the autotune script and searched for good parameters using it. Vectorization: worked on finding good ILP (manually as well as with my autotune script), adapted unrolling to ensure correctness even for sizes not multiples of 8. Mainly focused here on the vectorized version with precomputations. Adapted precomputations to keep them in cache whenever possible. Blocking: implemented the blocked versions of the vectorized algorithm with Nino. Also created the roofline plot and a good amount of the analysis coming from it. Including cache analysis with valgrind. Also took part in some experiments that did not work out as expected (such as changing layout, increasing ILP further by making dependency graphs, ...).

Alexandre B.. First teamed up with Nino to work on the restructure of the algorithm. Worked on an early implementation of blocking. Worked with Felicite on vectorizing the code, especially for avoiding the load of non-allocated memory location. In charge of profiling with Vtune to give hints about the performance bottleneck.

Félicité. First focused on precomputations without restructuring to reduce number of operations with Alexandre P. Experimented with different memory layout and loop orderings to improve locality. In particular, implemented the single contiguous array and the $(W, H, 3)$ layout. Worked with Alexandre B. on vectorizing the code. In the vectorized code, worked on alignment of memory addresses and added FMAs where possible.

Nino. Basic Implementation: Implementation of the overall algorithm || Restructured Implementation: Came up with the idea to integrate box-filtering (using kernel separability) to reduce the computational complexity. Implemented it and eliminated expensive operations on the go by math simplification or computation rearrangement || Precomputations: Detected shared intermediate computations and implemented it || ILP/SSA: Worked together with Alex P. on eliminating long latency operations and breaking long dependencies. Analyzed the computation flow to find the sweet spots to increase ILP - applied unrolling and integration of accumulators || Vectorization: Improved the order of computation and integrated further FMA's || Blocking: Peformed Cache Analysis by using Valgrind to analyse last level cache misses and memory bandwidth utilization. Teamed up again with Alex P. to implement the two mentioned blocking versions to overcome memory boundness || Memory Layout: Worked on multiple buffer memory layouts to further increase locality including an overall data layout for controlling cache misses - due to the extensiveness of the algorithm it was unfortunately not possible to finish this in the limited time || Analysis: Continous Profiling/Roofline Analysis to guide next steps of optimization. Created automated bash scripts to evaluate runtime and performance including the semi-automated creation of the corresponding plots (Jupyter Notebook).

7. REFERENCES

[1] James T Kajiya, "The rendering equation," in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986, pp. 143–150.

[2] Robert L. Cook, Thomas Porter, and Loren Carpenter, "Distributed ray tracing," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1984, SIGGRAPH '84, p. 137–145, Association for Computing Machinery.

[3] Nima Khademi Kalantari and Pradeep Sen, "Removing the noise in monte carlo rendering with general image denoising algorithms," in *Computer Graphics Forum*. Wiley Online Library, 2013, vol. 32, pp. 93–102.

[4] Michael D McCool, "Anisotropic diffusion for monte carlo noise reduction," *ACM Transactions on Graphics (TOG)*, vol. 18, no. 2, pp. 171–194, 1999.

[5] Ruifeng Xu and Sumanta N Pattanaik, "A novel monte carlo noise reduction operator," *IEEE computer graphics and applications*, vol. 25, no. 2, pp. 31–35, 2005.

[6] Ryan S Overbeck, Craig Donner, and Ravi Ramamoorthi, "Adaptive wavelet rendering.," *ACM Trans. Graph.*, vol. 28, no. 5, pp. 140, 2009.

[7] Fabrice Rousselle, Marco Manzi, and Matthias Zwicker, "Robust denoising using feature and color information," in *Computer Graphics Forum*. Wiley Online Library, 2013, vol. 32, pp. 121–130.

[8] Matthias Zwicker, Wojciech Jarosz, Jaakko Lehtinen, Bochang Moon, Ravi Ramamoorthi, Fabrice Rousselle, Pradeep Sen, Cyril Soler, and S-E Yoon, "Recent advances in adaptive sampling and reconstruction for monte carlo rendering," in *Computer graphics forum*. Wiley Online Library, 2015, vol. 34, pp. 667–681.

[9] Benedikt Bitterli, Fabrice Rousselle, Bochang Moon, José A Iglesias-Guitián, David Adler, Kenny Mitchell, Wojciech Jarosz, and Jan Novák, "Nonlinearly weighted first-order regression for denoising monte carlo renderings," in *Computer Graphics Forum*. Wiley Online Library, 2016, vol. 35, pp. 107–117.

[10] Thijs Vogels, Fabrice Rousselle, Brian McWilliams, Gerhard Röthlin, Alex Harvill, David Adler, Mark Meyer, and Jan Novák, "Denoising with kernel prediction and asymmetric loss functions," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 1–15, 2018.

