Exploring HW/SW Co-Optimizations for Accelerating Large-scale Texture Identification on Distributed GPUs

Junsong Wang
junsong.wang@easy-visible.com
V-Origin Technology
Beijing, China

Xiaofan Zhang
xiaofan3@illinois.edu
University of Illinois at Urbana-Champaign
Urbana, IL, USA

Yubo Li
yubo.li@easy-visible.com
V-Origin Technology
Beijing, China

Yonghua Lin
yonghua.lin@easy-visible.com
V-Origin Technology
Beijing, China

ABSTRACT

Texture identification has been developed recently to support one-to-one verification and one-to-many search, which provides much broader support than texture classification in real-life applications. It has demonstrated great potentials to enable product traceability by identifying the unique texture information on the surface of the targeted objects. However, existing hardware acceleration schemes are not enough to support a large-scale texture identification, especially for the search task, where the number of texture images being searched can reach millions, creating enormous compute and memory demands and making real-time texture identification infeasible. To address these problems, we propose a comprehensive toolset with jointly optimization strategies from both hardware and software to deliver optimized GPU acceleration and leverage large-scale texture identification with real-time responses. Novel technologies include: 1) a highly-optimized cuBLAS implementation for efficiently running 2-nearest neighbors algorithm; 2) a hybrid cache design to incorporate host memory for streaming data toward GPUs, which delivers a 5× larger memory capacity while running the targeted workloads; 3) a batch process to fully exploit the data reuse opportunities by considering available compute resources and memory bandwidth constraints. 4) an asymmetric local feature extraction to reduce the memory footprint for keeping feature matrices of reference texture images. To the best of our knowledge, this work is the first implementation to provide real-time large-scale texture identification on GPUs. By exploring the co-optimizations from both hardware and software, we can deliver 31× faster search and 20× larger feature cache capacity compared to a conventional CUDA implementation. We also demonstrate our proposed designs by proposing a distributed texture identification system with 14 Nvidia Tesla P100 GPUs which can complete 872,984 texture similarity comparisons in just one second.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP '21, August 9–12, 2021, Lemont, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3473520

KEYWORDS

texture identification, GPU acceleration, cuBLAS, batching, hybrid cache, nearest neighbor, feature extraction, SIFT

ACM Reference Format:
Junsong Wang, Xiaofan Zhang, Yubo Li, and Yonghua Lin. 2021. Exploring HW/SW Co-Optimizations for Accelerating Large-scale Texture Identification on Distributed GPUs. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3473520

1 INTRODUCTION

Texture analysis is one of the most critical tasks in computer vision and has been widely used in a broad range of applications, such as texture classification based remote sensing [4], materials recognition [29], and segmentation based biomedical imaging [13]. There are rich studies on deploying texture analysis on the classification tasks, which intend to classify textures based on their different surfaces. In most of the texture classification pipelines, the hand-engineered features are firstly extracted using off-the-shelf algorithms, such as SIFT [17], SURF [2], and a pre-trained CNN extractor [11]. After that, a dictionary can be learned and the feature distributions are embedded with bag-of-visual-words (BoW) [26] or other more advanced encoders, such as VLAD [11] and Fisher Vector [21]. Finally, a classifier, such as SVM, is trained to deliver classification results. Researchers in [29] also proposed a novel encoding layer to integrate the entire dictionary learning and encoding pipeline into one single module. Large datasets are also proposed to help benchmark the performance of various emerging texture classification algorithms, such as MINC-2500 [3] and Flickr material [23] dataset.

More recently, authors in [27] introduced a new application direction of deploying texture analysis and identified it as a texture identification task. This task requires one-to-one verification and one-to-many search, which is very similar to the facial recognition task [7, 24]. Unlike most of the tasks related to texture analysis, the identification task aims to learn the similarity between any two texture images instead of classifying the input images as all the texture images belong to the same category. In [27], texture identification was successfully applied to enable the traceability of agricultural products.

To efficiently deliver texture identification, authors in [27] have investigated several off-the-shelf approaches, including BoW based

image retrieval, image matching, and deep metric learning. Among them, the image matching approach is likely to achieve the most decent performance in both one-to-one verification and one-to-many search tasks. Local features are extracted both from the reference and query images using SIFT algorithm. The 2-nearest neighbors algorithm that calculates each query feature's nearest and the second-nearest neighbors is applied to find the distinct matching points. However, the 2-nearest neighbors algorithm exhibits extremely high compute demands with the computational complexity as $O(dN^2)$, where d and N are the dimension of feature vector and the number of feature extracted for each texture image, respectively. Initial experiments show that the throughput performance of texture image search using a high-end GPU (Nvidia Tesla P100) is about 2,012 images/s by using a native OpenCV CUDA implementation. By considering the amount of millions of images, it costs more than 8 minutes to perform one texture image search process, which can not be tolerated in practical cases.

One of the major problems resulting in unsatisfactory performance is the low utilization of available compute resources in the targeted GPU. After examination, we found that the existing implementation (developed by a native OpenCV CUDA solution) only exploit 4.4% of the GPU's computational potential. It means the image matching performance could be significantly improved once the GPU resources are fully utilized. In this paper, we propose a comprehensive toolset of hardware-software co-optimization strategies to deliver a hardware-efficient texture matching algorithm (SIFT + 2-nearest neighbors) and an improved GPU-based hardware acceleration design to facilitate large-scale texture identification with real-time responses. We will use two critical performance metrics to evaluate our proposed optimization strategies which include **capacity** to measure how many feature matrices of the reference texture images can be cached in the search system's memory; and **speed** to indicate how many texture image similarity comparisons can be completed in one second. The performance improvement after applying the proposed optimization strategies is illustrated in Fig. 1, where 20× larger capacity and 31× faster speed are achieved.

To summarize, our main contributions are as follows.

(1) **A highly optimized cuBLAS implementation dedicated to the 2-nearest neighbors algorithm.** The original CUDA based KNN implementation is optimized by our proposed cuBLAS implementation using half-precision floating-point format (PF16), which successfully increases the speed and reduces the memory footprint while applying to the 2-nearest neighbors.

