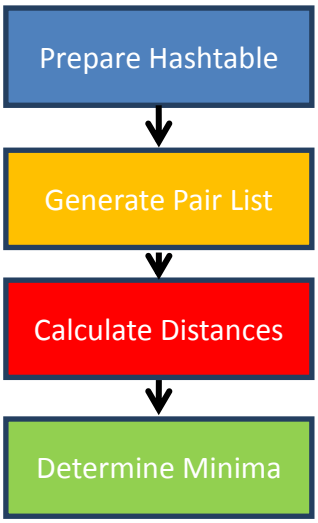
Juliane Hüttl, Moritz Vossenberg, Michael Winkelmann

What is the purpose and functionality of the algorithm?

The tree-based hashing feature matching algorithm (TBH) is an improved version of the SIFT Feature Matching Algorithm. Itconstructs a hash table by simulating a kd-tree in order to reduce complexity. The Euclidean Distance determines the similarity between two feature points in each hash bucket (the best match has the lowest Euclidean Distance).

Our solution:

The evalBucketList consumes 61% of the overall computation time. Hence, we focused on optimizing this part of the algorithm. We considered two possible optimizations: To process one hash bucket per thread or one pair of insert and found values per thread.



We decided to implement the latter variant because the number of insert and found values highly varies in different buckets. Hence, using a thread for each bucket would result in high thread divergence. Moreover, for small hash tables only a small amount of threads would be needed, meaning we could not exploit the full computing power of the GPU. By computing all possible pairs, one pair per thread can be processed and the kernel execution for each thread is more consistent. The disadvantage is that a larger amount of memory is required as the pair list size grows quadratically with the size of the feature point lists.

First of all, we optimized the generation of the hash table on the CPU. The fillHashTable is subdivided into a preparing step in which the number of insert and found values for each bucket is determined and the required memory for the pair list is allocated. In the second step, the pair list is filled with the insert and found pairs. Afterwards, the pair list as well as the feature point lists are copied to the GPU memory. As access to the data of the feature point lists is read-only, as well as non-coalesced, we use the texture memory space for storage. On the GPU, the compDistKernel is executed.

Afterwards, the minimum distances need to be determined. We implemented two versions of TBH_CUDA: One performs the computation of minima on the CPU and the other on the GPU.

Unfortunately, the GPU does not work as fast as expected. This is caused by frequent random memory accesses (caching of data not possible) which are very slow on GPU. Since we have no prior knowledge about the position of each feature point or the number of times the feature points occur in the pairs array, it is not possible to use shared memory to speed-up the memory accesses. Moreover, the integer performance of GPUs is worse than on CPU.

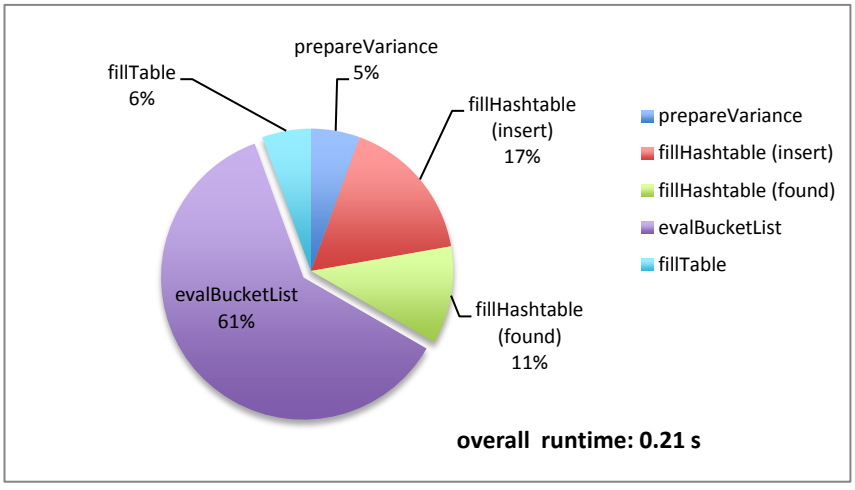


Figure 1: Relative run times of the sub-algorithms of TBH.

