GPU-boosted online image matching

Alexandre Chariot and Renaud Keriven
CERTIS, Ecole des ponts, Paris-Est, France
{*chariot, keriven*}*@certis.enpc.fr*

Abstract

Matching feature points between images is a key point in many Computer Vision tasks. As the number of images increases, this rapidly becomes a bottleneck. We here present how to use the power of GPUs to obtain image matching in typically 20 ms for one thousand points. This speedup makes applications like interactive image matching possible. Such a portable system, dedicated to 3D large scale reconstruction, is reported.

1. Motivation

Matching a given image with many other ones, i.e. the process of detecting common points between an incoming image and a set of images, is a key task in many Computer Vision problems like Structure From Motion[5] or Object Recognition. This is usually done by detecting interest points (*features*) and matching them amongst images. A widespread tool for feature detection and characterization is the *Scale Invariant Feature Transform* (SIFT), proposed by David Lowe[7]. After an invariant Difference of Gaussian based detection of point plus scale pairs, features are described by a high dimension vector, in $\mathbb{R}^{128}$. In most cases, an initial step is to find the k-nearest neighbors of a given descriptor in $\mathbb{R}^{128}$ (usually, $k = 2$). With hundreds of images, each one of them containing thousands of features, matching descriptors rapidly becomes an under-estimated but crucial step with respect to computing time, even with recent CPUs. This clearly is a main bottleneck. The increasing programming flexibility and computational power offered by *Graphics Processing Units* (GPUs), as well as their low cost, provide a now standard way of getting large speed-ups for many algorithms. Point matching is a good candidate for such an implementation. We have developed a GPU-based image matching method that processes a large set of images in a very reasonable time, typically 20ms per image pair for one thousand features. Such a running time allows interactive online applications, such as matching every new image with previous ones during an acquisition process. Our implementation uses the OpenGL library and the Cg shading language but is not restricted to this particular choice (other usual choices are GLSL or Cuda).

2. Previous work

2.1 Feature point based matching

SIFT SIFT[8] provides a local feature detector and descriptor, robust to translation, rotation, scaling and illumination changes. The method is composed of the two following steps:
(i) *Feature detection*: A Gaussian scale space pyramid is built from the input image, yielding the construction of *Differences of Gaussian* (DoG) images. Candidate features are localized in this DoG scale space as local extrema. They are then refined and are assigned an orientation which is the highest peak in the histogram of local gradients orientations.
(ii) *Feature description*: Features are afterward described by a vector in $\mathbb{R}^D$, with $D = 128$, representing gradient orientation histograms on 4×4 neighborhoods in the closest image in scale.
Some work have been proposed to improve SIFT running time: a GPU-based SIFT implementation[12] and *Speeded Up Robust Features* (SURF)[3].

SIFT on GPU This work by S.N.Sinha et al.[12] accelerates some parts of the SIFT algorithm using the hardware capacities of GPUs. Almost the whole feature detection process is deported on GPU. Gradient histograms and descriptor construction are computed on CPU. For technical reasons, this part would not be efficient on GPU. A 10x speedup is obtained, allowing applications on video-sized images.

SURF SURF is a variation of SIFT, proposed by H.Bay et al.[3]. Relying on the Hessian of the image, detection is speeded up thanks to fast derivatives computations using an integral image[13]. Descriptors are also simplified, consisting of sums of first order derivatives, giving a $\mathbb{R}^{64}$ vector.

SIFT on GPU and SURF accelerate feature extraction for each image. Yet, they do not reduce matching time, which is the main bottleneck.

2.2 Approximate Nearest Neighbors

Given two images I and J, matching consists in finding a feature i in I and a feature j in J, such as their respective descriptors are "close" in $\mathbb{R}^d$ for a certain distance $d(i,j)$. A naive algorithm comparing each pair of features yields a quadratic complexity $O(m^2)$, m being the (mean) number of features per image. This complexity could be reduced to $O(m*log(m))$ at the price of approximation, using *Approximate Nearest Neighbors* (ANN)[1].

