CONTENT-AWARE IMAGE RESIZING WITH OPTIMIZED SEAM CARVING

Aydin Faraji, Dora Neubrandt, Ioana Stefan, Tijana Klimovic

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

The project optimizes the seam carving algorithm with the help of hardware-specific features. The seam carving algorithm resizes images while keeping the most important information. It is mostly used for resizing images for different screen sizes and so the optimization of the algorithm is important for a reduced runtime, especially in the context of large images. For this project, only the size reduction of images was implemented and optimized. The content-aware reduction is achieved by computing (and removing) an 8-connected path, containing pixels of minimum importance, named seam. Multiple seams are removed for an image in an optimal order.

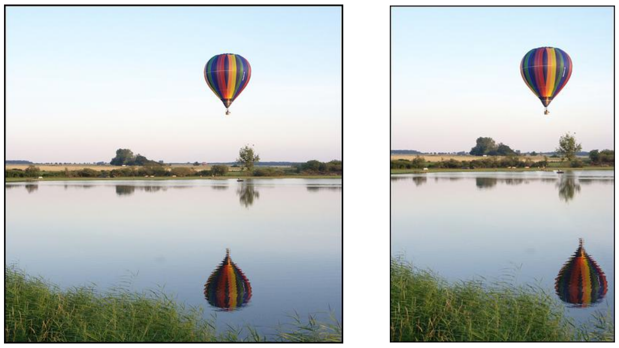
1. INTRODUCTION

This section introduces the importance of the Seam Carving algorithm and related work in this field. An overall contribution of this project is also presented in this section.

Motivation. The purpose of the Seam Carving algorithm is to realize a content-aware resizing of images. HTML, as well as other standards, can support dynamic changes of page layout and text. Images, however, retain their original size no matter the screen size of the client device. Therefore, dynamic resizing of images while keeping the most important information is desired to make the size changes automatic in all elements of a page. Moreover, automatic change in the size/aspect ratio of an image may be desirable when needing the same image to be printed on different paper sizes or resolutions. Due to this real-time demand for image resizing, there is a need for high-performance implementations.

Achieving this image resizing through image cropping or standard image scaling is not sufficient since these methods are oblivious to the image content, which can result in insignificant parts of images being preserved and important ones being removed.

Seam carving is trying to take into account the downside of these approaches. It uses an energy function defining the importance of each image pixel. A seam is an 8-connected path of low energy pixels crossing the image from top to bottom, or from left to right. By successively removing or inserting seams we can reduce, as well as enlarge, the size of an image. An example of such an image transformation is given in Fig. 1.



(a) Original image (b) Resized image

Fig. 1: Size reduction for an example image, where the main reduction is focused on the width.

It is hard to achieve fast implementations of the Seam Carving algorithm because, in order to calculate the importance of a pixel, the energy function needs to account for dependencies on the neighboring pixels. In addition, computing the seam of minimum importance requires accounting for the entire image, which results in sequential dependencies.

Contribution. An optimized implementation of the Seam Carving algorithm is presented in this paper. This implementation is faster than other straight-forward or available implementations by using hardware-specific features. This implementation has a dedicated hardware and it exploits different characteristics of the hardware, such as execution units, caches and SIMD.

Related work. The algorithm used in this project is the one introduced by Avidan et al. [1]. This is one of the first image resizing approaches that took into consideration the entire content. It was designed for both reduction and expansion of images, making object removal also possible. The aspect ratio of an image is changed by repeated removals or insertions of seams.

Other works made different optimizations based on the

Seam Carving algorithm, such as the work of Dong et al. [2]. This paper focuses on an optimized content-aware image resizing, which combines the Seam Carving algorithm along with image scaling.

Different variations of the Seam Carving algorithm were also presented in previous papers. One interesting variation in the Perceptual Seam Carving algorithm, introduced by Hwang et al. [3]. This work uses the human attention model to compute the importance of pixels. In addition, it tries to avoid the excessive distorting of images that is sometimes encountered with the original Seam Carving algorithm by combining the original algorithm with resampling.

Similarly to the image algorithm considered here, content-aware resizing was also desired for videos. Multiple works want to achieve this and one of the most known ones is the work of Rubinstein et al. [4]. A slightly modified Seam Carving algorithm is introduced here, the changes to the algorithm being caused by the increased size. The removal of 1-dimensional seams from images is replaced by the removal of 2-dimensional seams from videos, which are considered to be 3-dimensional volumes.

As a comparison with the presented works, the current paper optimizes the original Seam Carving algorithm for images not by changing the algorithm or combining it with other approaches, but by mapping the original algorithm to a specific hardware, in accordance with its structure and options.

2. ALGORITHM BACKGROUND

This section briefly describes the Seam Carving algorithm and additional information related to each of the steps involved in the algorithm. The cost analysis and the asymptotic complexity of the algorithm are also mentioned later in this section.

Algorithm. A restricted version of the Seam Carving algorithm is presented in this paper: only size reductions are considered. Both vertical and horizontal reductions are possible. The Seam Carving algorithm can be structured into three separate steps or stages.