(2) **A batch process for reference feature matrices.** It helps increase the data reuse opportunities during matrix multiplication and better leverage the RootSIFT algorithm we adopt for feature extraction.

(3) **A hybrid cache design** to leverage both GPU and host memory space to significantly enlarge the memory capacity for keeping reference feature matrices. Multiple CUDA streams is utilized to overlap the computation and transmission time, which makes the search speed degradation caused by the hybrid cache minimized.

(4) **An asymmetric local feature extraction** is adopted for reducing memory footprint. By applying this method, we capture fewer features from the reference texture image while more features for the query texture image instead of keeping these two parts equally. It saves the memory space for a larger-scale design with improved search speed without sacrificing obvious accuracy.

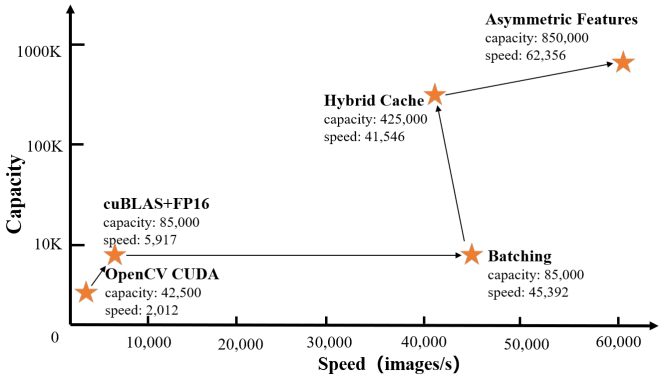

Figure 1: The performance of running large-scale texture identification after adopting four optimization strategies (corresponding to four major contributions) proposed in this paper. The performance is measured by a system setup with single Nvidia Tesla P100 GPU with 16GB memory and an extra 64GB host memory space.

(5) **A distributed texture identification system** to demonstrate our proposed designs with 14 Tesla P100 GPUs. By applying the proposed HW/SW co-optimization strategies, the capacity of this system can be boosted to ten million level and its search speed can reach million images per second. We also provide RESTful APIs to ensure easy access to the system.

The rest of the paper is organized as follows. Sec.2 introduces the related work while Sec.3 summarizes the texture matching procedure and challenges we encountered. From Sec.4 to 7, we describe the detailed optimization strategies to improve the capacity and search speed. In Sec.8, we describe a ready-to-deploy search system with the proposed optimization strategies and in Sec.9, we conclude this paper.

2 RELATED WORK

Texture analysis is import for a variety of applications, such as materials classification [20], remote sensing [4], biomedical imaging [13] and industrial automation [14]. Among these texture analysis based tasks, feature extraction is one of the most fundamental technologies to capture visual content of the targeted images. In the past few decades, various of texture extraction methods were proposed for different texture applications, such as the grey level co-occurrence matrix (GLCM), filter bank transformer [5], and SIFT [17]. In recent years, the convolutional neural network (CNN) was proposed and started dominating most of the vision tasks with state-of-the-art accuracy. Authors in [8] provided a comprehensive comparison of features extracted from different layers of CNN to standard SIFT descriptors. In [28], an end-to-end framework is also proposed to detect key-points, estimate orientation, and compute descriptors, and an attention mechanism is used in [20] to improve the accuracy of key-point detection.

Texture identification is first proposed in [27] to identify product counterfeiting and facilitate reliable product traceability. By using the classical image matching pipeline [18], both one-to-one verification and one-to-many search tasks can achieve excellent accuracy in the Pu'er tea brick dataset. However, the intensive computation coming from the nearest neighbor algorithm would be problematic, which slows down the search speed when the amount of image reaches million-level in real-life applications even with GPU acceleration. To accelerate the nearest neighbor algorithm, approximate nearest neighbor is proposed to overcome the curse of dimensionality, such as KD-trees [25] and hierarchical k-means [19]. These improved methods have been successfully applied in content-based image retrieval (CBIR) [15, 20]. Researchers also investigate the data compression for improved execution speed on GPUs. For example, the SIFT feature vectors are compressed in [15] by using local sensitive hashing (LSH) [6] along with a GPU implementation to accelerate the query process. Product quantization [10] is also proposed to decompose the high dimensional feature space into Cartesian product of low dimensional sub-spaces. Researchers then accelerate the nearest neighbor search by separately quantizing the sub-spaces. In industry, a billion-scale similarity search engine (Faiss) that can be accelerated by GPUs is released by Facebook [12]. It also adopts product quantization and can deliver state-of-the-art performance. This engine can be applied to different similarity search tasks, such as brute-force, approximate and compressed-domain search.

In this paper, we focus on texture identification, which looks similar to another popular task called content-based image retrieval (CBIR), but they are fundamentally different. The goal of CBIR is to retrieve images containing the targeted content from a large-scale dataset given the query content. Unlike CBIR, texture identification aims at finding the exactly same texture requested by the the query from a large-scale reference dataset, where at most one image that can be matched. Usually, the dataset is fine-grained from a single category. The accuracy requirement in texture identification, especially for product traceability, is also much higher than CBIR. Therefore, in CBIR, the features extracted from the reference dataset images are all combined together and then compressed or clustered into a single feature database. When performing the retrieval, only single nearest neighbor across all the features is conducted. However, in texture identification as discussed in [27], given a query texture image, the reference images in the dataset are matched separately (in one-by-one manner). The dimension in each nearest neighbor during the image matching is small, but it will be repeated M times, where M is the number of texture images in the reference data and can reach to million scale. These completely different computation pattern invalidates the previous proposed compression and GPU acceleration approaches when targeting texture identification tasks.

3 TEXTURE IDENTIFICATION METHOD AND CHALLENGES

There are three major approaches that deliver texture image matching by using CBIR, deep metric learning, and image matching. By considering the CBIR related approaches, they can be very efficient but suffer low accuracy. It means these approaches are probably lacking the discriminate capability especially in fine-grained identification tasks. Regarding the methods using deep metric learning, they have proven great advantages in facial recognition, but still can not perform well in our targeted task as the texture image dataset is very different from the facial image dataset. The unaligned texture images and insufficient positive pairs significantly damage the achievable accuracy when using deep metric learning. Contrary to the first two approaches, image matching based methods show better potential to leverage texture matching with a decent accuracy even without using any pre-training processes as demonstrated in [27].

