REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

Full GPU Acceleration of Local Descriptors using CUDA

Kristleifur Daðason, Herwig Lejsek, Björn Þór Jónsson, Laurent Amsaleg

RUTR-CS09003 — July 2009

Reykjavík University - School of Computer Science

Technical Report

ISSN 1670-5777

Full GPU Acceleration of Local Descriptors using CUDA

Kristleifur Daðason*, Herwig Lejsek*, Björn Þór Jónsson†, Laurent Amsaleg‡

Technical Report RUTR-CS09003, July 2009

Abstract: Video analysis using local descriptors demands a high-throughput descriptor creation process. The most practical method to this goal is to use GPUs that come with most recent computers. In this report, we adapt the computation of the Eff^2 descriptors, a variant of SIFT, to the GPU. We compare our GPU-Eff^2 descriptors to SiftGPU and show that while both GPU-based variants yield similar results, the GPU-Eff^2 descriptors require significantly less processing time.

Keywords: Local image descriptors, GPU, Eff^2, SIFT, CUDA.

(Útdráttur: næsta síða)

* Eff2 Technologies ehf., Kringlan 1, IS-103 Reykjavík, Iceland. kristleifur/herwig@eff2.net
† School of Computer Science, Reykjavík University, Kringlan 1, IS-103 Reykjavík, Iceland. bjorn@ru.is
‡ IRISA–CNRS, Campus de Beaulieu, 35042 Rennes, France. laurent.amsaleg@irisa.fr

Heildstæð GPU hröðun á útreikningi staðværra lýsinga með CUDA

Kristleifur Daðason, Herwig Lejsek, Björn Þór Jónsson, Laurent Amsaleg

Tækniskýrsla RUTR-CS09003, Júlí 2009

Útdráttur: Greining myndbanda með staðværum lýsingum (e. local descriptors) krefst mikilla afkasta við gerð lýsinganna. Skilvirkasta leiðin að þessu marki er að nota GPU myndvinnslugjörvann sem fylgir öllum nýrri tölvum. Í þessari skýrslu lögum við útreikninga Eff^2 lýsinganna, sem eru tilbrigði við SIFT lýsinga, að GPU. Við berum GPU-Eff^2 lýsingana saman við SiftGPU og sýnum að þótt bæði GPU-byggðu tilvikin gefi svipaðar niðurstöður, þá þurfi GPU-Eff^2 lýsingarnir mun minni vinnslutíma.

Lykilorð: Staðværir myndlýsingar, GPU, Eff^2, SIFT, CUDA.

(Abstract: previous page)

Contents

1 Introduction

Video analysis is a central component in many applications, such as video surveillance, news analysis, and video copyright protection. Recent methods for such analysis are typically based on computing many local descriptors per frame; these frame-based descriptions are then merged to form the video description. As many applications require real-time performance, high demands are made on the efficient computation of the local descriptors, particularly on high throughput.

The traditional method for achieving high throughput is using a cluster of multi-core computers. The advantage is that the same description code can be used, but there are two major disadvantages. First, merging the local descriptors in the correct order may require major coding efforts for synchronization. Second, computing clusters are very expensive and it is difficult in practice to deliver cluster-based software products to end users.

An alternative method, solving both problems, is to use powerful graphics processing units (GPU). The advent of highly scalable and parallel, yet inexpensive, GPUs has constituted a minor revolution in the computer industry; many projects have therefore investigated the use of GPUs for a variety of tasks, such as image processing [1], feature tracking [6] and local descriptor computation [7].

The disadvantage of GPUs, however, is that the description code does not work unchanged. In fact, adapting computations to GPU has been considered a complex and time-consuming task. Data and computations had to be adapted to meet severe constraints on the access patterns and operations available on-GPU. As a result, some computational processes could not be completely adapted, forcing data loadback to the host CPU for completion (e.g., see [7]). Fortunately, however, GPUs have now become much easier to utilize due to the recently released CUDA programming environment from NVIDIA [2]. The CUDA model allows a larger set of access patterns to data, and supports a large set of computing primitives. While the learning curve is still significant, adaptation to GPU has become easier and more complex computations made tractable.

In this paper, we adapt the computation of the Eff^2 descriptors, a variant of SIFT, to the GPU through the CUDA environment. We compare the GPU-Eff^2 descriptors to SiftGPU [7], another GPU-based variant of SIFT, and show that while both GPU-based variants yield similar results (better than SIFT, and comparable to Eff^2), the GPU-Eff^2 descriptors require significantly less processing time.