Note that D. Lowe uses a *Best-Bin-Search* algorithm, a slight variant of *kd-tree* search, on which ANN is also based. When looking for non-ambiguous matches, the following filter is also recommended: for each feature $i \in I$, the two nearest neighbors j and j' in J are found (j beeing the closest one) and the match (i,j) is kept if the ratio $d(i,j)/d(i,j')$ is lower than a given threshold. We will refer to this process as the *two best matches filter*.

Unfortunately, the improved complexity of ANN is not sufficient for online applications. Our claim here is that this asymptotic complexity is not efficient enough for the typical number of features per image (typically a few thousands). On the contrary, a GPU implementation of the naive quadratic match will prove to be significantly faster than ANN-based matching (up to 30 times). In the sequel, we briefly recall GPU computing principles, before describing our matching implementation. A last section gives computational times and speedups, mentions a practical application and concludes.

3. GPU Matching

3.1 General Purpose computation on GPU

General Purpose Computation on GPU (GPGPU) refers to the use of the parallelism of graphics hardware for non graphical calculations. A large panel of parallel computational schemes are well suited to this kind of treatment. The reader is invited to consult [4] for an overview and [11, 10] for advanced examples.

GPU pipelines Figure 1 summarizes the organization of a recent GPU and its parallel pipelines (typically 128 ones). Each pipeline receives a part of the vertices flow from the CPU application. A first programmable computing unit (*Vertex Unit*) processes these vertices, modifies their positions and/or other attributes like normals, colors, etc. They are then passed to a second programmable unit (*Geometry Unit*) which possibly generates new geometrical primitives. Then, the *rasterizer* produces *fragments* (briefly said, pixels), that are finally processed by the third and last programmable unit (*Fragment Unit*), which computes a color for each fragment and stores the result in a (not necessarily) displayed buffer.

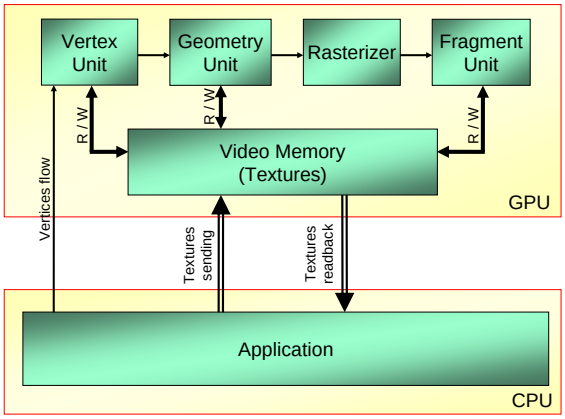

Figure 1. Simplified GPU pipeline

Computational model For our purpose, only Fragment Units are used. Rendering a simple rectangle, the card computes in parallel

$$\forall (u,v),\ T_{\text{out}}(u,v) = \mathcal{F}(T_{\text{in}_1}, T_{\text{in}_2}, ..., T_{\text{in}_n}, u, v)$$

where T_{out} is an output texture, T_{in_k} input textures, (u,v) the pixel coordinates and $\mathcal{F}$ a computing kernel (or *shader*). Textures are assimilated to 2D arrays storing integer or floating point data. $\mathcal{F}$ can read any input texture location (*gather* operations are allowed), but can only write to its own location. For instance, it is possible to calculate $T_{\text{out}}(u,v) = T_{\text{in}}(u+1,v)$, but not $T_{\text{out}}(u+1,v) = T_{\text{in}}(u,v)$. The main limitation of GPGPU programming is this impossibility to perform this *scatter* operation. This model is commonly known as the Concurrent Read Exclusive Write (CREW) model.

3.2 SURF with GPU

Although it is not our main contribution, we have implemented a mixed version of SURF descriptors. As in [12], the image derivatives are computed on the GPU, while descriptors are still constructed on CPU, using the original library[2], yielding some speedup. Our implementation is straightforward. Again, because descriptor computation is only done once for each image in less than a second, we do not consider it as a bottleneck, contrary to image matching that has to be conducted for each pair of images.