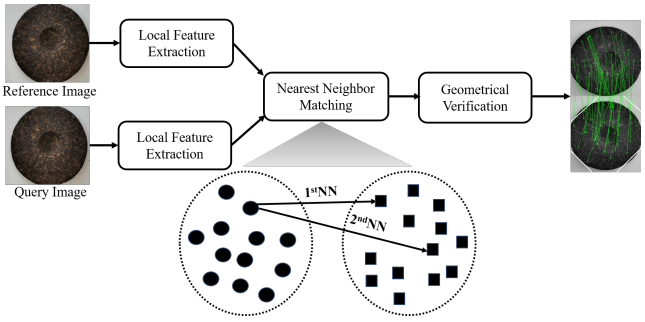

Figure 2: Overview of the image matching algorithm.

3.1 Our Texture Identification Method

The image matching approach [18] can identify the similar content between two images following a pipeline of local feature extraction, feature matching, and geometric verification as illustrated in Fig. 2. The local feature extraction can adopt one of the following algorithms, such as SIFT [16, 17], SURF [2], and ORB [22]. For the matching process, it generally takes the nearest neighbor algorithm, which first calculates the distance between each reference-query and then checks the distance ratio (called ratio test) between the nearest and the second-nearest neighbor. We call this kind of nearest neighbor approach as 2-nearest neighbors. If the distance ratio is below than a predefined threshold, this pair is considered as a good match. Finally, the geometric verification is performed to remove the outliers that may be mismatched. The number of matched keypoint is also considered to determine whether two texture images contain the same texture. Only when the number higher than a pre-defined threshold can these two images be considered with the same texture.

In this paper, leveraging the analysis in [27], we use SIFT feature extraction and 2-nearest neighbors for large-scale texture identification to achieve better accuracy.

3.2 Dataset

We select the tea-brick texture dataset [27] to evaluate the texture identification accuracy. Each tea-brick is compressed with numerous tea leaves and has sufficient texture information. All the tea-brick images in this dataset are collected from the manufactures and customers using industry cameras and smartphone cameras, respectively. This dataset has well considered the diverse image

capturing conditions, such as viewpoints, occlusions, and illuminations. The reference dataset contains 300,000 tea-brick images, and the query dataset contains 354 tea-brick images. Top-1 accuracy is adopted for evaluating the identification algorithm.

3.3 Challenges

Challenges of efficiently running SIFT feature extraction and 2-nearest neighbors may come from different places when targeting different tasks. By considering the verification task, the feature extraction step dominates the compute demands as it is the most compute-intensive part in the pipeline. However, in this paper, we focus on the identification task of searching in a large reference texture image dataset. The 2-nearest neighbors matching becomes the most complicated step and asks for most of the compute resources since the features of the reference texture images can be calculated offline. When delving into the 2-nearest neighbors matching, we can see it requires two steps. The first step calculates the similarity matrix that contains the distances between any two local features from the two texture images. It requires N^2 feature vector Euclidean distance calculation where N is the number of features extracted for each texture image. Assuming 768 features are extracted in each texture image and the descriptor of each image is a 128-dimension vector (SIFT feature), each matching requires 75 million multiply-add operations. If we search in a million texture images, we need to handle 75 trillion operations, which is extremely enormous. The second step needs to find the top-2 nearest neighbors, which can be considered as memory bandwidth intensive the data movement speed limits its performance.

Although the OpenCV library already integrates a CUDA implementation for the KNN matching, in the typical setting of 768 SIFT features, its performance is relatively low with only 2,012 and 2,937 images/s when running on a Tesla P100 and a V100 GPU, respectively. By considering a large-scale texture search with millions of images, it is far from enough to provide real-time responses. The major reasons are, 1) this library is designed for universal KNN algorithm acceleration but not specifically for image matching where k is usually set to 2; 2) this library uses the native CUDA implementation instead of the newer cuBLAS implementation with better optimizations provided by Nvidia, such as half precision data format and tensor core acceleration support in modern GPUs.

4 CUBLAS FOR 2-NEAREST NEIGHBORS

4.1 cuBLAS Implementation

cuBLAS is a celebrated and highly optimized linear algebra library for vector and matrix operation based on Nvidia's GPU. In literature [9], the authors reformulate the similarity matrix calculation to a matrix multiplication implementation as illustrated in Eq.1

$$\rho^2(R,Q) = N_R + N_Q - 2R^TQ \tag{1}$$

R and Q are two matrices of dimension $d \times m$ and $d \times n$, which contain the local features of m reference keypoints and n query keypoints from reference and query texture images, respectively. Each extracted feature of the keypoint is a d dimension vector. Take the SIFT feature as an example, d is 128, while d is 64 for SURF features. In the following, we call them reference feature matrix and query feature matrix. In this paper, without specific mentioned, we adopt the SIFT feature. The elements of the i^{th} row of N_R are equal to the corresponding feature's squared L_2-norm $\|r_i\|^2$. The elements of the j^{th} column of N_Q are equal to the corresponding feature's squared L_2-norm $\|q_j\|^2$. To save the GPU memory, the N_R and N_Q are stored with two vectors of dimension m and n, respectively. Details of the cuBLAS based implementation for KNN is illustrated in Algorithm 1 as discussed in [9]. Step 6 and 7 can be merged into a single kernel to save data movement.

