

# **LARGE-SCALE TEXTURE IDENTIFICATION WITH DISTRIBUTED GPU ACCELERATION FOR PRODUCTS TRACEABILITY**

Junsong Wang (王均松)

[junsong.wang@easy-visible.com](mailto:junsong.wang@easy-visible.com)

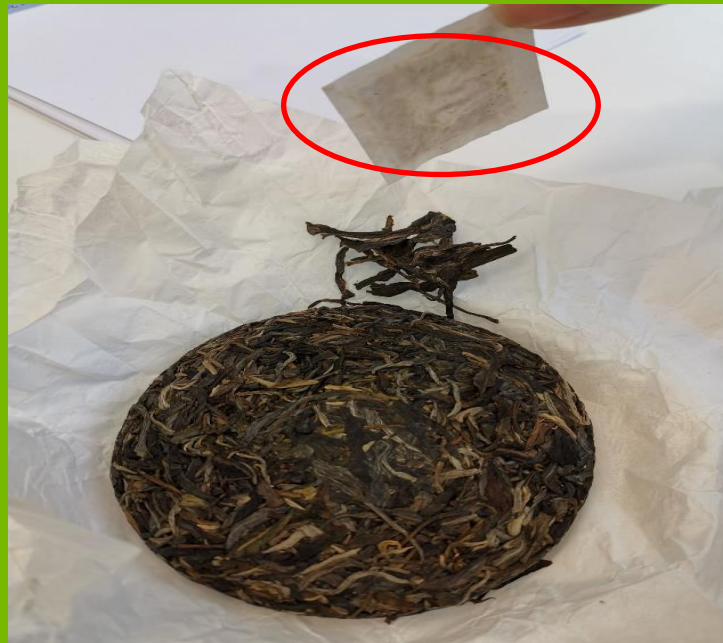
V-Origin Technology, Beijing, China

April 12, 2021

# Traditional Traceability Technologies

## Disadvantages:

- BarCode and QRCode are both very easy to fake.
- Label can be recycled and attached in another faked product.



NFC in Pu'er tea-bricks



QRCode in bird's nest



RFID in Cow



BarCode in lemon

All the label based technologies are only protecting the label (either BarCode/QR code or electronic tag), not the product itself.



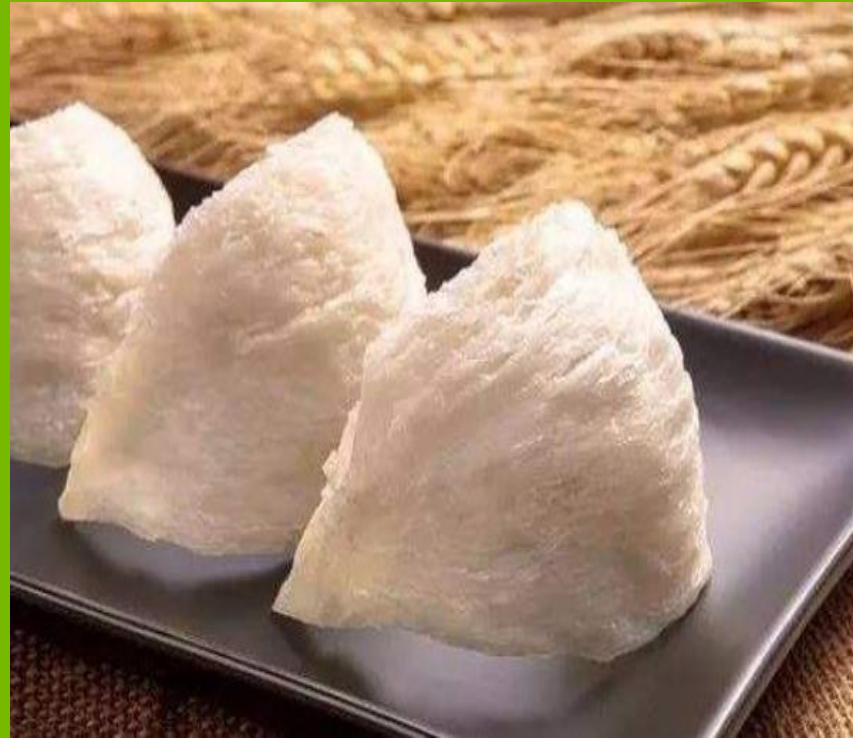
# Texture Based Traceability

Natural texture exists in lots of the products. Some of them are born with different nature texture patterns, such as wood, jade, bird's nest and meat(ham). Some of them are generated during production, such as compression tea and cork of wine.