Step 1. The importance of each pixel in the image is computed with the help of an energy function. This function is applied to each pixel, over all channels. The results of one pixel from all channels are summed in order to obtain the overall energy value of a pixel, which is further considered as the importance of that pixel. The energy function mainly presented in the original paper that introduces Seam Carving [1] is also used in this project:

$$e(I) = |\frac{\partial}{\partial x}I| + |\frac{\partial}{\partial y}I| \quad (1)$$

Step 2. 8-way connected paths of pixels are constructed vertically or horizontally. The computation of each of these seams is done based on the formulas used by Avidan et al. [1]. For an image of size $n \times m$ A vertical seam is defined as:

$$s^x = \{s_i^x\}_{i=1}^n = \{(x(i), i)\}_{i=1}^n \quad (2)$$

and a horizontal seam is defined as:

$$s^y = \{s_j^y\}_{j=1}^m = \{(j, y(j))\}_{j=1}^m \quad (3)$$

As a result, the pixels of a path of a vertical seam s will be:

$$I_s = \{I(s_i)\}_{i=1}^n = \{I(x(i), i)\}_{i=1}^n \quad (4)$$

and the pixels of a path of a horizontal seam s will be:

$$I_s = \{I(s_j)\}_{j=1}^m = \{I(j, y(j))\}_{j=1}^m \quad (5)$$

Only one seam is selected for a reduction at a time and that seam is the one with the minimum cost, meaning with the minimum sum over the energy values of its pixels. The formal definition of such a minimal vertical seam is given in the following formula:

$$s^* = min_s E(s) = min_s \sum_{i=1}^{n} e(I(s_i)) \quad (6)$$

The optimal seam is identified with the help of dynamic programming (DP): the entire image is traversed and intermediate cost sums are computed for each possible seam; the minimum is stored in the DP matrix. The formal definition of the DP used in this step, for a vertical seam, can be found below:

$$\begin{aligned} M(i,j) = e(i,j) + min(&M(i-1,j-1), \\ &M(i-1,j), M(i-1,j+1)) \end{aligned} \quad (7)$$

Step 3. Vertical and horizontal seams are removed in an optimal order until the desired image size is reached. The objective function used by Avidan et al. [1] to search for the optimal order to remove seams is defined below:

$$min_{s^x, s^y, \alpha} \sum_{i=1}^{k} E(\alpha_i s_i^x + (1-\alpha_i) s_i^y) \quad (8)$$

Similar to the previous step, dynamic programming is also used when computing the optimal order. The formal definition of the recursive step of this DP is presented in the following equation:

$$\begin{aligned} T(r,c) = min(&T(r-1,c) + E(s^x(I_{n-r-1 \times m-c})), \\ &T(r,c-1) + E(s^y(I_{n-r \times m-c-1}))) \end{aligned} \quad (9)$$

Cost Analysis. Since the algorithm handles only pixel values and different mathematical computations with these values, integers were used in the value representation of this project. The following cost measure refers to integer operations and pointer computations:

$$C(n) = (adds(n), mults(n), shifts(n)) \tag{10}$$

The `shifts(n)` measure is used later in the solution provided by this paper, by the optimizations. While the exact cost calculation was done by instrumenting the implementation, an approximated result for the base implementation is given below:

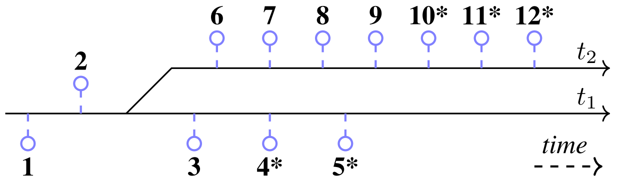
$$C(n) = ((1412nd * md - 419)m * n + 118nd^2md^2 \\ -1137nd^2md^2m - 1095nd^2md^2n, (490nd*md+287)m*n \\ + 48nd^2md^2 - 476nd^2md^2m - 477nd^2md^2n, 0) \tag{11}$$

where `n` and `m` are the height and the width of the original image and `nd` and `md` are the height and the width difference in pixels, more precisely the number of horizontal and vertical seams to be removed from the original image.

The asymptotic complexity of the base Seam Carving algorithm is $O(n \times m \times nd \times md)$.

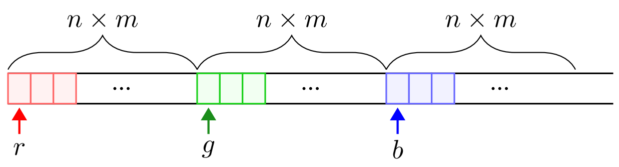
3. OUR PROPOSED METHOD

This section covers the baseline code as well as the step wise analysis and optimizations performed on top of this base that lead to the final high performance implementation. The organization of this section follows the organization of our approach to the high performance result and is illustrated below in a timeline.