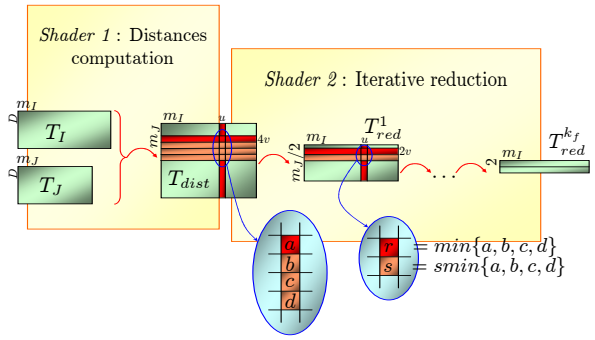

Figure 2. GPU Matching

3.3 Matching

The matching process is done on GPU according to figure 2. Given two images I and J, the principle is the following:
(i) the GPU computes in parallel the distances for each pair of descriptors $(i,j) \in I \times J$.
(ii) targeting to *two best matches filtering* (see section 2.2), the GPU then looks for the two closest descriptors j_i and j'_i for each descriptor i using a so-called *reduction* operation. Note that the symmetric computation is conducted at the same time from J to I, though we will only detail it from I to J.

More precisely, for each image I, let m_I be the number of features detected in I and D the dimension of descriptors ($D = 64$ for SURF). A texture T_I of size $m_I \times D$ is created and filled with the m_I descriptors values, each one occupying a column. Note that for sake of clarity, we will suppose that the features are indexed by integers and denote indifferently by i a feature of I, its descriptor and its index.

Given I and J, a new texture T_{dist} of size $m_I \times m_J$ is created and filled with a first shader computing the Euclidean distances $d(i,j)$. Actually, because the matching should also output a reference to the matched points j_i and j'_i, and not only their distances to i, T_{dist} stores pairs of distance×index. This is easily handled by GPU, for which basic types go from scalars to 4D vectors:

$$T_{dist}(i,j) = (\sum_{k=1}^{D} (T_I(i,k) - T_J(j,k))^2, j)$$

The second step is an iterative texture reduction, done with an unique second shader, in which the height of the final texture will be scaled down to 2, to keep the two nearest neighbors for each i. This pyramid of textures goes from $T^0_{red} = T_{dist}$ to $T^{k_f}_{red}$ where k_f depends on m_J: $k_f = \log_2 m_J - 1$. The k^{th} iteration producing a texture T^k_{red} from T^{k-1}_{red} uses the following shader:

$$\begin{aligned} T^k_{red}(u,2v) = \\ min\{&T^{k-1}_{red}(u,4v), T^{k-1}_{red}(u,4v+1) \\ &T^{k-1}_{red}(u,4v+2), T^{k-1}_{red}(u,4v+3)\} \\ T^k_{red}(u,2v+1) = \\ smin\{&T^{k-1}_{red}(u,4v), T^{k-1}_{red}(u,4v+1) \\ &T^{k-1}_{red}(u,4v+2), T^{k-1}_{red}(u,4v+3)\} \end{aligned}$$

where $min\{\cdot\}$ (respectively $smin\{\cdot\}$) returns the first (respectively the second) minimum of 4 pairs, according to the order on the first element of the pairs. Finally, $T^{k_f}_{red}$ is sent back to the CPU, giving, for each feature i of I, the two nearest neighbors and the related distances: $(d(i,j_i), j_i) = T^{k_f}_{red}(i,1)$ and $(d(i,j'_i), j'_i) = T^{k_f}_{red}(i,2)$.

Note that, as a final optimization not detailed here, we actually use textures of 4D vectors instead of 2D vectors. Doing so, we merge $T^k_{red}(u,2v)$ and $T^k_{red}(u,2v+1)$ at the same location (u,v) of a smaller texture and, more importantly, compute min and $smin$ at once.