Pu'er Tea-brick



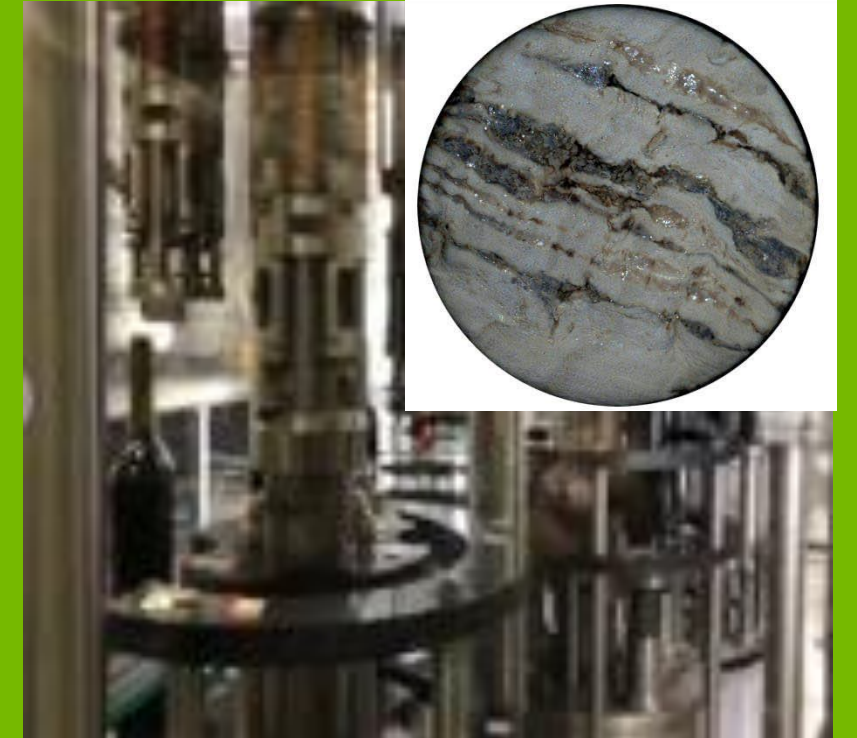
Bird's Nest



Ham



Wine



The best approach for reliable traceability is to extract some natural and unique information from the product itself, which is impossible to be duplicated or counterfeited.