As can be seen from the timeline the project branches out into two separate tracks (t_1 and t_2) after step 2. Track t_1 has the data representation presented in the Baseline subsection whilst track t_2 has the data representation described in the RGB sub-section. The optimizations described here are mostly applied to `convolution.c` and `min_seam.c` files, since the profilers VTune and perf identified these two to be the bottlenecks w.r.t. both access memory and port utilization. We distinguish between two optimizations in our timeline. The optimizations annotated with a * give a performance boost only when compiled with `-O`. (no boost with `-Ofast`). The optimizations without * give an improvement in either case. Apart from these improvements, in the final sub-section we list and briefly explain other less successful attempts.

1. Baseline. The base code was implemented from scratch in C, taking inspiration from the python implementation of Karthik Karanth[5]. The images are represented as a 3D integer array of size $3 \times n \times m$, where n is the height and m is the width of the image, with the following layout in memory:



where the pointers r, g and b point to memory locations containing the red, green and blue value of the first pixel in the first row and first column of the image respectively. The red, green and blue $n \times m$ 2D arrays save their color value of each pixel in row-major order.

The code consists of three main `.c` files: `convolution.c`, `min_seam.c`, `optimal_image.c`, which implement step 1, step 2 and step 3 of the algorithm respectively, described in section 2.

The energy function in equation 1 uses Sobel filters to compute the partial gradients of an image. These filters are convolutional kernels that are run independently over each pixel of the image on every channel. To ensure convolution is performed on the border pixels too, we pad each channel with 0s. This gives rise to two 3D arrays each corresponding to the result of convolution with one of the 2 filters. We then take the element-wise absolute value of these arrays and sum them into a 3D array `result`. Finally, we sum the 3 channels of `result` into a $n \times m$ 2D array which is the energy function.

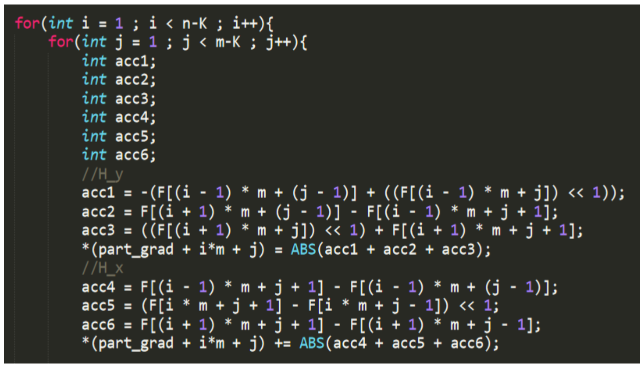
In order to find the seam with the overall minimum energy, we traverse the energy map as computed by the energy function. Following equation 7 for a vertical seam, we start from the second row and for each pixel, find the minimum between the 3 upper pixels and add it to the current pixel energy (we also keep track of which pixel was the minimum, so that we can backtrack later to get the seam path). This way we calculate the cumulative energy in the seam up until the current pixel. After computing the last row, we simply find the minimum in the last row and follow the backtrack to get the seam. For horizontal seams, we follow a similar algorithm in which the image is traversed from left to right.

The computation of the optimal reduced image is implemented using dynamic programming, based on the model presented in equation 9. The reduced image on position $nd \times md$ of the DP is the wanted result, where `nd` refers to the number of horizontal seams to be removed and `md` refers to the number of vertical seams to be removed. Furthermore, an optimization is added to the previously presented algorithm: only the last two rows used are stored for the `T` matrix, in order to reduce memory consumption.

2. Explicit Sobel filter and ILP in convolution. The Sobel filters in two different directions of the image are:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

When computing the convolution of one channel of a pixel, instead of multiplying all 8 surrounding pixels' values by the corresponding values in the Sobel filters and summing them, we can hard code the calculation. Namely, instead of multiplying by 0 we simply don't include it in our computation since it is irrelevant. Multiplying by 1 is also unnecessary and hence is reduced to just adding the neighbour value. Multiplying by 2 can be done as a shift left by 1, which is faster. Instead of multiplying the neighbour values by -1 and -2 and only then adding them, we take -1 as the common factor and first add the neighbour values (with one of them being shifted to the left by 1) and then multiply with the common factor. Moreover, we use accumulators and rearrange the calculations to provide better ILP. The code snippet in Fig. 2 summarizes this optimization:

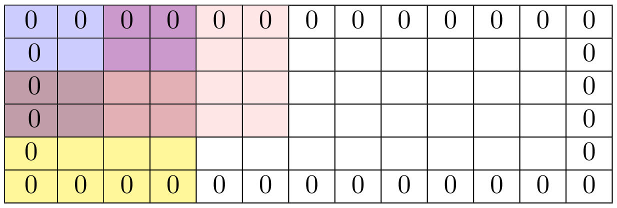


```
for(int i = 1 ; i < n-K ; i++){
    for(int j = 1 ; j < m-K ; j++){
        int acc1;
        int acc2;
        int acc3;
        int acc4;
        int acc5;
        int acc6;
        //H_y
        acc1 = -(F[(i - 1) * m + (j - 1)] + ((F[(i - 1) * m + j]) << 1));
        acc2 = F[(i + 1) * m + (j - 1)] - F[(i - 1) * m + j + 1];
        acc3 = ((F[(i + 1) * m + j]) << 1) + F[(i + 1) * m + j + 1];
        *(part_grad + i*m + j) = ABS(acc1 + acc2 + acc3);
        //H_x
        acc4 = F[(i - 1) * m + j + 1] - F[(i - 1) * m + (j - 1)];
        acc5 = (F[i * m + j + 1] - F[i * m + j - 1]) << 1;
        acc6 = F[(i + 1) * m + j + 1] - F[(i + 1) * m + j - 1];
        *(part_grad + i*m + j) += ABS(acc4 + acc5 + acc6);
```

