ACCELERATION OF LARGE DISPLACEMENT OPTICAL FLOW ESTIMATION

Saiwen Wang, Yifan Su, Yijun Pan, Xinyuan Yu

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

DeepFlow [1] is an algorithm which compute the displacement of a pair of similar images. In this paper we will first explain the basic DeepFlow algorithm as the baseline. After that we will analyze the bottleneck and problems in the code and apply different optimization strategy. In the end, we will show the significant performance speedup in the result.

1. INTRODUCTION

Nowadays, many computer vision systems utilise optical flow detection as an early stage of the pipeline. Cases of small displacement has been extensively studied over the past few decades. That is, the motion of every pixel is limited to a few pixels. However, cases of unconstrained motion have received attention only recently and it remains an open problem [2, 3]. Large displacement optical flow covers the more realistic cases where the motion is not restricted and can be more large than the size of the object. The typical result of optical flow estimation is shown on Figure 1.

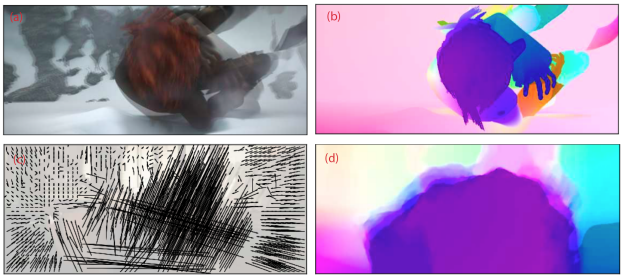

Fig. 1. Result of Optical Flow. (a)Two original overlapped images; (b) Real displacement of the two; (c) Vector representation of the displacement result found by optical flow; (d)Displacement result found by optical flow.

Motivation. An enormous amount of digital video content (home movies, films, TV) is becoming available, along with new challenges like action and activity recognition from realistic videos[4, 5] which further put an emphasis on processing speed. State-of-the-art optical flow algorithm provides a solid theoretical base for the estimation problem. But standard implementations usually work inefficiently on large datasets due to the complexity of updating and solving a dense non-linear system within multiple iterations. In our work, we experimented with several optimization methods and present an implementation which is marginally faster than the unaccelerated version.

Related work. Many state-of-the-art optical flow estimation algorithms work in the framework of variational method. Horn and Schunck[6] introduced a classical way of minimising an energy function based on a data term and a smoothness term. Ever since, research has focused on alleviating the drawbacks of this method. A series of improvements were proposed over the years. Brox and Malik[2] propose to add a matching term penalising the difference between flow estimation and matches. Xu[3] performs an expensive fusion of intensity flow, SIFT feature[7] matches and patch matches[8] at each level of image pyramid. DeepFlow[1] blends a matching term into the energy function and proposes a global matching algorithm, tailored to the optical flow problem, that allows to alleviate the over-constrained smooth terms on fast motions.
Our method improves time and memory efficiency based on DeepFlow's energy function and utilises several modern optimization technology, including blocking and instruction-level parallelism. The experiment result shows an effective acceleration of the implementation.

2. BACKGROUND

In this section, we formally define the energy function of deepflow[1], introduce the algorithms we use and perform a cost analysis.

Energy Function of DeepFlow. Deepflow is a variational version of optical flow, that blends the deep matching algorithm into an energy minimization framework. The baseline for optical flow estimation is to minimise an energy function based on a data term, a smoothness term and

a matching term.

$$E(\omega) = \int_{\Omega} E_D + \alpha E_S + \beta E_M \mathrm{d}x \tag{1}$$

The first term typically assumes that the colour and the gradient are constant along the flow while the second assumes that the flow is smooth. A matching term to penalise the difference between input matches and flow estimation is added to the classic data and smoothness terms. The concrete definition of the three terms is as below. Details are explained in the paper of Deepflow[1].

$$E_D = \delta\psi(\sum_{i=1}^{c} \omega^T \overline{J}_0^i \omega) + \gamma\psi(\sum_{i=1}^{c} \omega^T \overline{J}_{xy}^i \omega) \tag{2}$$

$$E_S = \psi(\|\bigtriangledown u\|^2 + \|\bigtriangledown v\|^2) \tag{3}$$

$$E_M = c\Phi\psi(\left\|w - w^{'}\right\|^2) \tag{4}$$

The algorithm flow is then drawn as Figure 2.