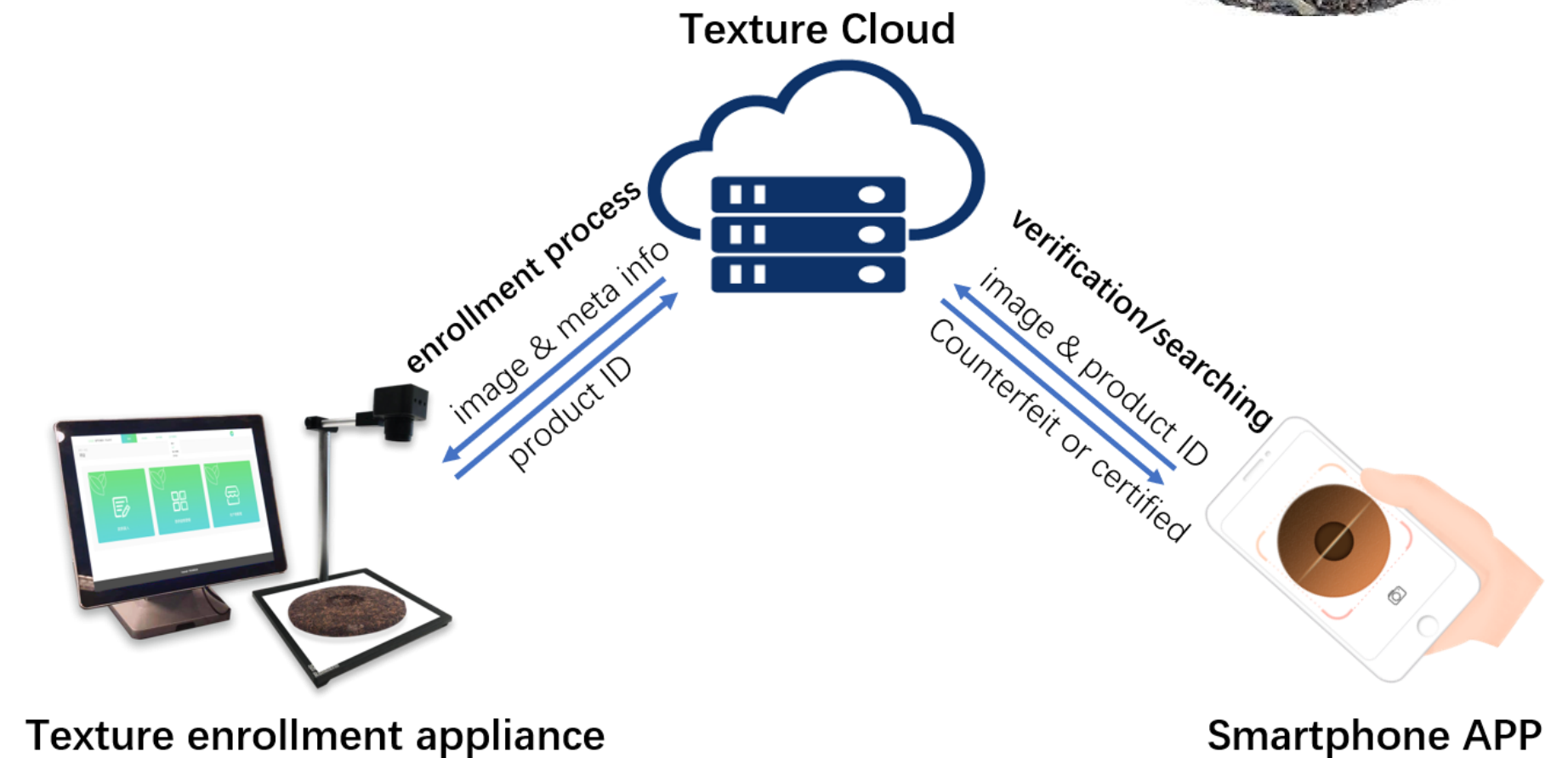
# Texture Traceability System Overview

**Texture enrollment appliance:** Designed for the manufacture's worker to capture the product's texture image and enroll to our TI-Trace system.

**Light-weight APP on the smartphone:** Developed for customers to perform product verification and searching using its natural texture.

## Texture cloud:

- Texture image and product information management.
- Texture verification by calculating the similarity between the enrolled image and the uploaded one.
- Retrieve an enrolled product only by uploading a texture image.