Fig. 2: ILP and explicit Sobel filter

3. Loop unrolling and scalar replacement for convolution. Basic optimizations like loop unrolling and scalar replacement were applied to track t_2 on top of the improvements presented in the previous paragraph. While loop unrolling could have had good potential, it resulted that only a small unrolling, of a factor of 2, would bring the maximum improvement. The reason for this was that the potential loops to be unrolled already contained a significant number of operations. Regarding scalar replacement, a good improvement was brought when the code is compiled without optimization flags. However, compiling with the `-Ofast` flag significantly reduces the gain since most of the optimizations were already achieved by the compiler.

4*. Blocking for L1 and registers.
Due to the dependency between the neighbouring pixels, simply doing convolution over smaller same size partitions of the image and combining these into the convolution result of the original is incorrect. The dependencies are presented below for a single channel(different blocks are colored differently to distinguish easier):



The diagram above shows the reuse of the purple block w.r.t. the pink horizontal neighbour block and the yellow vertical neighbour block. This is because the last row and last column of the smaller blocks serve as padding in their convolution calculation, and hence if the block is of size $M \times N$ we would be computing the convolution over $M - 2 \times N - 2$ pixels. Therefore, in order to calculate the convolution of the 'padding' pixels in the purple block, we have the overlaps as above.

Following the block image subdivision, we consider the working set analysis, in order to determine N and M. We move horizontally within a block to compute the convolution as it is more cache friendly(layout is row major order). When computing the convolution of one row of the block, we use a total of 3 rows, i.e the row we are computing, the row directly above and directly below it. Once we finish the computation of this row, we move onto the row that is directly below it. Now this second row's convolution involves 3 rows again, two of which were used by the first. Similarly the third row has 1 row reused from the first row's convolution. After this point the first row's convolution accesses are never reused. Due to this we would like to preserve at least 3 rows of our block in cache. However, due to non clean replacement, we end up needing 4 rows of size M preserved in our cache. Apart from the 4 rows, we need to make sure we have at least one integer space in our cache for our partial gradient entry to which we write the convolution result of our currently considered pixel. But again due to the access pattern, we need to preserve one row of the partial gradient too. Hence we end up with the following formula for our working set:

$$4 * M + M - 2 = \frac{C}{4} \tag{12}$$

where C is the capacity of the L1 cache, and the 4 is due to the type being integer. In the case of Skylake and Haswell $C = 32KB$ and hence we have $M = 1560$. With this analysis, we only have a constraint on the width of the block, and hence the height N is equal to the height of the padded image.

Regarding blocking for registers we have that the working set is:

$$3 * M + M - 2 = 16 \tag{13}$$

which gives us $M = 4$. However, even though we don't have a restraint on the height, we must take into account what is present in the L1 cache. If we take the height of the block to be the size of the image, once we finish computing the first row of the register block, we would continue onto the second, third and fourth, after which point we would not have the values we need in L1 anymore. To avoid this scenario, we restraint the height to be $N = 3$, ensuring that all values the register may need, are in L1. Therefore we have the register blocks being 4×3 in size. We, of course, traverse between the blocks in a horizontal manner, again ensuring all the values are in L1.

5*. Computation reuse. Consider we have the following pixels (only 1 channel is depicted for simplicity):

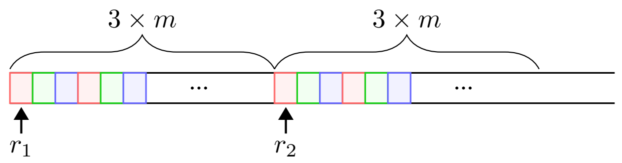
p_0	p_1	p_2	p_3	p_4
p_5	p_6	p_7	p_8	p_9
p_{10}	p_{11}	p_{12}	p_{13}	p_{14}

When we go over the image horizontally, based on the Sobel horizontal kernel, we have:

$$p_6 : (p_0 + 2p_5 + p_{10}) - (p_2 + 2p_7 + p_{12}) \tag{14}$$
$$p_8 : (p_2 + 2p_7 + p_{12}) - (p_4 + 2p_9 + p_{14}) \tag{15}$$

And we can see that the $(p_2+2p_7+p_{12})$ expression is reused and this reuse is repeated for all the pixels (with a stride of 2); As a result, we can save this expression result and reuse it. Moreover, since it has a stride 2 and we do not want to lose locality, we unroll the loop.

