CS 605: Parallelizing Scale-Space Representations

Basile Hurat

April 2020

1 Introduction

Image processing and computer vision are two fields that benefit a lot from parallelization, as the combination of high resolution images and the need for "real time" processing leads to large problems that need fast solutoins. Scale-space representations are not necessarily a tool you use for "real time" processing, but they are fantastic candidates for parallelization. Scale-space representations, or fine-to-coarse representations, are a tool with quite a bit of flexibility. These representations can be given by the convolution of a smoothing kernel $T(., t)$ and an input function, f, and their 2D discrete form can be given by

$$L(x, y; t) = \sum_{m=-\infty}^{\infty} T(m; t) \sum_{n=-\infty}^{\infty} T(n; t) f(x - m, y - n).$$

For our applications, we use the sampled Gaussian kernel, $T(m, t) = \frac{1}{\sqrt{2}} e^{-m^2/2t}$ While commonly used as a way to detect appropriate scales for template matching, scale-space representations have another very useful property: The non-creation of local extrema. As such, you can use scale-space representations to define the persistence of extrema, allowing for classification of important extrema versus unimportant extrema. My thesis research uses this and as such, I implemented scale-space representations. However, scale-space representationz require many convolutions, and it was a bottleneck of my code and as such, I wanted to parallelize it. However, we must begin by parallelizing convolutions.

2 Convolutions with CUDA

Convolutions are costly operations. If you convolve $f * g$, then for each of the samples of f, you must perform a multiplication and addition operation for each sample in g. You can reduce this slightly if do zero paddings and ignore the operation, but overall for an image f that is $M \times N$ and a kernel g that is $m \times n$ you have a complexity of $\mathcal{O}(MNmn)$. However, each pixel is independent and as such, you can use CUDA to calculate the convolutions of each pixel independently.

To see how this parallelization performs, we perform tests. We use two images: a smaller (340×405) image of a cat, and a larger (3024×4032) image of a sculpture. For our kernel, we use the sampled Gaussian kernel discussed earlier, with a window size of 13×13 and a t value of $t = 1.0$. We consider multiple grid sizes, and the results are shown in Table 1. What we see is that for the smaller image,

	Serial	CU 1×1	CU 8×8	CU 16×16	CU 24×24	CU 32×32
Cat	0.19379	0.01223	**0.0004656**	0.0004706	0.0005005	0.00058396
speedup		15.84	**416.2**	411.8	387.2	331.9
Two	16.68	1.0033	0.03400	0.03396	**0.03391**	0.03415
speedup		16.63	490.7	491.2	**491.9**	488.5

Table 1: Convolution speeds (in seconds)

a smaller blocksize is better. This is possibly due to the fact that the dimensions are not divisible by 32, and since warps have 32 threads per warp, this is less effective. Overall, we see a speedup of 410x faster than the serial program with block sizes of 8×8 and 16×16 performing well . For the larger image, we see that outside of the blocksize of 1, we see similar speedups for all block values tested, which makes more sense as the dimensions of the larger image are divisible by 32. Also, we see a faster speedup on the order of 490x faster than the serial program. Figure **??** shows the result of this convolution, which shows a slightly blurred image of the input.