# Texture Recognition

## 1-1 Verification



+

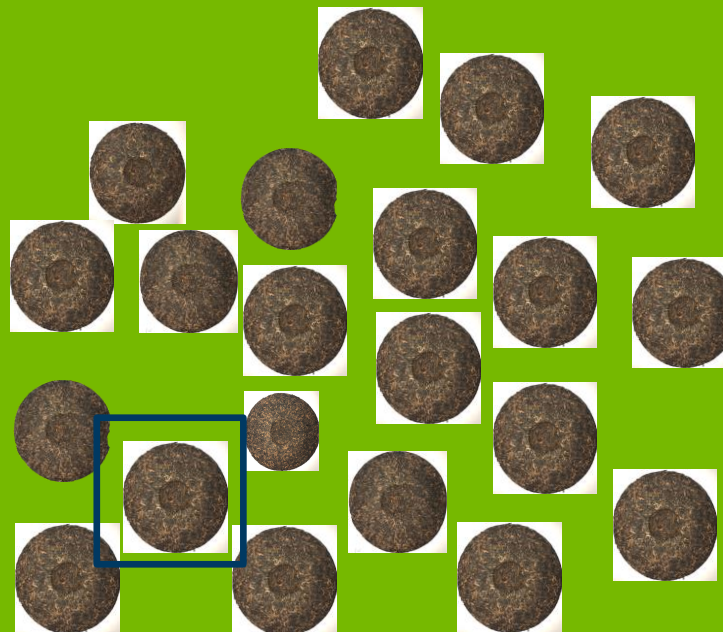


Similar to facial recognition, texture recognition should also support 1:1 verification and 1:M searching, and M could be million scale.

The recognition algorithm should be robust to:

- Diverse capturing viewpoints.
- Serious Occlusions.
- Insufficient Illuminations.

## 1-M Searching



Ideal case



Bad illumination



Bad viewpoint



Serious occlusion

# Texture Recognition Algorithm Study

## Challenges comparing to facial recognition:

- No landmark can be extracted, and the texture image can not be well aligned.
- No sufficient data for training, the facial database has 10+ million facial images.
- The 1:M fine-grained texture identification has never been discussed in academic research or industry .

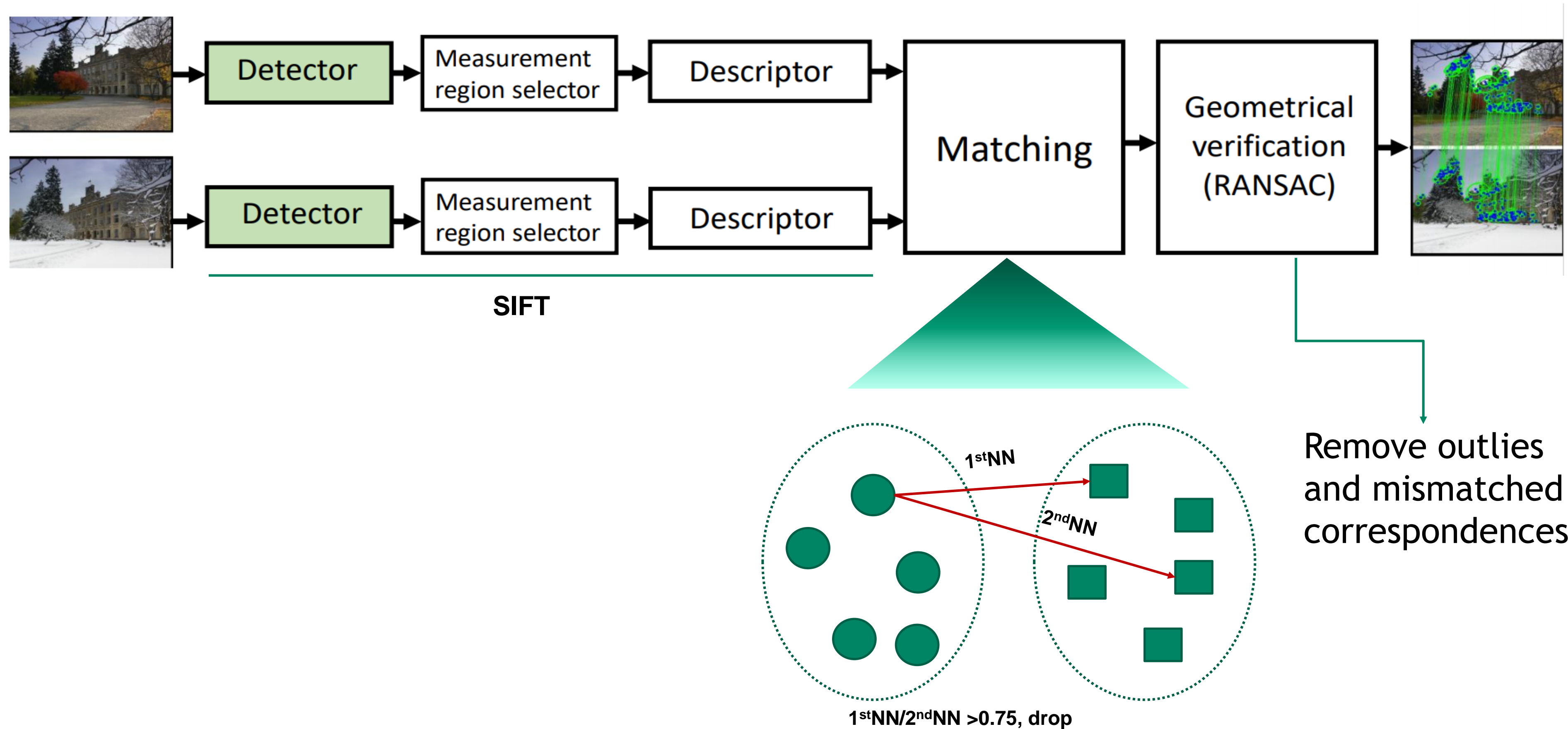
Methods	1:1 verification accuracy
Image Retrieval (Bag of Words + TIDIF Ranking)	73.0%
Metric Learning (ResNet50 + Triplet Loss)	80.1%
Image Matching (SIFT + Nearest Neighbor Matching)	99.6%