6. Spatial locality of RGB value per pixel. This optimization was merely a change in the memory layout of our data that would facilitate better spatial locality and easier parallelism of the convolution computation. Instead of the image being saved as described in the Baseline, we have the following memory layout:



where all three RGB values corresponding to the same pixel are next to each other in memory. The pointer r_1 points to the location of the red value of the pixel in the first row and column and r_2 points to the red value of the pixel in the second row and first column.

7. Energy map rotation. At minimum seam calculation when we want to find a horizontal seam we need to traverse the energy map in column-major order. This is not cache friendly as we loose spatial locality. Instead, we rotate the energy map at the beginning and parse that in a row-major order then convert back the indexes of the results in the required format. Even though we gain a small overhead by rotating the matrix, which can only be seen at very small images, we gain spatial locality which results in significant performance improvements.

8. SIMD for convolution. We leverage the fact that the calculations for convolution of different pixels are independent of each other. For the convolution of each pixel, we need the pixel values at the 8 surrounding pixels. We want to avoid doing different calculations for the subset of values in each register because that would require us extracting those data from the register, performing the calculations and then putting them back which is costly. Thus, for each register, we load the data that needs the same computation. Here's a figure for the pixels: (we are calculating the convolution for the pixel at *)

$$\begin{matrix} NW & N & NE \\ W & * & E \\ SW & S & SE \end{matrix}$$

If we are calculating the convolution for 2 pixels A and B, the best candidates who need the same computations are the cells that are located the same relative to the pixels A and B; i.e. one register gets loaded with NW pixels of A and B, another register gets loaded with N pixels of A and B, and so on. Since each register has a capacity of 16 shorts, and we're doing all the channels together (so NW actually is 3 shorts, one for each channel), we can fit at most 5 pixel RGB values. As a result, with 8 registers, we can do the convolution computation for 5 pixels together as follows:

reg_0	RGB values of NW cells for the 5 pixels
reg_1	RGB values of N cells for the 5 pixels
$\vdots$	
reg_7	RGB values of SE cells for the 5 pixels

Now, if we want to calculate e.g. the Sobel vertical kernel multiplication, $NE - NW + 2 \times (E - W) + SE - SW$ gets translated to $reg_2 - reg_0 + 2 \times (reg_4 - reg_3) + reg_7 - reg_5$ and 5 pixel convolutions are computed at the same time.

In order to extend this method to be able to use all the 16 registers, we need to assign computation results to registers in a clever way so that we don't use more than 8 registers (which means we cannot use $reg_{8..15}$ for intermediate results for the first 5 pixels) and we also don't lose ILP. For this, we first do the calculations for the cells that are not reused, this way we do not need to store them and can use their register as a space to hold intermediate results. Consider the following scenario:

r_0	r_1	r_2
r_3		r_4
r_5	r_6	r_7

For the convolution of the middle cell, we need:

$$\begin{aligned} H_x =& r_0 - r_2 + 2(r_3 - r_4) + r_5 - r_7 \\ H_y =& r_0 - r_5 + 2(r_1 - r_6) + r_2 - r_7 \end{aligned} \tag{16}$$

We can see that r_1, r_3, r_4, and r_6 are only used once. Thus, we first do:

$$\begin{aligned} r_1 = r_1 - r_6 \text{ then } r_1 = r_1 + r_1 \\ r_3 = r_3 - r_4 \text{ then } r_3 = r_3 + r_3 \end{aligned} \tag{17}$$

Now r_6 and r_4 can be used to store intermediate results:

$$\begin{aligned} & r_4 = r_0 - r_2 \\ & r_6 = r_0 - r_5 \\ & \rightarrow r_0 \text{ is free} \\ & r_0 = r_5 - r_7 \quad \rightarrow r_5 \text{ is free} \\ & r_5 = r_2 - r_7 \quad \rightarrow r_2, r_7 \text{ are free} \\ & r_2 = r_4 + r_3 \\ & r_2 = r_2 + r_0 \quad \rightarrow H_x \\ & r_7 = r_1 + r_6 \\ & r_7 = r_7 + r_5 \quad \rightarrow H_y \end{aligned} \tag{18}$$

We can do the same for $reg_{8..15}$. This way we use all the 16 registers efficiently and also preserve ILP since we still have independent chains of computation.

9. Blocking for convolution. The blocking for convolution in track t_2 is very similar to t_1 with an adaptation to registers being SIMD registers and the energy map data type being shorts instead of integers. When considering the L1 blocking, we have the working set formula becoming:

$$2 * 4 * M + 4 * (M - 2) = C \tag{19}$$

The change is only due to the energy map containing integers, while the image contains shorts. Therefore we have $M = 1141$. With respect to the register blocking we have 16 vector registers where each can contain up to 16 shorts. Due to the way we perform SIMD, we have 10 pixels' convolution (that belong to the same row) done in parallel.