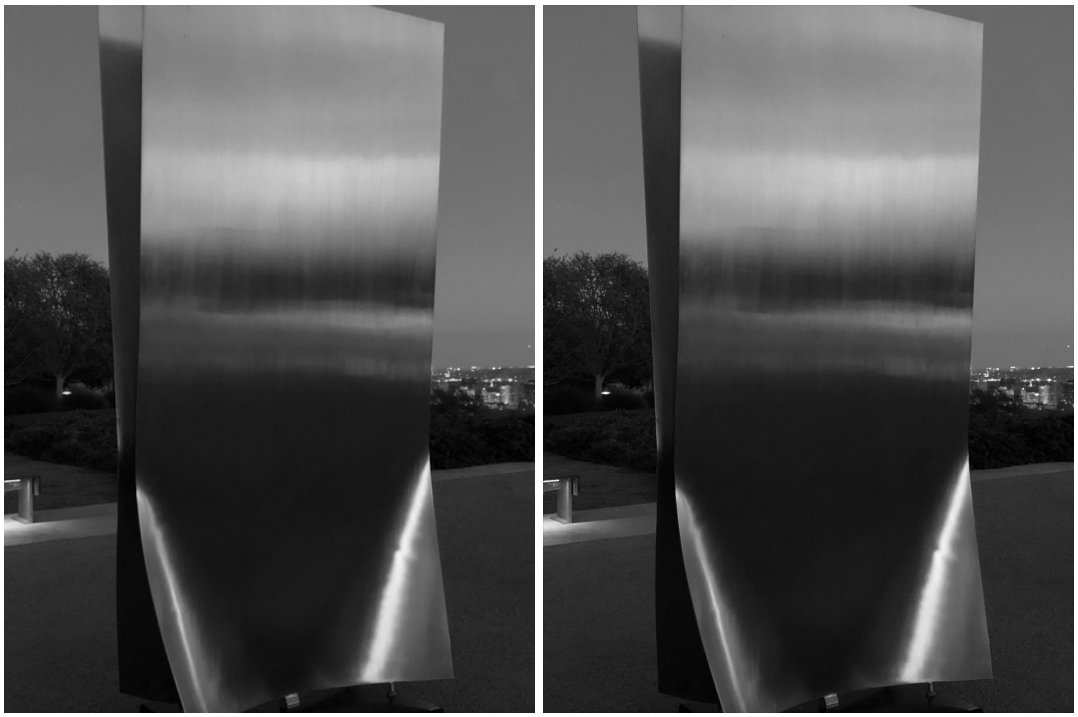

Figure 1: Results of Convolution

3 Scale-space Representations Using Hybrid Parallelizations

With convolution explored, we dive into scale-space. Scale-space representations are achieved in the discrete case by picking a step-size, t_0 and performing the convolution for a certain number of iterations, M_i and building the smoothing kernel for $t = t_0, 2t_0, \ldots, M_i t_0$. As such, we can modify our convolution code by simply adding an outer loop. However, since fine-to-coarse representations are always applied as a step into an algorithm, I wanted to simulate this. As such, inside the loop, I included the saving of the image to the disk and added that to the timing.

When it came to the parallelization, I considered multiple approaches. The first was to simply use CUDA for the convolution, and keep the outer loop serial. The second attempted to use OpenMP to distribute the outer loop to multiple CPUs while using CUDA for the convolution. The last attmepted to use MPI to distribute the work of the outer loop to two nodes and use both GPUs simultaneously.

For the serial, CUDA and OpenMP-CUDA implementations, we perform our tests on node 10 and for MPI-CUDA implementation, we perform on nodes 10 and 11. We perform the scale-space representation on the smaller image of the cat and use an initial step-size of $t_0 = 1.0$. For our CUDA blocksize, we use 16×16. The results are shown in Table 2.

There are two main observations. The first is that the bulk of the CUDA speedup is lost due to

	Serial	CUDA	OpenMP-CUDA	MPI-CUDA
Cat	41.637	7.6947	7.2727	**5.3299**
speedup		5.411	5.725	**7.812**

Table 2: Scalespace speeds (in seconds)

Figure 2: Scale-space Representation of Cat.jpg at $t = 1, 32, 64$

the large portion of the time being spent saving the images. However, we are still able to get speedup. Also, while OpenMP did not significantly speed up our scale-space program, the combination of MPI and CUDA performs the best, with a speedup on the order of 7x faster than serial and 1.4x faster than CUDA alone. Figure 2 shows the results of the scale-space representation.

4 Conclusion

In conclusion, we noticed the power of CUDA for parallelizing 2D convolutions. We also saw its limits by including code that could not be performed on the CPU. We looked to get around this using hybrid parallelizations, which worked to improve speedup. The OpenMP-CUDA implementation was only a minimal improvement but the MPI-CUDA showed a more significant improvement, though it could also be due to the use of double the GPU.

There is future work to consider if given more time. Firstly, the sampled Gaussian kernel is separable, and implementing a CUDA implementation of a separable 2D convolution would be even faster than we showed here. Also, the saving of resulting images is not particularly representative of the uses of scale-space and has many problems when it comes to correct timing. I would be curious to consider a proper local minima algorithm instead, which I imagine is recursive and thus not able to be coded in CUDA.