2 Background

In this section, we review the state of the art in local descriptor processing, as well as previous efforts to create GPU-based local descriptors. We end with a brief discussion of the new CUDA programming environment from NVIDIA, which we have used for our implementation.

2.1 SIFT and Eff^2 Descriptors

The SIFT local descriptors (Scale Invariant Feature Transform), developed by Lowe [5], have been considered the state of the art in image description for robotic vision and various other computer vision and image processing applications. The creation process consists of two steps, and can be roughly outlined as follows.

In the first step, keypoint detection, small regions of interest are detected where descriptors may potentially be created. This is done by creating a sequence of gaussian blurs at different scales, taking their differences, and then detecting local minima and maxima in the differences across the scales. The keypoint is then localized precisely to a sub-pixel accuracy. In the second step, the descriptor itself is created. First, the dominant gradient angle (direction of cegreatest contrast) around the keypoint is found. Then a 4×4 grid is created around the keypoint, aligned to the dominant gradient angle, and a gradient histogram of 8 bins created in each cell of the grid, obtaining a $4 \times 4 \times 8 = 128$ dimensional histogram of gradient strengths, finally normalizing it.

In this paper, we focus on the Eff^2 descriptors, proposed by Lejsek et al. [3], which are a more scalable variant of SIFT. There are three key differences to SIFT. First, relatively more descriptors are created at higher scales, as more scales are considered and gamma correction is applied to the increasingly coarse blurs. Second, a 3×3 grid is used around the point, resulting in $3 \times 3 \times 8 = 72$ dimensions in total. Third, advanced filters remove descriptors for lines and bright spots, that appear in very many images and have little information content (similar to common words in information retrieval). Several other minor changes were made, including parameter tuning, to improve the recognition of various image transformations. The Eff^2 descriptors have been shown to perform significantly better than SIFT for most image transformations [3].

2.2 GPU-based SIFT Descriptors

SiftGPU is an implementation of SIFT for GPU [7]. SiftGPU uses the parallel capabilities of the GPU to build the gaussian blur pyramids and detect keypoints. SiftGPU then uses a mixed GPU/CPU method to build compact keypoint lists. Finally keypoints are processed in parallel to compute the dominant gradient angle and the descriptor.

The SiftGPU code runs on GLSL, CG and CUDA; it is thus not optimized particularly for CUDA and despite our best efforts we were unable to run the code on our Linux-based system. As will be demonstrated in the experimental section below, the SiftGPU descriptors are orders of magnitude faster than the CPU based SIFT descriptors, and also yield better results.

2.3 The CUDA Programming Environment

A GPU supporting CUDA has multiple identical general-purpose single-instruction-multiple-threads (SIMT) parallel processors [2]. Each processor is capable of most regular CPU

operations. Some are performed very quickly (e.g., most floating point math and bit operations), while other operations are slower (e.g., integer modulus). The SIMT-processors are organized into several multiprocessors. Each multiprocessor must run the same instruction sequence on each of its sub-processors, and processing is thus arranged into groups of one or more multi-processors. Each of these groups is called a *block*, consisting of individual *threads*. Typically, each block processes a certain sub-segment of memory, and each thread is responsible for processing an atomic part of the problem based in that sub-segment. The arrangement of blocks is called a *grid*.

The CUDA GPU has its own on-device memory hierarchy with several types of memory each having different characteristics. The point of the different memory types is to keep the multiprocessors fed with data, in the manner most appropriate to each data access problem. At the bottom of the hierarchy is the *global device memory*, which is typically 512MB–1GB. This memory has high latency but also high throughput. For faster access, but to smaller quantities of data, there is fast but volatile random-access *shared memory*. This memory is typically 16KB, which are divided among live thread blocks. Then there is a set of very fast, low-latency, per-thread *registers* for small data that is local to each thread. For certain operations, CUDA also provides a spatial-locality cache on top of global memory, called *"texture" memory*. Finally, there is a small single-address broadcast-cached *constant memory*, which is essentially read-only, and is used when all simultaneously live threads in a block need to read the same small piece of data at the same time.

In summary, we have found that programming with CUDA consists of making "educated guesses" about the number of threads and type of memory to assign to each part of the problem. With more experience those guesses improve, resulting in better performance. CUDA provides a profiler which can be used to determine the worst bottlenecks. By addressing those bottlenecks in sequence, a reasonable solution is typically found in a reasonable time.