10*. SIMD for minimun seam calculation. When calculating a minimum seam, we use the dynamic progrfamming described in equation 7. The computations for each pixel value are independent from each other so it is suitable for SIMD. We need to calculate minimums from 3 pixels and then calculate an overall minimum from a row. We will not go into the details as the logic and the calculations are rather easy. Furthermore, it did not bring performance gain as when compiled with optimization flags the compiler was able to vectorize this part.

11*. Scalar replacement in convolution and min seam. Scalar replacement was applied on top of the SIMD optimizations presented before, but without runtime or performance gain since the method was already applied by the compiler, with corresponding optimization flags.

12*. Function inlining. We performed function inlining on the last version of our program on track 2. However, we did not get performance gain when we compiled with optimizations enabled.

Other attempts. We considered alternative ways of calculating the partial gradient of an image, i.e rather than using Sobel filters that give a more precise but also more expensive energy map we tried using the change in y and change in x formula, which would be faster. This alternative approach however gave too worsened of an energy map, rendering the small performance boost insignificant.

Regarding blocking, initially it was planned to be done for the entire memory hierarchy, i.e including registers, L1 L2 and L3 cache. However L2, L3 blocking gave no performance boost because their size 256KB and 2MB respectively is larger than any input image.

4. EXPERIMENTAL RESULTS

In this section, we present our experimental setup and the achieved results. We will show the performance gains of the most important optimization steps and refer to them by the number allocated to them in Section 3. For our experiments, as a baseline we used a straightforward implementation of the seam carving algorithm presented in Section 2 and referred to as (1). Our fastest implementation achieved an overall 3.7x (with -O0) and 2.53x (with -Ofast) performance speedup compared to the baseline. Regarding runtime speedup it achieved 9.54x (with -O0) and 4.43 (with -Ofast) compared to the base implementation.The best percentage achieved out of the peak performance is 75% for optimizations that do not include SIMD and 6.25% for optimizations that include SIMD.

Experimental setup. We used two different platforms on which we ran the experiments.

Intel Haswell with 2.4 GHz base frequency and with cache sizes: L1: 32 KiB, L2: 256KiB, L3: 3 MiB. Compiled with gcc (Apple clang version 11.0.0 (clang-1100.0.33 .16)).

Intel Skylake (Kabylake) with 1.6 GHz base frequency and with cache sizes: L1: 32 KiB, L2: 256 KiB, L3: 6 MiB. Compiled with gcc (version 9.3.0).

We divided our experiments to two main groups: when we use optimization flags (-Ofast) and when we do not use compiler optimizations (-O0) and on the plots we will refer to them by these tags. In these two groups we have different compiler flags depending on whether we have SIMD optimizations or not. These optimization flags are presented below:

- `-O0` (no SIMD)
- `-Ofast -fno-tree-vectorize` (no SIMD)

- `-O0 -march=native` (SIMD)
- `-Ofast -march=native` (SIMD)

For the experiments, we used 1.0 aspect ratio images with size in the range of [200, 4000] pixel width and height for Skylake and [200, 4000] pixel width and height for Haswell. As the performance of the algorithm does not depend on the image content, it is enough to specify the aspect ratio of the image we work with.

It is important to note that we will use the notation flops referring to operation count and [flops/cycle] referring to performance, however, in our program we only use integer operations and do not have any floating point arithmetic.

Results.

Baseline. We consider the straightforward implementation of the seam carving algorithm as it is presented in [1] as our baseline (1). In Figure 3 we see the achieved performance when compiled with gcc, -Ofast. We note 3 rapid drops in performance at each of the cache limits, indicating when our data no longer fits in L1/L2/L3 cache respectively. It is worth mentioning that our baseline is compiled with -Ofast rather than -O. We believe this to be a more accurate base because it corresponds more to what a software engineer would realistically do to boost performance of his/her algorithm. Just as a reference the performance on Skylake when compiled with -O0 is around 1 [flop/cycle].

Track 1 optimizations.

As described in Section 3 we divided our optimization work into two main tracks. We see the performance of optimization steps of track 1 on Haswell in Figure 4. We did not include the baseline (1) as it has a significantly different flop count. The baseline on Haswell compiled with flag -O0 has a performance of around 0.6 [flops/cycle]. The performances were gained by compiling the program with -O0 so we are sure that the only optimizations contributing to the performance boost are the ones made by us. The overall trend that can be seen is that each increasing step has better performance than the ones before it. The only exceptions to these monotonic increments across the different optimizations are steps (4) and (5). We believe that this might be due to the register spill that can occur in (5) and not in (4) within the inner most for loop of `calc_energy()` function in `convolution.c`. This is because the 'Computation reuse' optimization uses 18 variables in the inner most for loop whilst there are only 16 scalar registers available. This doesn't happen in the 'Blocking' optimization because the formulas regarding the working set account for this. However, since track t_1 is not the one we continued to improve upon, this small variance between the two steps remained.

Furthermore, as mentioned in Section 3 on track 1 most of these optimizations did not bring additional performance gain when having compiled with -Ofast.