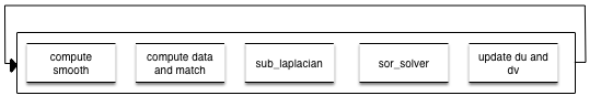

Fig. 2. Algorithm Flow of One Iteration.

The energy function is then minimized throw a coarse-to-fine scheme during iterations. The flow is estimated at a coarse version of the images and is iteratively refined. First part of each iteration computes the above three terms. It is then optimised using fixed point iterations and classic linear system solver such as SOR(Successive Over Relaxation).

Cost Analysis. For cost analysis, we measure flops as (addition, multiply, division, square root). We count each type of operation for simplicity. In fact, divisions and sqrts both cost 28 cycles. Regarding them as one flop may underestimate flop count, which leads to lower performance.
The cost of solver function is $41MNK_iK_mK_o$ flops with M, N as the height and the width of the input image, K_i, K_m, K_o as the numbers of inner, middle and outer loops. The compute function costs $263MNK_mK_o$ flops.
The time and memory complexity of the two functions we optimised is shown in Table 1, where m = width of input image(pixel), n = height of input image(pixel), K_{pyr} = pyramid level of each iteration, K_{inner} = inner loop of each iteration, K_{solver} = inside loop of solver function.

Function	Time/Memory Complexity
Sor Solver	$O(K_{pyr} * m * n * K_{inner} * K_{solver})$
Compute Data and Match	$O(K_{pyr} * m * n * K_{inner})$

Table 1. Time & Memory Complexity

3. METHOD

In this section we will present our step-wise implementation of the algorithm and indicate detail optimization methods.

Baseline. The baseline implementation of the algorithm is given by the authors of the paper [1]. It's implemented in a naive manner and no optimization method is performed.

The core algorithm subsamples the pair of original input images into a stack of different size of images, which looks very similar to the pyramid. For each pair corresponding subsampled images in the pyramid, both the first and second order derivative is computed and the flow will be computed in iterations. In each of the iteration, The smoothing term will be first computed and then served to the matching computation. After that a linear system will be solved and after each iteration the offset (displacement) will be updated.

All these algorithm will first go through the smallest image in the pyramid, and then the computed displacement of the previous images in the pyramid will become a base which helps to compute the displacement of the next image. In the end, the displacement of the original image will be computed and the algorithm terminated.

Bottleneck. After implementing the baseline version of the algorithm, we tried to analyze the performance bottleneck. By using google profiler, the plot of the calling graph below can be helpful to find the performance hotspot.

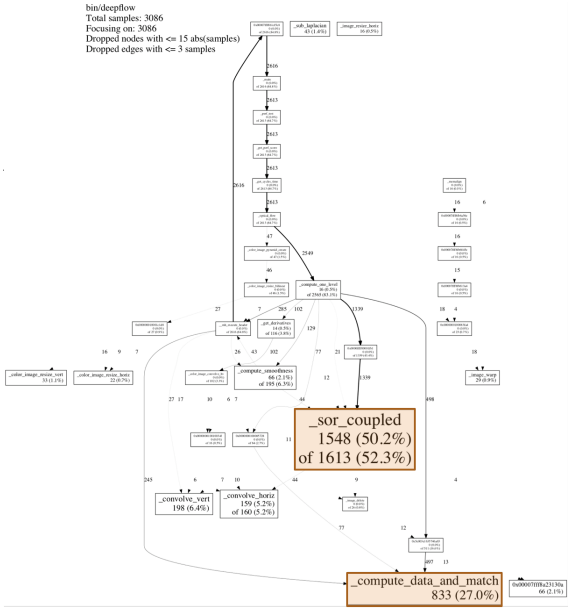

Fig. 3. Bottleneck of Algorithm Flow. Time consumption precentage is showed in each function box.