Algorithm 1 KNN cuBLAS implementation

1: Compute the Vector N_R using CUDA;
2: Compute the Vector N_Q using CUDA;
3: Compute the $m \times n$ matrix $A = -2R^TQ$ using cuBLAS's GEMM;
4: Add the i^{th} element of N_R to every element of the i^{th} row of the matrix A using CUDA; // in-place, no extra GPU memory
5: Sort each column of A in parallel using CUDA; // in-place, no extra GPU memory;
6: Add the j^{th} element of N_Q to the first k elements of the j^{th} column of A using CUDA; // in-place, no extra GPU memory
7: compute the square root of the first k elements using CUDA; // in-place, no extra GPU memory
8: Extract the uppermost $k \times n$ matrix of A, which is the distance matrix for the k-nearest neighbors of each query feature. Move this sub-matrix and its corresponding keypoint index to CPU host memory for further process.

In our search design, all the reference feature metrics (R) are calculated offline and stored in GPU memory as well as their squared L_2-norm vectors (N_R). The local features of the query image Q is calculated first in CPU and move to GPU memory. The N_Q vector is calculated with GPU. The search is performed by calculating the similarity of query image and every reference image one by one according to Algorithm 1 with the performance illustrated in Table 1. In this experiment, we use the same settings as [27], where 768 SIFT features are extracted for each image ($m = n = 768$, $d = 128$). The post-processing in this experiment only includes ratio test and edge feature removing, and no geometrical verification is conducted.

The cuBLAS implementation has significant speedup as discussed in [9] when the feature matrix is large. However, in our image matching scenario, the feature matrix is relatively small. Although the computation dominated part is well accelerated by general matrix to matrix multiplication (GEMM), the performance gain is still only 1.5× compared to OpenCV's CUDA implementation as illustrated in Table 1. By profiling each step of the cuBLAS implementation, we found the sorting step is the bottleneck, which occupies 67% of the computation time. The reason is that the modified insertion sorting is used, which involves too many data load and store from/to the GPU memory. Actually, sorting is a bandwidth intensive task, and we should minimize the data movement to break the memory wall. Fortunately, in our nearest neighbor, k is always equal to 2. We can simplify the sorting process by keeping two registers in each GPU thread to store the top-2 elements. Therefore, we can find the top-2 elements by scanning all the elements of the distance vector only once. No data storing to GPU memory occurs,

Table 1: cuBLAS implementation performance, $m = n = 768$, $d = 128$, measured in Nvidia Tesla P100/16GB GPU. The GPU memory usage is evaluated with storing 10,000 reference feature matrices and their corresponding N_R.

Execution step	CUDA (OpenCV)	cuBLAS[9]	cuBLAS (ours)	cuBLAS+FP16 (ours)
GEMM/setp3(us)	-	35.22	35.22	24.92
Add N_R/step4(us)	-	8.94	8.94	8.98
Top-2 sort/step5(us)	-	221.5	40.20	68.32
Add N_Q and Sqrt/step6 &7 (us)	-	4.71	4.71	4.87
Device to Host memory copy/step8(us)	-	47.32	47.32	44.73
Post-processing/CPU(us)	-	12.60	12.60	17.18
Total time(us)	497.0	330.3	148.5	169.0
Speed (images/s)	2012	3027	6734	5917
GPU Memory usage (MB)	4271	4307	4307	2307

and the data movement is minimized. By applying this optimization, the sorting time is significantly reduced by 81.9%, and the speed is further improved by 2.2×.

4.2 FP16 Implementation

FP16 is supported in modern Nvidia's GPUs, such as Tesla P100, V100, and A100 card. We can use the FP16 instead of the full precision floating-point format to represent feature matrices, which can save 50% memory usage and has 2× faster in theory. Besides, FP16 can also explore the capability of modern tensor core architecture, which is introduced after Volta GPU. Since FP16 has much smaller numerical range, a scale factor is usually applied to avoid overflow before converting from full precision to FP16. As illustrated in Eq. 2, we use compression error to evaluate the numerical influence of using FP16 precision when calculating the similarity matrix.

$$comp\_error = \frac{1}{m \times n} \sum_{i=1}^{m} \sum_{j=1}^{n} \frac{|\rho\left(R_i, Q_j\right) - \rho_{FP16}\left(R_i, Q_j\right)|}{\rho\left(R_i, Q_j\right)} \quad (2)$$

$\rho\left(R_i, Q_j\right)$ is the distance with full precision between the i^{th} column of R and j^{th} column of Q, and $\rho_{FP16}\left(R_i, Q_j\right)$ is the corresponding distance when using FP16. In our experiment, we randomly select 1,000 reference and query image pairs from the tea-brick dataset for compression error evaluation. The averaged compression error and search accuracy with FP16 are both illustrated in Table 2. A large scale factor may cause the similarity matrix computation overflow, while a small scale factor will increase the compression error and further affect the accuracy. From Table 2, we demonstrate that the search accuracy can be very robust when the scale factor is from 2^{-2} to 2^{-12}. In real practice, the scale factor is set to 2^{-7}.

The performance of FP16 implementation is also shown in Table 1. As we expected, the GPU memory usage and GEMM computation time are reduced by 46.4% and 29.2%, respectively. The memory copy time (device to host memory) is also slightly reduced by 5.5% because the size of the resulting distance matrix is reduced by 50%. The post-processing time is increased by 36.3% since extra conversion from full precision to FP16 is needed. However, the top-2 sorting time is unexpectedly increased by 70%, which decreases the final speed by 12.1%. The possible reason is that an extra half precision intrinsic function is called for each comparison operation. We will address this issue by enabling multiple CUDA streams which will be introduced in Section 6.2.

5 BATCHING REFERENCE FEATURE MATRIX

In this section, we will introduce RootSIFT to simplify the nearest neighbor algorithm and use batching technology to significantly improve the search speed and GPU efficiency, which has already been widely used in deep neural network training and inference.

5.1 RootSIFT

By deeply investigating the nearest neighbor algorithm as illustrated in Algorithm 1, we notice that the whole pipeline will be further simplified if the SIFT features are L_2 normalized. However, if we directly normalize the SIFT features, the search accuracy is largely affected. In this paper, we introduce RootSIFT [1], which normalizes each of the SIFT feature vector with L_1-norm, followed by the element-wise square root. The Euclidean distance between two RootSIFT features is equivalent to using the Hellinger kernel to compare the original two SIFT features. The Hellinger kernel for two L_1-normalized histograms x and y is defined as $H(x, y) = \sum_{i=1}^{n} \sqrt{x_i y_i}$, which is good at quantifying the similarity between two probability distributions. Actually, the SIFT descriptor is a histogram and is naturally a probability distribution. With this approach, we can normalize the SIFT vectors and the search accuracy loss is very limited (with only 0.84%).