3 Implementation

In this section we describe our implementation of the Eff^2 descriptors within CUDA. We first present the keypoint extraction process and then our major contribution, which is the parallelization of the descriptor extraction process to fully use the capabilities of the GPU. This parallelization involves some changes to the descriptor extraction part which retain full descriptors quality while yielding significant speed improvements.

3.1 Keypoint Extraction

The scale space creation process is relatively easily parallelized, as the process of repeated gaussian blurring of octaves and the subtraction of adjacent blurrings is a prime example of where each thread can be applied to produce pixels independent of other threads.

Once the rough keypoint location is determined, it must be localized as precisely as possible. In the CPU based implementation, this is done by shifting the points by fractional amounts in the two spatial dimensions, and sometimes by moving between scales in the scale

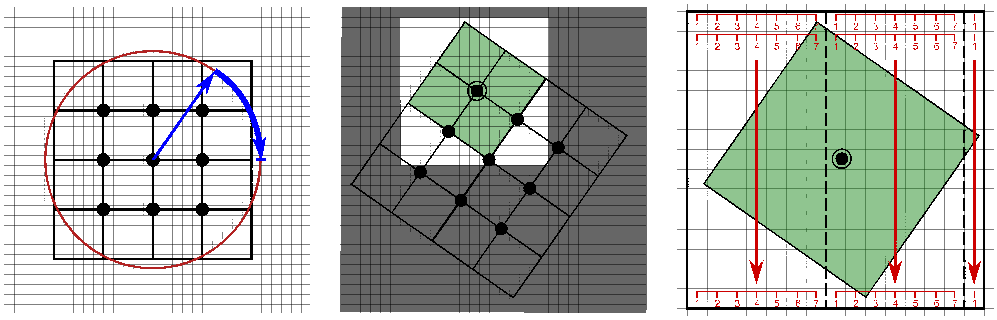

Figure 1: Details of the Descriptor Extraction

space. In the CUDA implementation, however, the two-dimensional shift is implemented, but not the scale-space jumps.

The reason for this change is as follows. On GPUs it is efficient to assign a certain segment of problem space to each thread. In our implementation, one thread's processessing is done in one scale, in one octave. Allowing jumps out of the scale being processed requires either a synchronisation step or significantly more complex interest point detection processing. We determined, through experimentation, that scale jumps increased execution time significantly with no measurable accuracy benefits.

3.2 Descriptor Extraction

While building up a scale-space of an image has been discussed in several papers before, descriptor extraction on the GPU has been too complex for the previously available interfaces. The first of two processing steps in descriptor extraction is determinnig the dominant gradient angle around the interest point. Next, the descriptor histogram is computed using that angle. The following description of the process is aided by Figure 1.

3.2.1 Dominant Angle Determination

To identify the dominant gradient angle around the interest point, we must compute a gradient orientation histogram within a circular window around the point (Figure 1(a)). Note that histogram computation is considered difficult to parallelize, as writes to histogram buckets must be co-ordinated to avoid write conflicts, limiting effectiveness on the GPU. In order to make effective use of the parallelization capabilities of the GPU, we have 1) created a process that works in thread-shared cache as much as possible, and 2) tweaked the gradient histogram itself to allow more efficient processing.

The histogram computation proceeds as follows. First, we take the smallest bounding box that fits the circle and process every pixel row in parallel. The gradient orientation of each pixel is weighted based on its distance to the center point (using weights stored in

the constant device memory) and the gradient strength of each pixel is split up among two adjacent orientation buckets. For each pixel in the row the two bucket identifiers, and their corresponding gradient strenghts, are stored in a shared memory block on the GPU. Then the threads are synchronized after each row and two single threads sum up the processed values into the histogram. Note that using two threads does not result in write conflicts, as by design, writes occur to two independent buckets of the histogram, while using additional threads would give rise to write conflicts. Once all the bounding box' rows have been completed, a single thread calculates the strongest angle from the histogram.