*\* Accuracy evaluated on Pu'er tea-brick images*

Image Matching can performed well and can be applied for texture recognition.



# Image Matching Pipeline



# Nearest Neighbor (NN) Matching

The computation complexity of Nearest Neighbor (NN) matching includes:

**Compute the Euclidean distance for each pair of descriptors from the two images.**

- Computation intensive, computation complexity is  $O(kn^2)$ , where  $k$  is the dimension of feature descriptor, and  $n$  is the number of feature descriptors.
- Considering 768 SIFT descriptors, each matching requires 75 million multi-add operations, and the searching in a million texture images need **75 Tera** operations.

**For each query descriptor, find the top-2 nearest neighbors**

- Bandwidth intensive
- Reduce the data movement as much as possible.



# GPU Acceleration for NN Matching

**Prior works:** OpenCV library already has a CUDA implementation for the NN matching, but the performance is relatively low, only has 2,937 images/second in Nvidia Tesla V100 GPU card, which is far beyond real-time large-scale searching. The GPU's capacity is not well explored.

## Optimization Metrics:

- **Capacity:** how many texture images (768 SIFT descriptor for each image) can be stored in the system.
- **Speed:** how many texture images can be searched in one second.

We involve the following optimization strategies to improve the capacity and speed:

- cuBLAS implementation
- Half Precision Implementation
- Hybrid Caching
- Batching
- multiple CUDA streams
- Descriptor Normalization
- Distributed Computing

# Optimization 1: cuBLAS implementation

The core computation is two vectors' Euclidean distance

$$\rho^2(x, y) = (x - y)^T (x - y) = \|x\|^2 + \|y\|^2 - x^T y$$

$$\rho^2(\mathbf{R}, \mathbf{Q}) = N_R + N_Q - 2\mathbf{R}^T \mathbf{Q}$$

The core of cuBLAS is GEMM:

$$\mathbf{C} = \alpha * \mathbf{A} * \mathbf{B} + \beta * \mathbf{C}$$

1. Compute the vector  $N_R$  using CUDA, (could be calculated offline)
2. Compute the vector  $N_Q$  using CUDA.
3. Compute the  $m \times n$ -matrix  $\mathbf{A} = -2\mathbf{R}^T \mathbf{Q}$  using cuBLAS;
4. Add the  $i$ th element of  $N_R$  to every element of the  $i$ th row of the matrix  $\mathbf{A}$  using CUDA. The resulting matrix is denoted by  $\mathbf{B}$  ;
5. Sort in parallel each column of  $\mathbf{B}$ ; The resulting matrix is denoted by  $\mathbf{C}$ ; (involve some tricks for top-2 sorting)
6. Add the  $j$ th value of  $N_Q$  to the first two elements of the  $j$ th column of the matrix  $\mathbf{C}$  using CUDA; The resulting matrix is denoted by  $\mathbf{D}$ ;
7. Compute the square root of the first two elements of  $\mathbf{D}$  to obtain the two smallest distances; The resulting matrix is denoted by  $\mathbf{E}$ ;




# Optimization 2: Half Precision Implementation

- Half precision is supported in modern Nvidia GPUs, it can save 50% memory usage, 2x faster.
- Half precision can explore the capability of Tensor Core, which is introduced after Volta GPU.