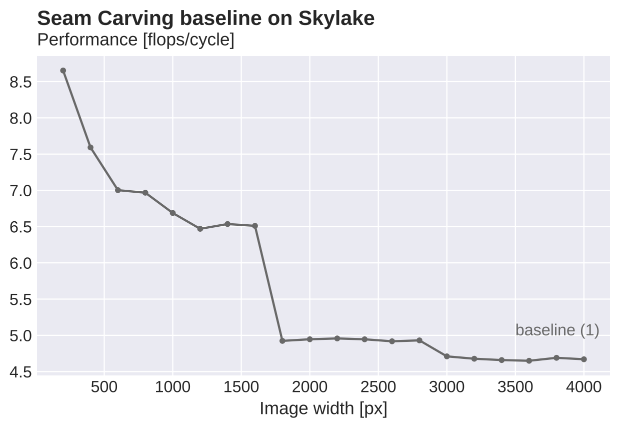


Fig. 3: Performance of the baseline implementation of seam carving(-Ofast).

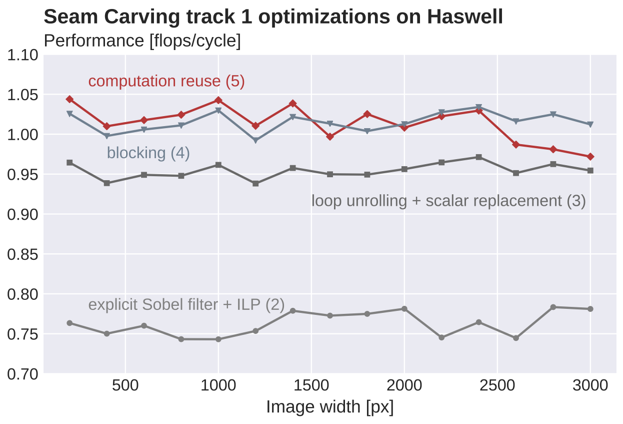


Fig. 4: Performance of the optimizations on track 1 on Haswell(-O).

Track 2 optimizations. After seeing that on track 1 we cannot achieve significant performance gain when compiled with optimization flags (-Ofast), we switched to track 2 and considered it as the main branch on which we can achieve the best performances. Thus on this track we also made experiments on Skylake.

The main optimization steps at track 2 on Skylake are shown in Figure 8. We cannot compare the performance improvements directly to the baseline because we significantly changed the number of operations in the code at the RGB base optimization step (6) . However we see the run time improvement of 3x speedup by switching to an RGB base representation (6) in Figure 6.

Likewise, we can see the main optimization steps and their results in Figure 7 when compiled on Haswell with optimization flags (-Ofast). For the same reason as on Skylake we cannot include on this plot the baseline, which is around 2-3 [flops/cycle] when compiled with -Ofast.

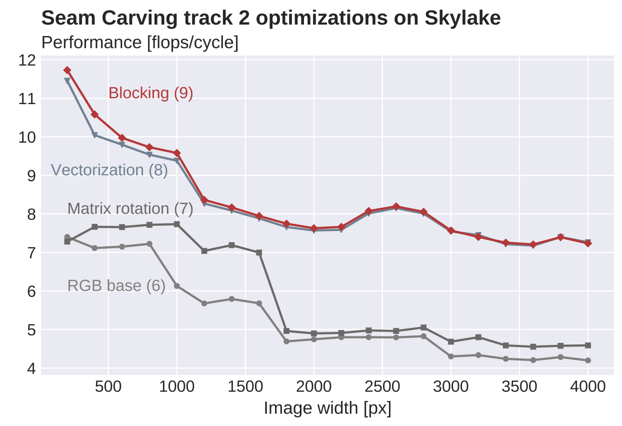


Fig. 5: Performance of the optimizations on track 2 on Skylake(-Ofast).

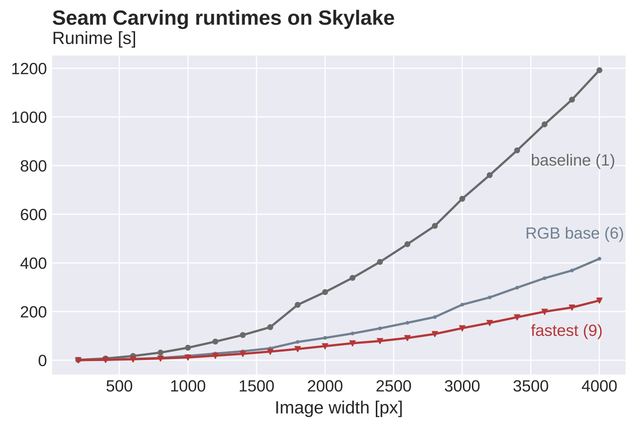


Fig. 6: Runtime improvement when compiled with -Ofast.