Results

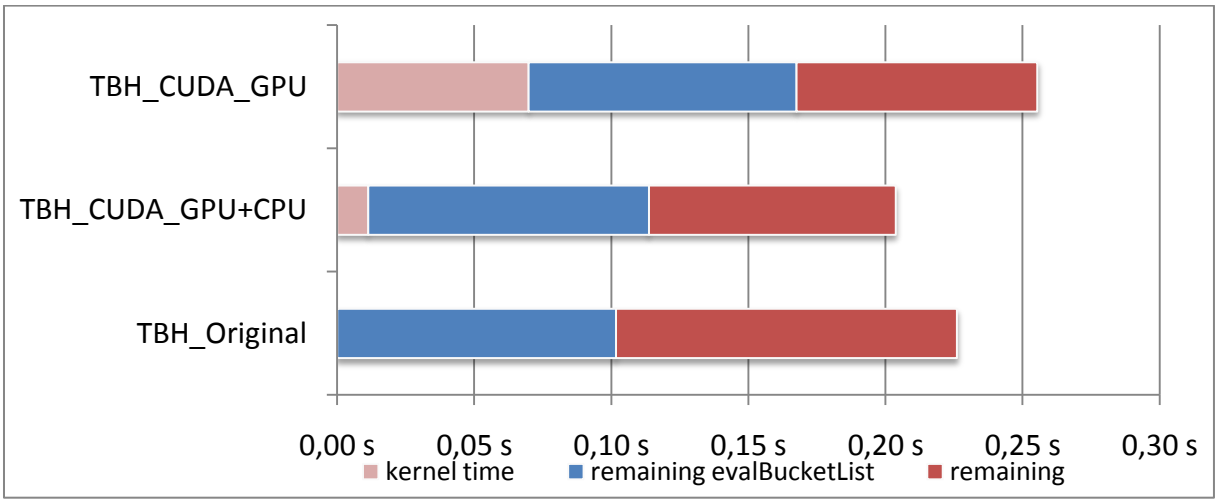


Figure 2: Time measurement for 2 feature descriptor lists
with 6101 and 5960 descriptors, 2743 matches found.

Due to the disadvantages mentioned above our version computing the distances and minimum distances on the GPU performs worse than the original TBH version (with an overall runtime of about 0.25 seconds). Computing only the distances on the GPU and the minima on the CPU instead, reduces the overall kernel runtime to about 0.011 seconds. Even though the run time of evalBucketList of our version is longer than the original evalBucketList, we managed a performance gain of about 0.3 seconds. This is the result of our optimization of the functions executed prior to evalBucketList (code executed on CPU only).

Further improvements:

All steps are processed in sequential order. This leads to a performance penalty because not all steps can exploit the full GPU power and their intermediary results like the lists of the pairs and distances are very large. It also causes a lot of computation overhead as these large data sets have to be copied into device memory.

A pipelined execution of the steps could handle this problem more efficiently. In this approach, different steps will be assigned to different GPU multiprocessors. We have three different steps which use three groups of microprocessors (groups A,B and C). For example, two GPU multiprocessors (A) handle the pair list generation, 12 multiprocessors could be responsible for distance calculation (B) and another two for determining the minimum distances (C).

Group A computes a small subset of the pair list which fits into the device memory. If a new subset of the pair list is available, this subset is processed by Group B and the resulting distances are processed by Group C. While the distances are calculated and the minima are determined, new subsets of the pairs are generated at the same time. Hence, after a short phase of initialization (where the generation of the pair list subset has not been finished yet), all GPU multiprocessor are kept busy. The whole process chain is repeated until the full pair list was processed.

The disadvantage of this approach is the difficulty of implementation. Moreover, it is unclear how much computation overhead is required to realize this approach.