It's obvious that the majority of the time is consumpted in function `sor_solver` and `compute_data_and_match`. So in the following part of the optimization steps, we will

focus in optimizing these two functions.

Basic Optimizations. We first apply some basic optimization methods to the code.

Code motion. We observe some repeated division sub-expressions in the code. It is known that division operations has lower throughput and requires longer cycles to finish computation. By replacing division sub-expressions with precomputed shared values, the number of computation flops is expected to be lower.

Scalar replacement. We observed that the code requires large amount of memory access. The original image and sub-sampled images, helper values includes first and second order derivative and displacement results are all stored in memory. In each inner iteration, multiply values in the memory are accessed multiple times. We apply scalar replacement to memory location being accessed. Thus repeated memory accesses are reduced to two (one for loading and one for storing).

Memory and locality. *Reuse memory.* As indicated above, the algorithm requires large amount of memory. It is noticeable that for each single (possible sub-sampled) image, 23 helper data structures with the same size of the image will be created and accessed. The large amount of momory space is too large to be fitted in either the first level cache or the second level.

In order to reduced the possible data transfer between cache and memory, we try to reduce the number of the duplicated helper data structures. We decide to use a unique set of helper data structure for all size of sub-sampled images, instead of creating a different set for each of sub-sampled images.

After the reuse of memory, we expect the memory of the helper data structure is already sitted in the cache when revisiting them in the next iteration, which introduce temporal locality.

Unrolling. In each of the inner iteration, the data of input image and helper data structure is accessed one by one in a row order. For the solver part, not only the corresponding pixel is access, the nearby pixels (the upper, lower, left and right, which form like a cross) are also accessed. We observed a strong spatial locality here.

As indicated above, the large number of helper structures and possible data alignment can cause repeated cache misses when accessing neighbour pixel even if strong spatial locality is observed. We use unrolling the increase the spatial locality and try to load as much as possible in each transferred cache block and seperate the memory access and computation.

For function `compute_data_and_match`, only the corresponding pixel is accessed and there's no dependence between each pixel. So basic unrolling is performed and 16 pixel of `float` data is loaded from every helper structure, computed and then written back.

For function `sor_solver`, accessed pixels are not only neighbour pixels in the same row, but also some pixels in the upper and lower row. So the unrolling is done by blocking, which introduce blocking size of either 2 by 2 or 4 by 4. So the expected memory access can be reduced as the data in each cache block can be accessed at once instead of possibly repeated access and transferred.

Vectorization. In the code we observed that same computation is done pixel by pixel in the inner iteration. So vectorization is a good possible way to speed up computation.

First we change the compiler flag to enable automatically vectorization. And then we introduce intel AVX [9] intrinsic to manually performs vectorization. Neither the input images nor the sub-sampled image promise row alignment of 8 `float`. We add build-in row alignment of 8 so that all the images are forced to have row length divisible by 8.

In function `sor_solver`, neighbour pixel computations are correlated, so AVX is not performed on the major part if the code, but only to the part of matrix operation.

4. EXPERIMENTAL RESULTS

In the experiment part, we show the performance plots and roofline model of the optimized algorithm.

Experimental setup. We setup experiment on a machine with Sandy Bridge processor. The compiler is gcc with version 4.9.2 on Redhat Linux. We use `-O0` and `-fno -tree-vectorize` as compiler flag firstly and then change it to `-O3` and `-core-avx-i` in the latter experiment. The input of our program is a pair of images with slight difference. In our experiment, we test with images in a range from 26×64 to 4096×1716

Results. Since we optimize two functions in the algorithm, we show the results of each optimized function separately.