[11] Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony Derose, and Fabrice Rousselle, "Kernel-predicting convolutional networks for denoising monte carlo renderings.," *ACM Trans. Graph.*, vol. 36, no. 4, pp. 97–1, 2017.

[12] Camille Sutour, Charles-Alban Deledalle, and Jean-François Aujol, "Adaptive regularization of the nl-means: Application to image and video denoising," *IEEE Transactions on image processing*, vol. 23, no. 8, pp. 3506–3521, 2014.

[13] Agner Fog, "Instruction tables - lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd, and via cpus," 2020.

[14] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G Spampinato, and Markus Püschel, "Applying the roofline model," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 76–85.

[15] JishinMaster (Github), "A header only library implementing common mathematical functions using SIMD intrinsics," https://github.com/JishinMaster/simd_utils, 2020, Online; last accessed 12 June 2020.

[16] Alexei Botchkarev, "Performance metrics (error measures) in machine learning regression, forecasting and prognostics: Properties and typology," *arXiv preprint arXiv:1809.03006*, 2018.

8. SUPPLEMENTARY MATERIAL

8.1. Exact Operation Counts of a Straightforward Implementation

Operation	Exact Number of Executions
ADD	$129600(n-16)^2 + 864(n-4)^2 + 576R^2(n-2(R+1))^2 + 5184R^2(n-2(R+3))^2 + 48n^2$
MUL	$64800(n-16)^2 + 432(n-4)^2 + 288R^2(n-2(R+1))^2 + 2592R^2(n-2(R+3))^2 + 45n^2$
DIV	$21600(n-16)^2 + 144(n-4)^2 + 96R^2(n-2(R+1))^2 + 864R^2(n-2(R+3))^2 + 9n^2$
EXP	$21600(n-16)^2 + 12(n-4)^2 + 96R^2(n-2(R+1))^2 + 864R^2(n-2(R+3))^2$
MIN/MAX	$21600(n-16)^2 + 144(n-4)^2 + 96R^2(n-2(R+1))^2 + 864R^2(n-2(R+3))^2$

Table S1. Exact operation counts of the straightforward implementation for images of squared size with image width/height n and window radius R. The filtering parameters are chosen as in Algorithm 1.

8.2. Distribution of Operations for Different Implementations

Operation	Percentage of Total Operations		
	Basic	Restructure w/o. Precomp.	Restructure w. Precomp.
ADD	49.3	50.3	53.1
MUL	25.4	20.5	20.0
DIV	8.3	7.1	7.1
EXP	7.9	1.6	2.4
MIN/MAX	9.1	20.5	17.4

Table S2. Distribution of operations for a fixed image size of 1024×1024 and $R = 10$

8.3. Illustration of Computational Complexity of Major Code Versions and its Repective Speedup

Figure S1 illustrates the reduction of floating point operations for major code improvements.

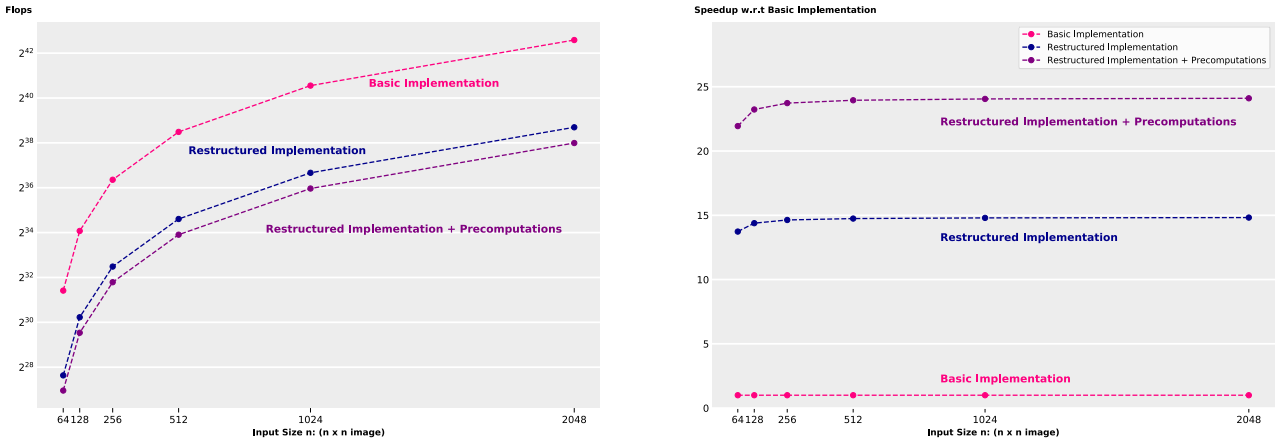

Fig. S1. Illustration of the reduction of operations for fixed $R = 10$ between major code versions (left) and its respective speedup with respect to the number of operations of the basic implementation

8.4. Benchmarking for varying parameter R

The neighborhood parameter R defines the precision of the denoising algorithm. The greater R is, the greater the precision of the filter will be. However, it also increases the number of operations as the complexity of the algorithm is proportional to R^2.