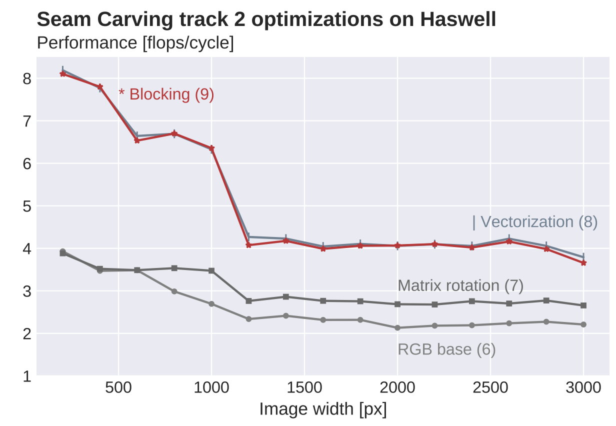


Fig. 7: Performance of the optimizations on track 2 on Haswell(-Ofast).

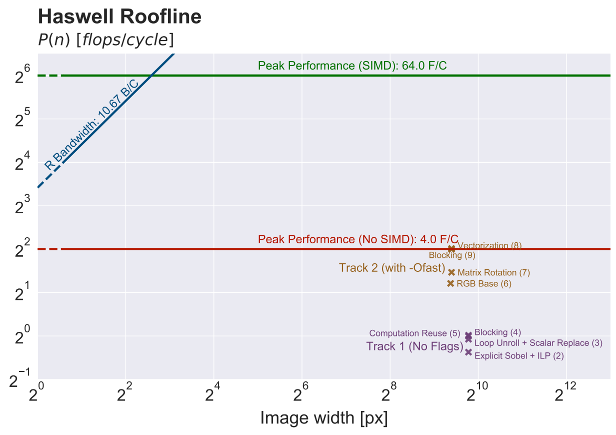


Fig. 8: Haswell roofline.

5. CONCLUSIONS

The results obtained through this project are important in the context of fast content-aware resizing. This is usually required for web content, which can be displayed on multiple different screen sizes.

In this project, we managed to achieve a more performant implementation of the Seam Carving algorithm for image size reduction. Our implementation is dedicated to Intel specific hardware, more specifically the Skylake microarchitecture. The best speedup achieved by our implementation was of 9.54x, compared to a straightforward base implementation. Furthermore, even if the current approach was oriented on size reduction, the optimization techniques could be easily adapted for the more general Seam Carving algorithm.

6. CONTRIBUTIONS OF TEAM MEMBERS

Aydin. `min_seam.c` base implemenation. Manual FLOP count. Performance infrastructure and plots. Code instrumentation. Explore padding alternative solutions. Worked with Tijana on the explicit Sobel filter and ILP for the `calc_energy()` function. Worked with Tijana on blocking for L3, L2 L1 and registers for `calc_energy()` for track t_1. Worked with Tijana on computation reuse in t_1. Worked with Tijana on SIMD for `calc_energy()` in track t_2. Worked with Tijana on blocking for `convolution.c` for L1 and registers for `calc_RGB_energy()` in track t_2.

Dora. Input/output image parsing implementation. Focused on code refactorings. Changed data representation to uchar and to RGB based. Memory management optimizations, optimize order of freeing unused data in optimal image. Focused on optimizations in min_seam.c, like refactoring branches and energy map rotation for better locality.

Worked with Ioana on SIMD for min_seam. Did inlining for track 2. Did runtime and performance measurements, did runtime and performance plots for Skylake.

Ioana. `optimal_image.c` base implementation. Cost analysis and count. Validation infrastructure and plots. perf bottleneck detection. Focused on ILP and scalar replacement for both tracks of the algorithm. Worked with Dora on the SIMD optimization for track 2, for the minimum seam computation. Worked with Dora on restructuring conditional blocks for track 2, in order to reduce bad speculation and operations count.

Tijana. `convolution.c` base implementation. Timing infrastructure. VTune bottleneck detection. Simplified gradient calculation. Code instrumentation. Worked with Aydin on explicit Sobel filter and ILP for `calc_energy()` function. Worked with Aydin on blocking for L3, L2 L1 and registers for `calc_energy()` for track t_1. Worked with Aydin on the computation reuse in track t_1. Worked with Aydin on the SIMD for `calc_energy()` in track t_2. Worked with Aydin on blocking for `convolution.c` for L1 and registers for `calc_RGB_energy()` in track t_2.

7. REFERENCES

[1] Shamir A. Avidan, S., "Seam carving for content-aware image resizing," *ACM SIGGRAPH*, , no. 10-es, 2007.

[2] Zhou N. Paul J. C. Zhang X. Dong, W., "Optimized image resizing using seam carving and scaling," *ACM Transactions on Graphics (TOG)*, , no. 28(5), pp. 1–10, 2009.

[3] Chien S. Y. Hwang, D. S., "Content-aware image resizing using perceptual seam carving with human attention model," *IEEE International Conference on Multimedia and Expo*, pp. 1029–1032, 2008.

[4] Shamir A. Avidan S. Rubinstein, M., "Improved seam carving for video retargeting," *ACM transactions on graphics (TOG)*, , no. 27(3), pp. 1–9, 2008.

[5] Karthik Karanth, "Implementing seam carving with python," https://karthikkaranth.me/blog/implementing-seam-carving-with-python.