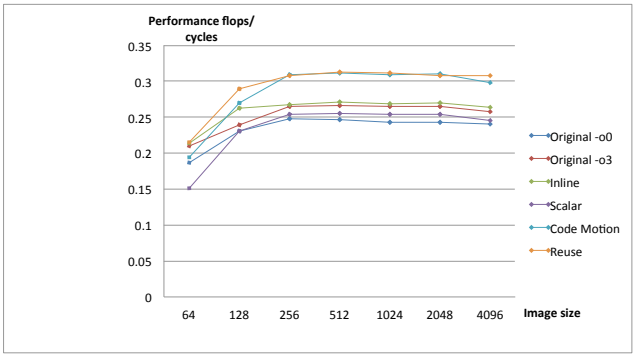

Fig. 4. performance plot of Compute Data and Match without AVX

Fig 4. is the performance plot of Compute Data and Match without AVX. We can see the performance is improved slightly after scalar replacement and code motion. In Scalar replacement part, we reduce duplicate memory access. In the code motion part, we precompute share division components, which is very expensive. Then, we change the compiler flag to -O3 and -core-avx-i which improves the performance by 25%.

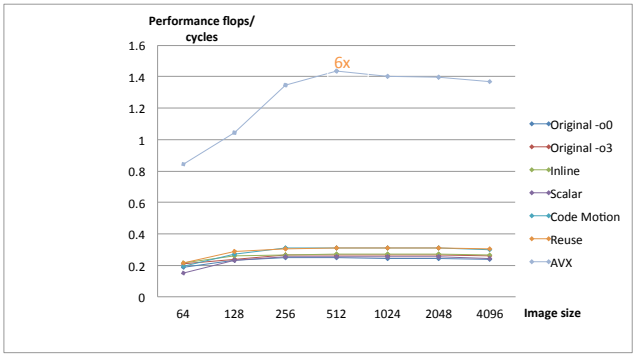

Fig. 5. performance plot of Compute Data and Match with AVX

In Fig 5. we can see that the code benefits from blocking and parallel computing by using AVX. The performance is improved significantly. Since we test on a Sandy Bridge machine, the peak performance is 16 flops/cycle. The final performance we get is 1.4 flops/cycle, which is 6 times of the performance of the original code. It is about 8.8% of the peak performance.

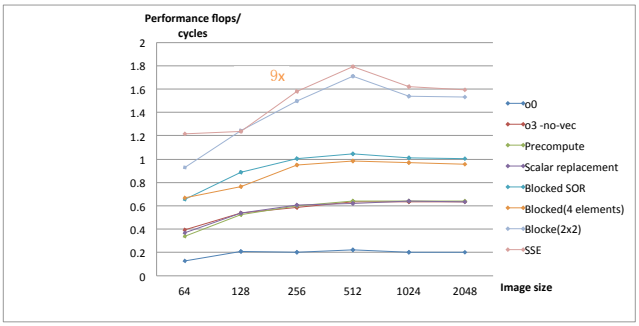

Fig. 6. performance plot of solver

In Fig 6. we can see that there are three optimizations which improve the performance significantly, which are changing compiler flag from -o0 and -fno-tree-vectorize to -o3, blocking and blocking with 2×2 mini block.

The code can benefit from better locality by using mini blocks with the size of 2×2. However, in the function of solver, only a little part can be changed to SSE. Thus the performance is not improved much by the optimization of SSE. In this way, it is meaningless to optimize the function using AVX as well.

In the part of solver function, the performance is improved to 1.8 flops/cycle, which is 9 times of the original performance(0.2 flops/cycle). Since the peak performance of Sandy Bridge is 8 flops/cycle with SSE. Thus, it is 22.5% of the peak performance.

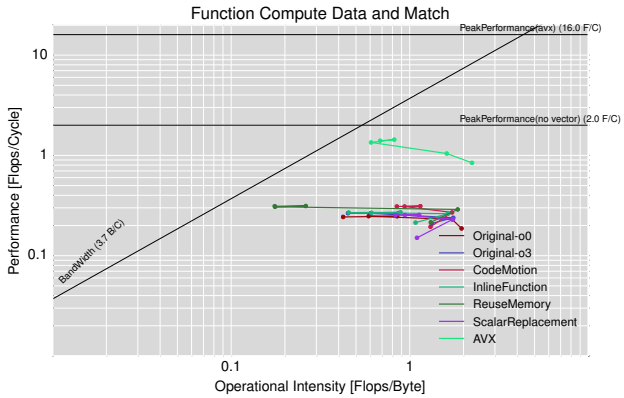

Fig. 7. roofline of compute data and match

Fig 7 is the roofline model of compute data and match function. In the roofline model, we can see that the intensity is affected by the size of input image. If small image is used, most of data can be saved in cache. We have less data transfer and the intensity is higher than using large image. Besides, in our algorithm, there are several division and SQRT operations, which are expensive. If we re-weight one division or SQRT operation by 16 flops (a div or SQRT operation costs 16 cycles while mul costs only 1 cycle), the performance is close to peak performance. Therefore, our algorithm is computational bound.