Tests performed for correctness show that the denoising algorithm achieves good precision for $R \geq 10$ (the precision being measured with the RMSE factor [16]). Moreover, figure S2 seems to show that R does not impact performance significantly. However it does impact runtime as stated above. Therefore we have chosen to fix $R = 10$ as input value for most parts of the paper.

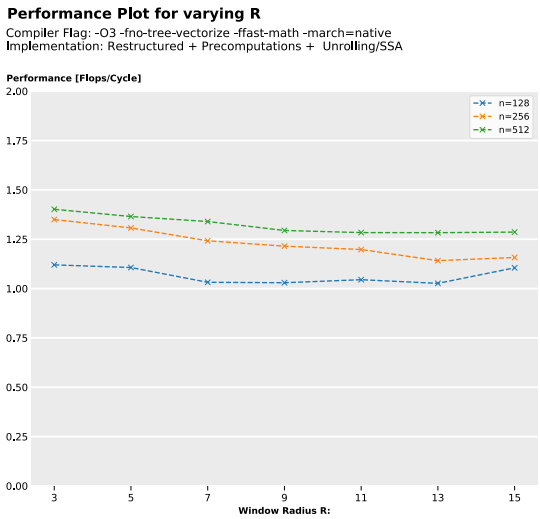

Fig. S2. Illustration of the impact of different values for the neighborhood parameter R

8.5. Runtime Speedup Plot for major implementations

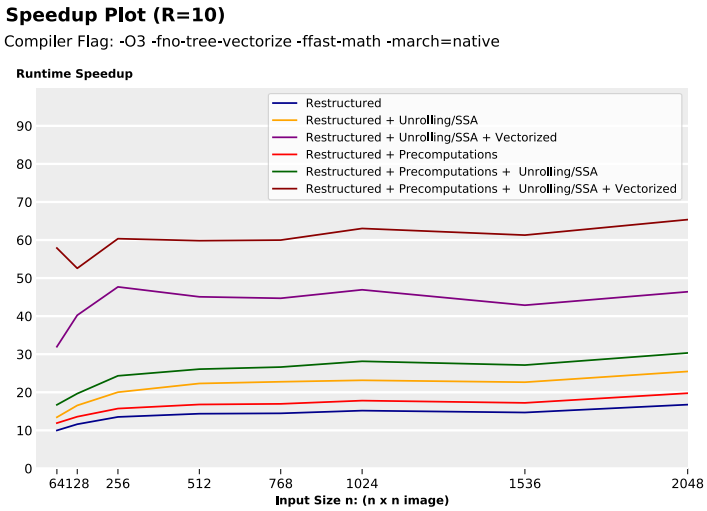

Fig. S3. Illustration of runtime speedups for major implementations

8.6. Runtime Evaluation of Blocking Implementations

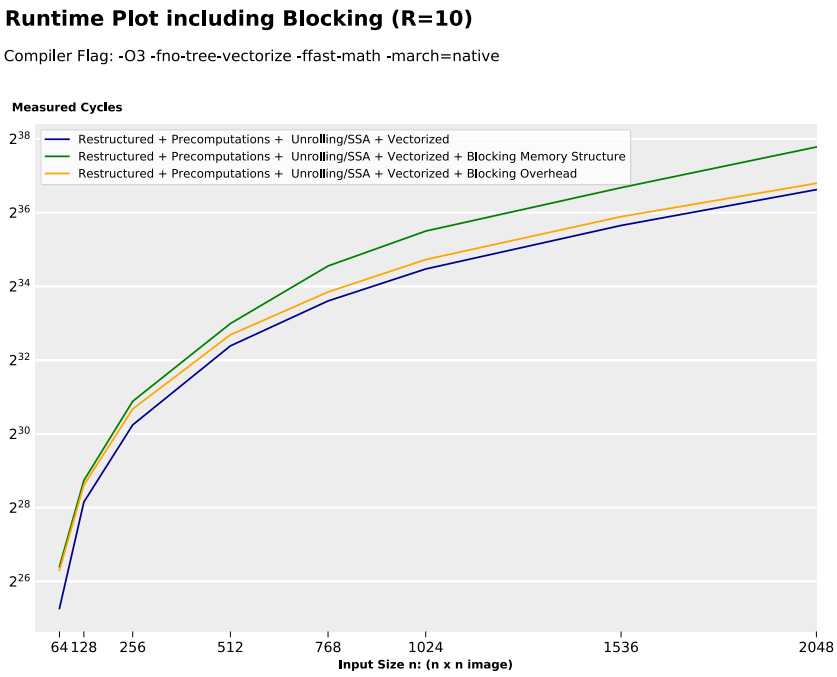

Fig. S4. Illustration of runtime for different blocking implementations in comparison to the default vectorized version