Optimization Methods	Capacity	Speed (images/second)
Telsa P100, OpenCV Baseline (CUDA Impl.)	43,400	2,012
Telsa P100, cuBLAS + FP32	43,400	6,711
Telsa P100, cuBLAS + FP16	86,800	5,910
Telsa V100, OpenCV Baseline (CUDA Impl.)	43,400	2,937
Telsa V100, cuBLAS + FP32	43,400	10,989
Telsa V100, cuBLAS + FP16	86,800	8,620

## For Tesla V100, 16GB, PCIE:

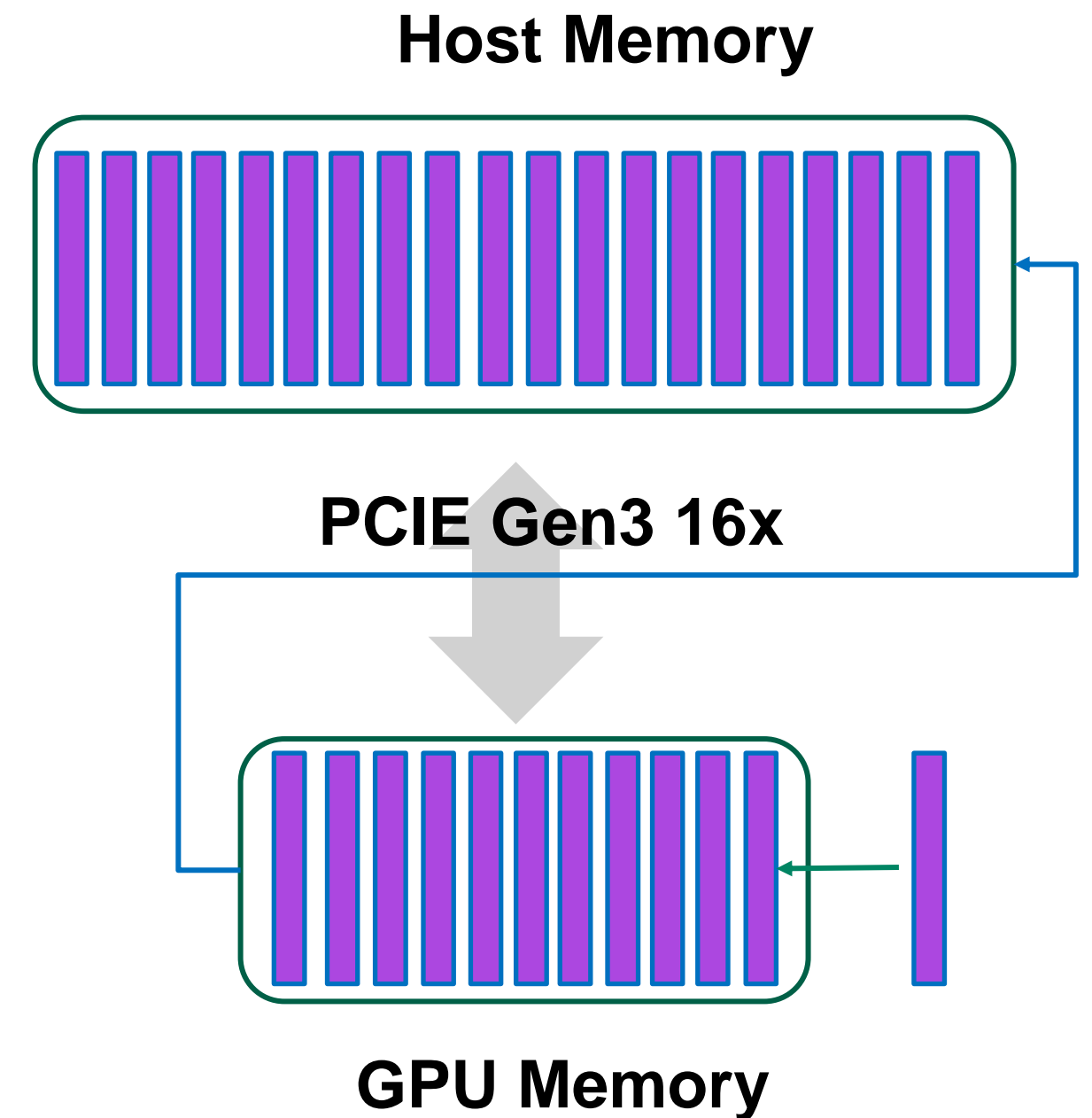
- cuBLAS + FP32, the speed is 10,989 images/s with 3.74x speedup comparing to OpenCV baseline.
  - cuBLAS + FP16, the speed is 8,620 images/s with 2.93x speedup comparing to OpenCV baseline.
  - FP16 can reduce half of the GPU memory, resulting 2x improvement in terms of capacity.
-  The speed of FP16 is decreased by 12.6% decrease comparing to FP32 implementation. The major reason is that the matrix is too small to explore the full capability of CUDA cores and tensor cores, while involves extra float<->fp16 conversion.