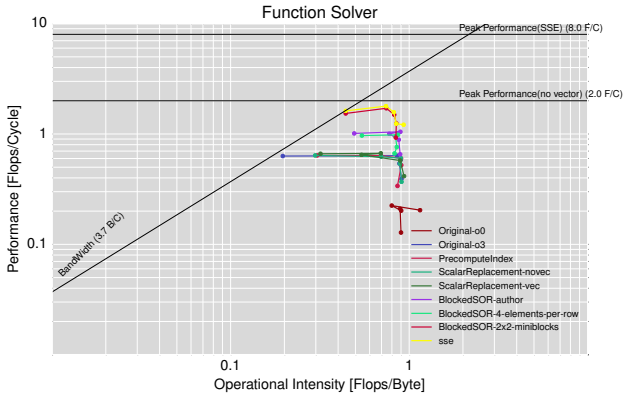

Fig. 8. roofline of solver

Fig 8 is the roofline model of function of solver. In this model, after re-weighting the division operation, the perfor-

mance is close to the peak performance. Although, when using large image size, the point is close to the memory bound, we can see the performance is not affected. Thus, we can conclude that the performance is not affected by memory transfer. It is also computational bound in this part.

5. CONCLUSIONS

By applying our optimization method to the baseline algorithm, we have gained significant performance increase. We have gained up 5.6 and 7 time speedup in two bottleneck functions we mentioned above. Even through we have only focused in optimizing the two bottleneck functions, the overall performance of the deepflow algorithm is significantly increased.

Our optimization of the deepflow algorithm can be very helpful to any applications which requires expensive image displacement computation. The key algorithm itself can also be applicable to similar image processing algorithms.

In the next step, we expect to focus in minimizing memory footprint of our algorithm. Also we will try to find an approximated algorithm which can introduce similar image displacement computation, but with much less computation time.

6. REFERENCES

[1] Philippe Weinzaepfel, Jerome Revaud, Zaid Harchaoui, and Cordelia Schmid, "Deepflow: Large displacement optical flow with deep matching," in *Proceedings of the 2013 IEEE International Conference on Computer Vision*, Washington, DC, USA, 2013, ICCV '13, pp. 1385–1392, IEEE Computer Society.

[2] Thomas Brox and Jitendra Malik, "Large displacement optical flow: Descriptor matching in variational motion estimation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 3, pp. 500–513, Mar. 2011.

[3] Li Xu, Jiaya Jia, and Y. Matsushita, "Motion detail preserving optical flow estimation," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, June 2010, pp. 1293–1300.

[4] I. Laptev and P. Perez, "Retrieving actions in movies," in *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, Oct 2007, pp. 1–8.

[5] Heng Wang, A. Klaser, C. Schmid, and Cheng-Lin Liu, "Action recognition by dense trajectories," in *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, June 2011, pp. 3169–3176.

[6] Simon Baker and Iain Matthews, "Lucas-kanade 20 years on: A unifying framework," *Int. J. Comput. Vision*, vol. 56, no. 3, pp. 221–255, Feb. 2004.

[7] Richard Szeliski, *Computer Vision: Algorithms and Applications*, 2010.

[8] Connelly Barnes, Eli Shechtman, Dan B. Goldman, and Adam Finkelstein, "The generalized patchmatch correspondence algorithm," in *Proceedings of the 11th European Conference on Computer Vision Conference on Computer Vision: Part III*, Berlin, Heidelberg, 2010, ECCV'10, pp. 29–43, Springer-Verlag.

[9] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo, "Intel avx: New frontiers in performance improvements and energy efficiency," *Intel white paper*, 2008.