This approach is illustrated in Algorithm 2, which helps compress the original solution (Algorithm 1) to 4 steps, and neither N_R and N_Q are required to be calculated. In Algorithm 2, step 3 and 4 can be further merged since the $2 + A$ and square root can be calculated in-place directly after the top-2 sorting of each column of A. It can further minimize the data movement as an in-register computation without any extra data load and restore. This simplification also makes the batching process more efficient.

5.2 Batching

Even with cuBLAS implementation, the actual computation is only 0.87 TFLOS (Nvidia Tesla P100, FP16 precision), which is far from the GPU's 18.7 TFLOS peak capacity. The major reason is that the

Table 2: Compression error and search accuracy with difference scale factors when using FP16 precision. For search accuracy, we set $m = n = 768$, $d = 128$.

precision	full precision	FP16						
scale factor	-	1	2^{-1}	2^{-2}	2^{-7}	2^{-12}	2^{-14}	2^{-16}
averaged compression error	-	overflow	overflow	0.1026%	0.1026%	0.1026%	0.1043%	0.3492%
accuracy	98.58%	-	-	98.58%	98.58%	98.58%	98.31%	98.31%

Algorithm 2 Simplified cuBLAS implementation of 2-nearest neighbors with RootSIFT

1: Compute the $m \times n$ matrix $A = -2R^T Q$ using cuBLAS's GEMM;
2: Sort (find top-2 smallest elements) each column of A in parallel using CUDA; // in-place, no extra GPU memory
3: compute the square root of the first two elements of $2 + A$; // in-place, no extra GPU memory
4: Extract the uppermost $2 \times n$ matrix of A, which is the distance matrix for the 2-nearest neighbors of each query feature. Move this sub-matrix and its corresponding keypoint index to CPU host memory for further process.

two feature matrices from the reference and query texture images are too small, and there are not enough data reuse opportunities that can be exploited. To alleviate this problem, we should increase the arithmetic intensity to exploit the full capacity of GPU. Thus, we can batch the reference feature matrix as illustrated in Fig. 3 to significantly increase the size of the matrix to achieve higher data reuse, which results in more efficient GEMM calculation.

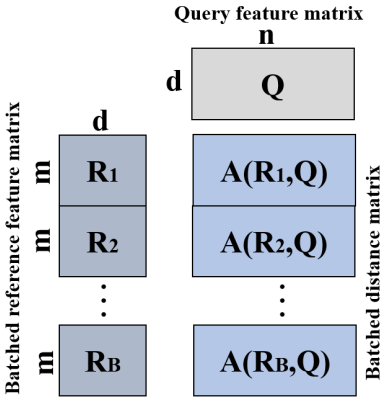

Figure 3: Matrix multiplication illustration of batched reference feature matrix.

Fig. 4 shows the batching performance using Nvidia Tesla P100 and V100 GPU. In this experiment, we assume all the reference and query features are pre-stored in GPU memory. From Fig. 4, we can see that the speed is significantly improved with the increase of batch size, and the performance gain becomes flat when the batch size is larger than 256. For the P100 GPU, the speed is improved from 5,753 images/s to 45,539 images/s with a significant speedup of 7.9× when the batch size is increased from 1 to 1024. Similar speedup of 7.5× is observed when using a V100 GPU. The major reason is that the data reuse is largely increased and the CUDA cores are efficiently exploited in GEMM computation and top-2 sorting. If the tensor core is enabled, we can achieve the peak performance at 86,519 images/s which is an additional 1.3× speedup when batch size is 1024. However, only 1.15× speedup is observed when the batch size is 1, which indicates that the tensor core can also benefit from large matrix with sufficient data reuse.

Table 3: Performance of batched reference feature matrix based on Algorithm 2 with half precision ($m = n = 768$, $d = 128$) measured in Nvidia Tesla P100/16GB GPU. For BatchSize=1024, the time is normalized by 1024 for more intuitive comparison.

Execution step	BatchSize=1	BatchSize=1024
HGEMM/step1(us)	26.11	11.58
Sort and Sqrt/step2 & 3(us)	70.69	3.82
D2H memory copy/step4(us)	60.15	2.72
Post-processing/CPU(us)	16.85	3.85
Total time(us)	173.8	21.96
Speed (images/s)	5,753	45,539

5.3 GPU Efficiency

Table 3 shows the detailed time distribution when batch size is set from 1 to 1204. From Table 3, we can see that FP16 GEMM (HGEMM in cuBLAS library) computation time is reduced by 55.6% because of the much larger matrix and higher data reuse. The sorting process time is significantly reduced by 94.5%. In our CUDA kernel design, the sorting of each column of A is executed in a single thread. If the batch size is 1, there are totally 768 threads could be utilized in parallel, which is still a very small part of the modern GPU's thread capacity. When the batch size is set to 1024, there would be nearly 0.8 million sorting tasks, and all the GPU's threads could be fully utilized. Besides, the time of moving data from GPU memory to host memory is also significantly reduced because it is more efficient to transmit a large block in single DMA than to transmit multiple small blocks. The post-processing time is also reduced since more CPU parallelism could be explored.