# Optimization 3: Hybrid Caching

Even with half precision, each image's feature descriptors need ~0.2MB GPU memory, and single 32GB Tesla V100 card only can cache ~80,000 images.

**Hybrid memory cache** scheme by using the GPU memory as the first level cache and the host memory as the second level cache and perform in a FIFO way. The new item will firstly be enqueued to the GPU memory, while oldest item in GPU memory will be swapped to host memory if the GPU memory is full.

With this hybrid memory caching scheme, the capacity of single node with extra 64GB host memory can be improved by ~5x, however the searching speed is decrease to 6378 images/s (dropped 26%) in worst case according to different searching scope, since a host to GPU memory copy is required if the descriptors are cached in host side





# Can the NN cuBLAS approach be simplified?

## SIFT Feature Descriptor Normalization

1. ~~Compute the vector  $N_R$  using CUDA~~
2. ~~Compute the vector  $N_Q$  using CUDA.~~
3. Compute the  $m \times n$ -matrix  $A = -2R^T Q$  using cuBLAS;
4. ~~Add the  $i$ th element of  $N_R$  to every element of the  $i$ th row of the matrix  $A$  using CUDA. The resulting matrix is denoted by  $B$  ;~~
5. Sort in parallel each column of  $B$ ; The resulting matrix is denoted by  $C$ ;
6. ~~Add the  $j$ th value of  $N_Q$  to the first two elements of the  $j$ th column of the matrix  $C$  using CUDA; The resulting matrix is denoted by  $D$ ;~~
7. Compute the square root of the first two elements of  $D$  to obtain the two smallest distances; The resulting matrix is denoted by  $E$ ;

 Directly normalize the SIFT descriptor will hurt the matching accuracy

# Optimization 4: Descriptor Normalization

**Introduce the RootSIFT(\*) implementation.**

- a)  $L_1$  normalize the SIFT vector;
- b) square root each element.

RootSIFT descriptor is  $L_2$  normalized

$$\sqrt{X}^T \sqrt{X} = \sum_{i=1}^N x_i = 1$$

RootSIFT descriptors using Euclidean distance is equivalent to using the Hellinger kernel

$$\rho^2(\sqrt{X} - \sqrt{Y}) = 2 - 2H(X, Y)$$

Hellinger kernel: for two  $L_1$  normalized histograms,  $x$  and  $y$ :

$$H(X, Y) = \sum_{i=1}^N \sqrt{x_i y_i}$$

Hellinger distance is used to quantify the similarity between two probability distributions

\* R. Arandjelovic and A. Zisserman, "Three things everyone should know to improve object retrieval," in *Computer Vision and Pattern Recognition (CVPR)*, 2012.