The gradient orientation histogram was tweaked slightly, by reducing the number of orientation bins from the SIFT standard of 36 buckets to 32 buckets. While this change grows the bucket radius from 10 to 11.25 degrees, the already-present interpolation between buckets means that the influence on detection accuracy is less than 2%. The efficiency benefits due to this small implementation tweak are significant, however. On the GPU, integer modulus is a very expensive operation (140 clock cycles on our current GPU) while bitwise AND, which can be used in place of modulus with 32 bins, is much cheaper (only 16 clock cycles). The efficiency benefits of this minor change, which is also used in the descriptor histogram creation, are thus around 10% of the dominant angle determination.

3.2.2 Descriptor Histogram Computation

Once the strongest gradient direction has been found, the shape signal must be encoded into a 72-dimensional descriptor, which is created by computing a gradient histogram of 8 buckets for each block of the 3×3 grid - shown in Figure 1(b). As each subhistogram is calculated from a specific area around the point, all 9 areas can be processed in parallel by a separate set of threads.

For each block, we must first calculate the border vertices of those areas with respect to strongest gradient direction computed above. Each of the 9 threads rotates the four bounding vertices and determines the smallest pixel-aligned bounding box of the rotated square; this area is highlighted in Figure 1(b) and shown in detail in Figure 1(c). Within this bounding box, seven independent parallel threads process the gradient strength with respect to the distance to the actual interest point and the center of the actual cell. The computation of the gradient histogram then proceeds as in this histogram computation described above for the determination of the strongest gradient. The pixels are thus read in the order shown in Figure 1(c).

The reason that seven threads are applied to each histogram, is that there are nine areas processed in parallel, and $7 \times 9 = 63$, which is almost a full set of 64 threads. We experimented with assigning 14 threads to each histogram (for a total of $14 \times 9 = 126$ threads) but the process was slower due to less efficient register usage, as each thread must store some local information. This trade-off, of course, may change as the hardware develops.

Finally, after all $9 \times 8 = 72$ orientation buckets have been calculated, a single thread normalizes the descriptor.

	SIFT		Eff^2	
	descriptors	time [ms]	descriptors	time [ms]
CPU	1,212	1,200	698	360
GPU	486	37	686	27

Table 1: Descriptor creation statistics per image.

4 Experimental Evaluation

In this section we present the results of our experimental evaluation of the four descriptor variants: SIFT, SiftGPU, Eff^2, and GPU-Eff^2. All experiments have been performed on a desktop computer equipped with an Intel Q6600 processor and a NVIDIA GTX 280 GPU. First, we compare and analyze the time spent on descriptor extraction, and then we study the result quality with each of the four variants.

4.1 Extraction Time

In this experiment, we applied each of the variants to a collection of 29,277 high-quality news images [3], where each image's longer edge is rescaled to 512 pixels. Table 1 shows the number of descriptors created and the running time. The descriptors of each image are computed three times, and the average time of the second two runs is used, in order to avoid measuring initialization effects that are common between the GPU-based variants.

As Table 1 shows, descriptor creation on the GPU is almost an order of magnitude faster than Eff^2 on the CPU and nearly another order of magnitude faster than SIFT. Furthermore, GPU-Eff^2 performs descriptor creation significantly faster than SiftGPU.

Table 2 shows a more detailed break-down of the execution time for each of the GPU-based variants. The first two lines indicate the time required to create the entire scale-space and to localize the keypoints. As the table shows, this is significantly faster for SiftGPU. The primary reason is that SiftGPU uses fewer octaves (4 or 5 for our experimental collection, as opposed to 7 for GPU-Eff^2) and scales (3 vs. 7 for GPU-Eff^2), resulting in significantly less processing (at the expense of lower recognition for some image modifications, as shown below). Another reason is that this part is the earliest code of GPU-Eff^2, and SiftGPU is using some well-known optimizations that we have yet to apply.

The third line indicates the time required to gather the keypoints into a list for descriptor extraction. This part is partially implemented using the CPU with SiftGPU and is therefore slower. The final three lines indicate the cost of creating the descriptors themselves. The time required for feature orientation is with 0.7 ms very low as gradient calculation has been performed within the "Build pyramid" step. As mentioned above, the GPU-Eff^2 descriptors only extract a single descriptor per keypoint, and hence the multi-orientation is not needed. Finally, the descriptor creation itself also benefits from our efficient histogram calculation, and is significantly faster for GPU-Eff^2.