We also consider the GPU efficiency which is defined in Eq. 3.

$$GPU\_efficiency = \frac{Achieved\_TFLOPS}{Theoretical\_TFLOPS} \tag{3}$$

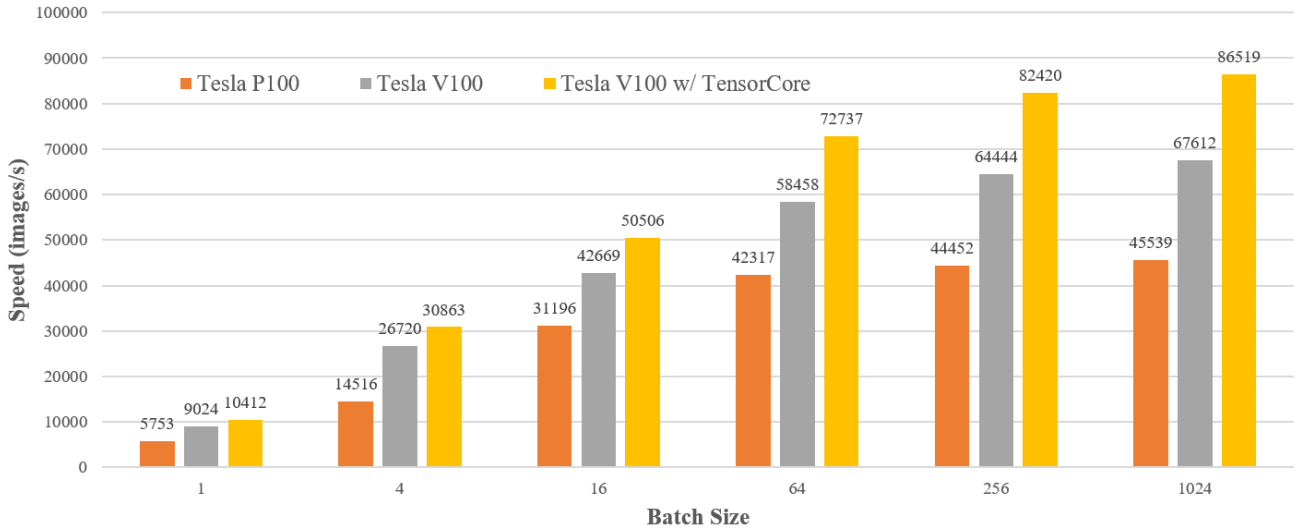

Figure 4: search speed with RootSIFT and batching technology ($m = n = 768$, $d = 128$) measured in Nvidia Tesla P100/16GB and V100/16GB GPUs with FP16. The batch size is ranged from 1 to 1024.

We achieved similar efficiency of about 36% for both P100 and V100 (w/o tensor core) GPU, which is still relatively low. But, if we only consider the HGEMM part, the efficiency could reach up to 67.9% and 65.7% for the P100 and V100, respectively. As illustrated in Table 3, except the HGEMM step, the sorting step is not computation intensive, and the other steps do not involve any GPU computations but they block the compute processes.

Similar to the batch process for reference feature matrix, the query feature matrix can also be batched for higher performance. However, the search latency also increases with worse achievable QoS. Similar trade-offs among throughput performance, end-to-end latency, and result quality (e.g., the accuracy result) have already been well discussed in recent hardware acceleration designs for running deep neural network inference with batched input data [30, 31], so we will not cover them in this paper.

6 HYBRID MEMORY CACHE

As discussed in [27], in order to achieve high search accuracy, each texture image needs to extract at least 768 SIFT features. Even with FP16, each reference feature matrix will occupy 187.5 KB GPU memory. In theory, a single 16GB P100/V100 GPU can only cache the features of 85,000 texture images without considering other GPU memory expense, which will significantly limit the capacity of the whole search system.

6.1 Hybrid Memory Cache

To significantly improve the capacity of keeping reference features, in this paper, we propose a hybrid memory cache scheme by using the GPU memory as the *first level cache* and the much larger CPU host memory as the *second level cache*. The proposed design is shown in Fig. 5. These two caches are performed in a first-in-first-out (FIFO) manner, where the new reference feature matrix is firstly enqueued to the GPU memory, and the oldest one will be swapped to the host memory once the GPU memory is full. If batching technology is adopted, the swap-out granularity is an entire batch. With this hybrid memory caching scheme, in theory, the capacity of a single node with 16GB GPU memory and extra 64GB CPU host memory can be improved by 5×.

However, if the reference matrices are stored in the host memory, an explicit host to GPU memory copy is required. As illustrated in Table 5, the search speed is decreased from 45,539 images/s to 25,362 images/s with a significant drop of 43.9% even if the pinned memory is enabled to eliminate an extra memory copy on the host side. The bottleneck is the PCIe interface bandwidth. Tesla P100 GPU adopts PCIe Gen3 16x, and its peak bandwidth is 16 GB/s. Considering the overhead, the actual bandwidth we tested in the cloud virtual machine is around 9.4 GB/s even with a very large transmission block. Therefore, 44.3% of the time is used to move the feature data from host memory to GPU memory, resulting in very low GPU efficiency. In the next subsection, we will discuss how to relax this negative effect.

6.2 Multiple CUDA Streams and Threads

When using Nvidia's GPUs, the computation and data transmission could be conducted simultaneously, which allows us to use multiple CUDA streams to overlap the computation and memory copy. In our design, we will use multiple CPU threads to process the search. Each thread takes the responsibility of one part of the reference feature matrices using a dedicated CUDA stream. Therefore, the number of CPU threads is equal to the number of CUDA streams. Usually, all the reference feature matrices are divided equally according to the number of enabled CPU threads. Therefore, not only the CUDA core computation and data movement could be overlapped, the CPU and GPU computation can also be well overlapped.

The performance of using multiple CPU threads/CUDA streams is shown in Table 6. In this experiment, we assume all the reference feature matrices are stored on the host memory. Due to the bottleneck of PCIe bandwidth (9.6 GB/s in real test), the theoretical speed

Table 4: GPU efficiency, $m = n = 768$, $d = 128$, batch size is 1024.