4. Results and discussion

All GPU tests were run on a $3GHz$ Xeon CPU equipped with an nVidia GeForce 8800 Ultra card enclosing 128 stream processors and 768 Mo of internal memory. Table 1 shows matching times in ms per image pair. Neither features extraction nor ANN tree construction are taken into account, since they are done once for every image. Our GPU matching method is compared to ANNs and naive $O(m^2)$ CPU matchings, in function of the average number m of features in each image. As expected, both times for CPU naive matching and our method grow approximately quadrat-

m	512	1024	2048	4096
CPU	160	660	4230	24220
ANN	73	290	1200	7990
GPU	6.8	19	71	270
CPU/ANN	2.2	2.3	3.5	3.0
CPU/GPU	23.5	34.6	59.6	89.7
ANN/GPU	10.7	15.3	16.9	29.6

Table 1. Times in ms per image pair and Speedup, w.r.t the number m of features. CPU=naive $O(m^2)$ algorithm, ANN=Approx. Nearest Neighbors, GPU=our method

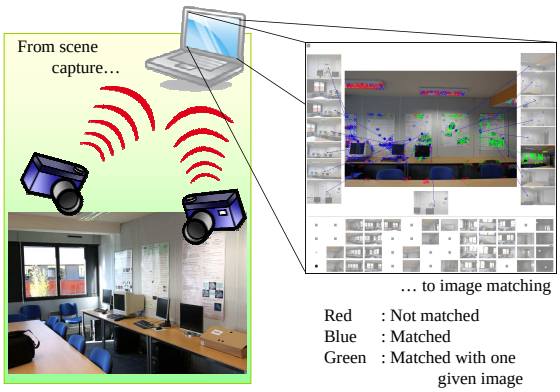

Figure 3. Our portable setup for interactive image acquisition (see text)

ically with m. Yet cache effects, fast GPU memory and CPU/GPU memory transfers make the times for the CPU method to grow faster than expected while they grow slower than expected for our method. This explains the speedup going from 23 to 90. Our ANN implementation uses the ANNLib open library[9]. Although more efficient than the naive CPU approach, this yield running times that do not follow the predicted asymptotic complexity. This might easily be explained by the small number of points w.r.t. the high space dimension ($d = 64$). As a result, we observe a 11 to 30 speedup between our method and ANNs. Boosted by these new possibilities, we developed a first application dedicated to help users to take photos of a large scale scenes for 3D reconstruction purposes[6]. In our portable setup (figure 3), several digital reflex cameras communicate via WiFi connections with a laptop equipped with an nVidia GeForce 8800M GPU. By incrementally matching images and displaying matched regions, the system guides users to places that have not yet been taken enough. Image pair matching runs in typically 25 ms, providing acceptable waiting delays between shots.

Conclusion This paper has presented a GPU-boosted system that makes possible online image matching applications, even in portable setups. We plan to make our matching application available on the WEB. This will be the first step of a large scale interactive 3D reconstruction system.

References

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[2] H. Bay, T. Tuytelaars, and L. V. Gool. SURF library, 2006. `http://www.vision.ee.ethz.ch/~surf/`.

[3] H. Bay, T. Tuytelaars, and L. V. Gool. SURF: Speeded up robust features. In *ECCV 2006 Proceedings*, pages 404–417, 2006.

[4] GPGPU.org. General purpose computation on GPUs. Website, 2004. `http://gpgpu.org/`.

[5] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition, 2004.

[6] P. Labatut, J.-P. Pons, and R. Keriven. Efficient multi-view reconstruction of large-scale scenes using interest points, delaunay triangulation and graph cuts. In *ICCV*, Rio de Janeiro, Oct 2007.

[7] D. G. Lowe. Object recognition from local scale-invariant features. In *ICCV*, pages 1150–1157, 1999.

[8] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.

[9] D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. Website, 2006. `http://www.cs.umd.edu/~mount/ANN/`.

[10] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007.

[11] M. Phar and R. Fernando. *GPU Gems 2*. Addison-Wesley Professionnal, 2005.

[12] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. Gpu-based video feature tracking and matching. *Workshop on Edge Computing Using New Commodity Architectures*, 2006.

[13] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR 2001 Proceedings*, volume 1, pages I–511–I–518 vol.1, 2001.