Phase	SIFT	Eff^2
Build pyramid	5.4 (14 %)	12.6 (47 %)
Detect keypoints	4.6 (12 %)	7.6 (28 %)
Gather keypoints	11.9 (32 %)	3.7 (14 %)
Feature orientation	5.8 (16 %)	0.7 (3 %)
Multi-orientation	2.5 (7 %)	- (-)
Descriptor creation	7.1 (19 %)	2.3 (8 %)

Table 2: Descriptor creation time breakdown [ms].

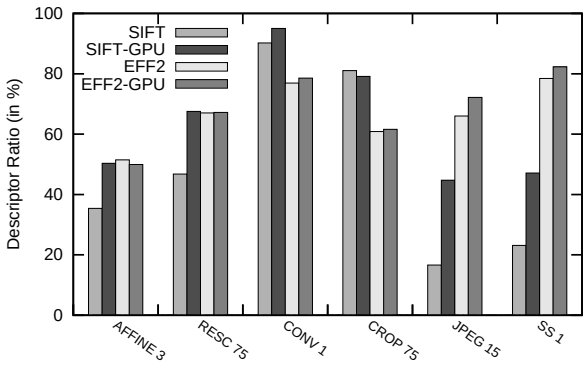

Figure 2: Detection rate for selected modifications.

4.2 Detection Capability

In order to study the detection capability of the descriptors, we loaded each descriptor collection into an NV-tree index [4]. We then used the same 108 original query images and 26 StirMark modifications used in [3] to evaluate the detection capabilities of the four different descriptor variants. As reported in [3], we found that about 41.2% of the SIFT query descriptors find a match from the original image in the top 30 neighbors, while about 57.1% of the Eff^2 query descriptors find a match. Both SiftGPU and GPU-Eff^2, however, perform similarly to Eff^2, finding 57.6% and 58.1% respectively. For individual modifications, however, the tradeoffs are slightly different and Figure 2 presents a representative set of six modifications that we now discuss.

The first two modifications, AFFINE 3 (affine transformation on both axes) and RESC 75 (rescaling to 75%) represent the majority of the modifications (18 in total). For these modifications SiftGPU, Eff^2 and GPU-Eff^2 descriptors yield very similar rates, while the SIFT descriptors yield a lower score. Most of these modifications are easy to detect; the lower ratio for SIFT is due to the abundance of descriptors.

The next two modifications in Figure 2, CONV 1 (low brightness) and CROP 75 (75% central crop), show a better detection rate with the SIFT variants than with the Eff^2 variants. This phenomenon was already described in [3] and is a result of the emphasis of the SIFT variants on low-scale, low-octave descriptors. Note that these were the only two modifications where the SIFT variants performed better.

The last two modifications, JPEG 15 (compression) and SS 1 (conversion to HSV color format), show an advantage for the Eff^2 variants compared to the SIFT variants. The reason is that these modifications remove some of the finer detail in the images, showing the advantage of theEff^2 variants' emphasis on higher-level descriptors. This effect is seen in a total of six modifications.

5 Conclusion

This paper has given an overview on the porting of the Eff^2 descriptors onto a GPU using the CUDA programming model. The experimental evaluation shows a significant speed advantage, not only over CPU implementations, but also beating SiftGPU. We have also shown, however, that GPU-Eff^2 still has room for improvement in scale-space computation and keypoint detection efficiency, which we aim to address in our future work.

References

[1] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan. GpuCV: An opensource GPU-accelerated framework for image processing and computer vision. In *Proceedings of ACM Multimedia*, Vancouver, BC, Canada, 2008.

[2] *NVIDIA CUDA Programming Guide, Ver. 2.0*, 2008.

[3] H. Lejsek, F. H. Ásmundsson, B. T. Jónsson, and L. Amsaleg. Scalability of local image descriptors: a comparative study. In *Proceedings of ACM Multimedia*, Santa Barbara, CA, USA, 2006.

[4] H. Lejsek, F. H. Ásmundsson, B. T. Jónsson, and L. Amsaleg. NV-tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2009.

[5] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[6] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. GPU-based video feature tracking and matching. Technical report, UNC Chapel Hill, 2006.

[7] Ch. Wu. SiftGPU: A GPU implementation of scale invariant feature transform (SIFT). http://www.cs.unc.edu/~ccwu/siftgpu/.

REYKJAVÍK UNIVERSITY

HÁSKÓLINN Í REYKJAVÍK

School of Computer Science
Reykjavík University
Kringlan 1, IS-103 Reykjavík, Iceland
Tel: +354 599 6200
Fax: +354 599 6301
http://www.ru.is