GPU Type	Speed (images/s)	Achieved TFLOPS	Theoretical TFLOPS (FP16)	Efficiency
Tesla P100 card	45,539	6.69	18.7	35.8%
Tesla V100 card w/o Tensor Core	67,612	9.94	28	35.5%
Tesla V100 card w/ Tensor Core	86,519	12.72	112	11.4%

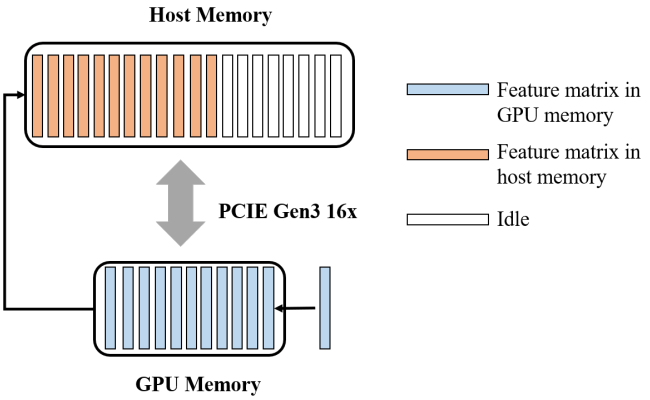

Figure 5: Hybrid memory cache by concatenating the GPU and host memory in a FIFO way.

Table 5: Performance with hybrid memory cache, $m = n = 768$, $d = 128$, batch size is 1024, measured in Nvidia Tesla P100/16GB GPU, PCIE Gen3 16x.

Cache Type	Speed (images/s)
GPU Memory	45,539
Host Memory w/o Pinned memory	17,619
Host Memory w/ Pinned memory	25,362

is 47,592 images/s. Here we use schedule efficiency to measure the performance gain obtained by multiple CPU threads/CUDA streams, which is defined in Eq.4.

$$schedule\_efficiency = \frac{Achieved\_speed}{Theoretical\_speed} \tag{4}$$

Table 6 shows that multiple streams can significantly boost the speed performance. For batch size = 512, we achieved 41,546 images/s using 8 streams with the schedule efficiency of 87.3%, which is very close to the theoretical peak speed. However, more extra GPU memory is used since each stream needs a dedicated GPU memory to store some temporary intermediate data, such as matrix A. These encouraging results show multiple streams could well overlap the computation and data transmission. The more streams are used, the higher speed is achieved.

7 ASYMMETRIC LOCAL FEATURE EXTRACTION

Among all the discussions above, we assume the same number of features extracted from reference and query images. By carefully investigating the algorithm of ratio test after 2-nearest neighbors computation during the image matching, we found that reference features are only used for ratio tests to distinguish distinct features (or keypoints) from non-distinct features in the query texture image. And usually, hundreds of features are extracted in each reference texture image. We suppose a slightly reducing feature number of reference texture images will not significantly affect the search accuracy.

In Table 7, m and n are the number of features extracted from reference and query images, respectively. We firstly fix $n = 768$ and reduce m from 768 to 256. The accuracy loss can be neglected when $m \geq 384$. But if we further reduce m to 256, significant accuracy loss is observed. Secondly, we fix $m = 384$ and reduce n from 1024 to 384. The accuracy is significantly affected, which indicates keeping a higher number of features for query images is more important than that for the reference image. This interesting finding validates our assumption.

From Table 7, we can observe that the optimal solution is $m = 384, n = 768$. The accuracy loss is very limited (only 0.28%), and the speed is improved by 34.6% compared to the baseline with $m = 768, n = 768$. In this scenario, the size of reference feature matrix is reduced by 50%. Thus, the capacity of the search system could be doubled, and the PCIe bandwidth requirement is also relaxed.

8 DISTRIBUTED TEXTURE SEARCH SYSTEM

We build a distributed texture search system in Alibaba Cloud with seven GPU virtual machines and one CPU virtual machine. Each GPU virtual machine is equipped with two Nvidia Tesla P100/16GB GPU cards. Kubernetes is used for resource management. As shown in Fig. 6, the search system totally contains 14 GPU containers for texture search (one GPU card for each container), one CPU container for Redis service, and 4 CPU containers for providing RESTful API based web service. The database is used for storing the reference feature matrices, which are serialized with Google's protobuf.

In each GPU container, 64GB host memory is reversed for caching the reference feature matrices. All the reference feature matrices are equally allocated to those 14 GPU containers. A web service is deployed to provide texture image related services. We can add, delete, update, and search a texture image through the provided APIs in this system. Regarding the 16 GB GPU memory, 4 GB is

Table 6: GPU efficiency with multiple CPU threads and CUDA streams, $m = n = 768$, $d = 128$, measured in Nvidia Tesla P100/16GB GPU.

BatchSize	CUDA Streams	Extra GPU Memory (GB)	Speed (images/s)	Efficiency
512	1	0.989	24,984	52.5%
512	2	1.667	29,459	61.9%
512	4	3.027	37,955	79.8%
512	8	5.819	41.546	87.3%
256	1	0.683	24,554	51.5%
256	2	0.911	28.259	59.3%
256	4	1.701	36.733	77.2%
256	8	3.053	40.310	84.7%

Table 7: Performance of asymmetric feature number for reference and query texture images, $d = 128$, batch size is 256, measured in Nvidia Tesla P100/16GB GPU.

m (reference)	n (query)	Accuracy	Speed (images/s)
768	768	97.74%	46,323
512	768	97.74%	57,859
384	768	97.46%	62,356
256	768	94.07%	68,472
384	1024	98.02%	46,204
384	768	97.46%	62,356
384	512	95.76%	91,367
384	384	91.81%	111,818

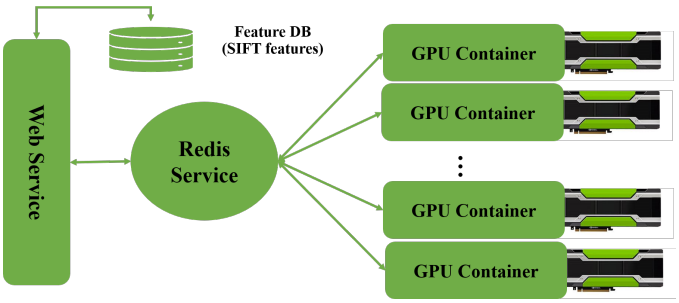

Figure 6: Distributed search system architecture.

reserved for the search engine's intermediate data. The remainder of memory is to cache reference feature matrices, resulting in a hybrid cache size of 76 GB for each container. With 14 containers, we have 1,064 GB memory for cache, which can store 10.8 million feature matrices ($m = 384$, FP16) of the reference texture images. The overall search speed is 872,984 images/s. Thus, we can complete the million-scale search within only 1.15 seconds.

9 CONCLUSION

This paper presented a comprehensive set of optimization strategies from both hardware and software aspects and delivered a highly optimized large-scale texture identification system on distributed GPUs. To improve the GPU acceleration design, we proposed a cuBLAS implementation of the 2-nearest neighbors algorithm, a batch process to handle the reference feature metrics, and a hybrid cache design to expand memory capacity. To make the algorithm more efficient, we adopted a RootSIFT algorithm to simplify the matching procedure and proposed an asymmetric local feature extraction to compress the feature space. With the above HW/SW co-optimization strategies, we deployed a large-scale texture identification system on 14 Nvidia Tesla P100 GPUs. It can deliver 31× faster search and 20× larger feature cache capacity compared to a baseline design using conventional CUDA implementation and its overall search speed can reach 872,984 images/s.

REFERENCES

[1] R. Arandjelović and A. Zisserman. 2012. Three things everyone should know to improve object retrieval. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
[2] H. Bay, T. Tuytelaars, and L. V. Goo. 2008. Speeded-up robust features (SURF). *Computer vision and image understanding* 110, 3 (2008), 346–359.
[3] S. Bell, P. Upchurch, N. Snavely, and K. Bala. 2015. Material recognition in the wild with the materials in the context database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
[4] M. Chica-Olmo and F. Abarca-Hernández. 2000. Computing geostatistical image texture for remotely sensed data classification. *Computers & Geosciences* (2000).
[5] O. G. Cula and K. J. Dana. 2001. Compact representation of bidirectional texture functions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
[6] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distribution. In *International Conference on computational Geometry*.
[7] J. Deng, J. Guo, N. Xue, and Z. Stefanos. 2019. ArcFace: Additive Angular Margin Loss for Deep Face Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
[8] P. Fischer, A. Dosovitskiy, and T. Brox. 2014. Descriptor matching with convolutional neural networks: a comparison to SIFT. *arXiv preprint arXiv:1405.5769* (2014).
[9] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. 2010. k-nearest neighbor search: fast GPU-based implementations and application to high-dimensional feature matching. In *Conference on Image Processing (ICIP)*.
[10] H. Jegou, M. Douze, and C. Schmid. 2011. Product quantization for nearest neighbor search. *IEEE Transactions on Software Engineering* 33, 1 (2011), 117–128.
[11] H. Jegou, M. Douze, C. Schmid, and P. Perez. 2010. Aggregating local descriptors into a compact image representation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
[12] J. Johnson, M. Douze, and H. Jegou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* (2019).

[13] A. Karahaliou, S. Skiadopoulos, I. Boniatis, et al. 2007. Texture analysis of tissue surrounding microcalcifications on mammograms for breast cancer diagnosis. *The British journal of radiology* (2007).
[14] K. Kumar and K. G. H. Pang. 2002. Defect detection in textured materials using Gabor filters. *IEEE Transactions on Industry Applications* 38, 2 (2002), 425–440.
[15] Y. Kusamura, Y. Kozawa, T. Amagasa, and H. Kitagawa. 2016. GPU acceleration of content-based image retrieval based on SIFT descriptors. In *International Conference on Network-Based Information Systems (NBiS)*.
[16] D. G. Lowe. 1999. Object Recognition from Local Scale-invariant Features. In *IEEE International Conference on Computer Vision (ICCV)*.
[17] D. G. Lowe. 2004. Distinctive image features from scale-invariant keypoints. *International journal of computer vision (IJCV)* 60, 2 (2004), 91–110.
[18] J. Ma, X. Jiang, A. Fan, J. Jiang, and J. Yan. 2021. Image Matching from Handcrafted to Deep Features: A Survey. *International Journal of Computer Vision (IJCV)* 129, 1 (2021), 23–79.
[19] D. Nister and H. Stewenius. 2006. Scalable recognition with vocabulary tree. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
[20] H. Noh, A. Araujo, J. Sim, et al. 2017. Large-scale mage retrieval with attentive deep local features. *IEEE International Conference on Computer Vision (ICCV)* (2017).
[21] F. Perronmin, J. Sanches, and T. Mensink. 2010. Improving the fisher kernel for large-scale image classification. In *European conference on computer Vision (ECCV)*.
[22] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. 2011. ORB: and efficient alternative to SIFT or SURF. In *IEEE International Conference on Computer Vision (ICCV)*.
[23] L. Sharan, C. Liu, R. Rosenholtz, and E. H. Adelson. 2013. Recognizing materials using perceptually inspired features. *International journal of computer vision* 103, 3 (2013), 348–371.
[24] K. Shlizerman, S. M. Seitz, D. Miller, and E. Brossard. 2016. The megaface benchmark: 1 million faces for recognition at scale. In *IEEE conference on computer vision and pattern recognition (CVPR)*.
[25] C. Silpa-Anan and R. Hartley. 2008. Optimized kd-trees for faster image descriptor matching. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
[26] J. Sivic and A. Zisserman. 2003. Video Google: A text retrieval approach to object matching in videos. *IEEE International Conference on Computer Vision (ICCV)* (2003).
[27] J. Wang, Y. Li, Z. Chang, et al. 2021. Fine-grained texture identification for reliable product traceability. *arXiv preprint arXiv:2104.11548* (2021).
[28] K. M. Yi, E. Trulls, V. Lepetit, and P. Fua. 2016. LIFT: Learned invariant features transform. In *European conference on computer Vision (ECCV)*.
[29] H. Zhang, J. Xue, and K. Dana. 2016. Deep TEN: Texture Encoding Network. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
[30] X. Zhang, H. Lu, C. Hao, et al. 2020. SkyNet: a hardware-efficient method for object detection and tracking on embedded systems. In *Machine Learning and Systems (MLSys)*.
[31] X. Zhang, J. Wang, C. Zhu, et al. 2018. DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.