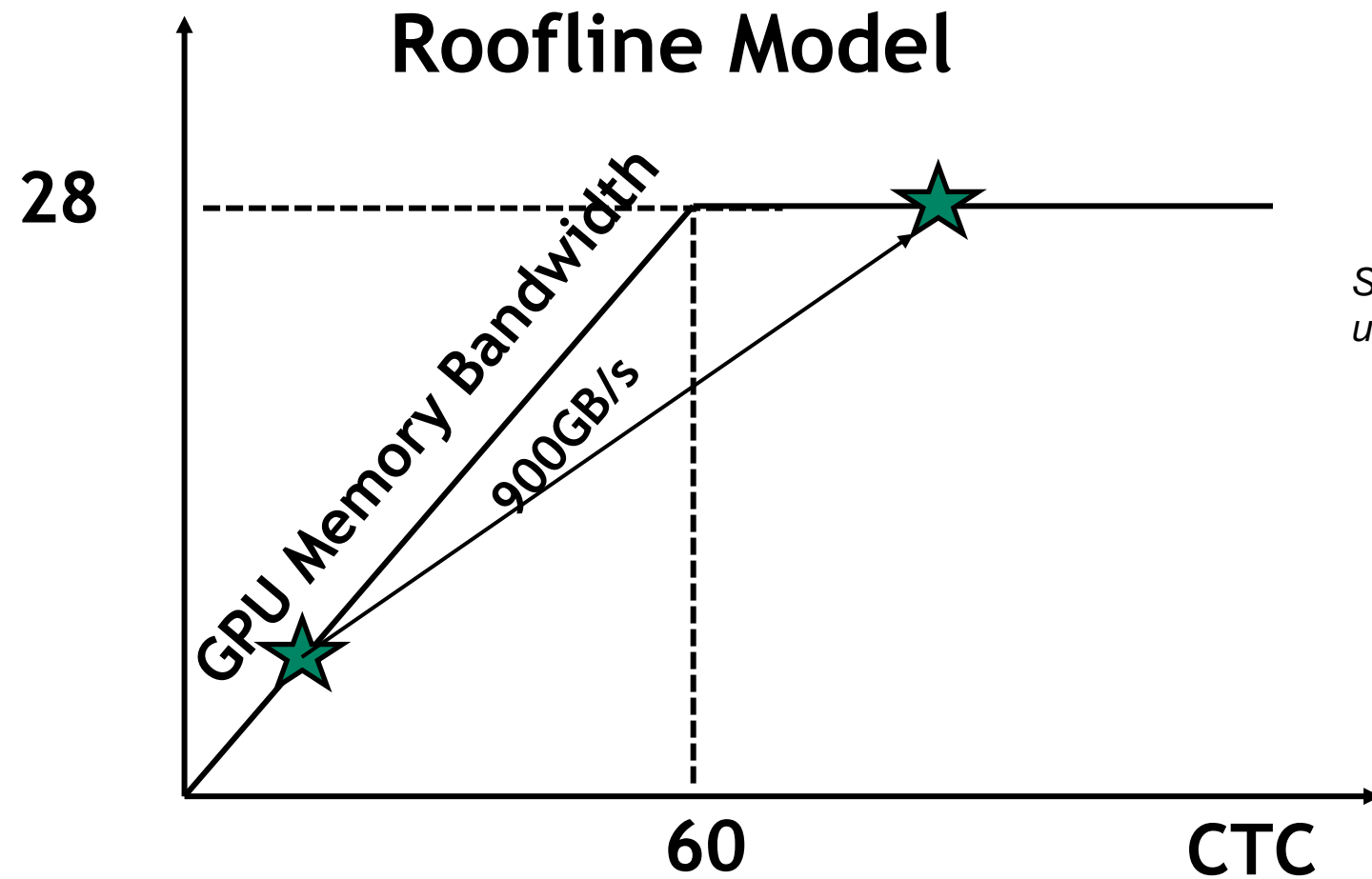
# Simplified Algorithm (only 2 steps):

1. Compute the  $m \times n$ -matrix  $A = -2R^T Q$  using cuBLAS;
2. Sort (find top-2 smallest) in parallel each column of  $A$ ; And compute the square root of the first two elements of  $2-A$

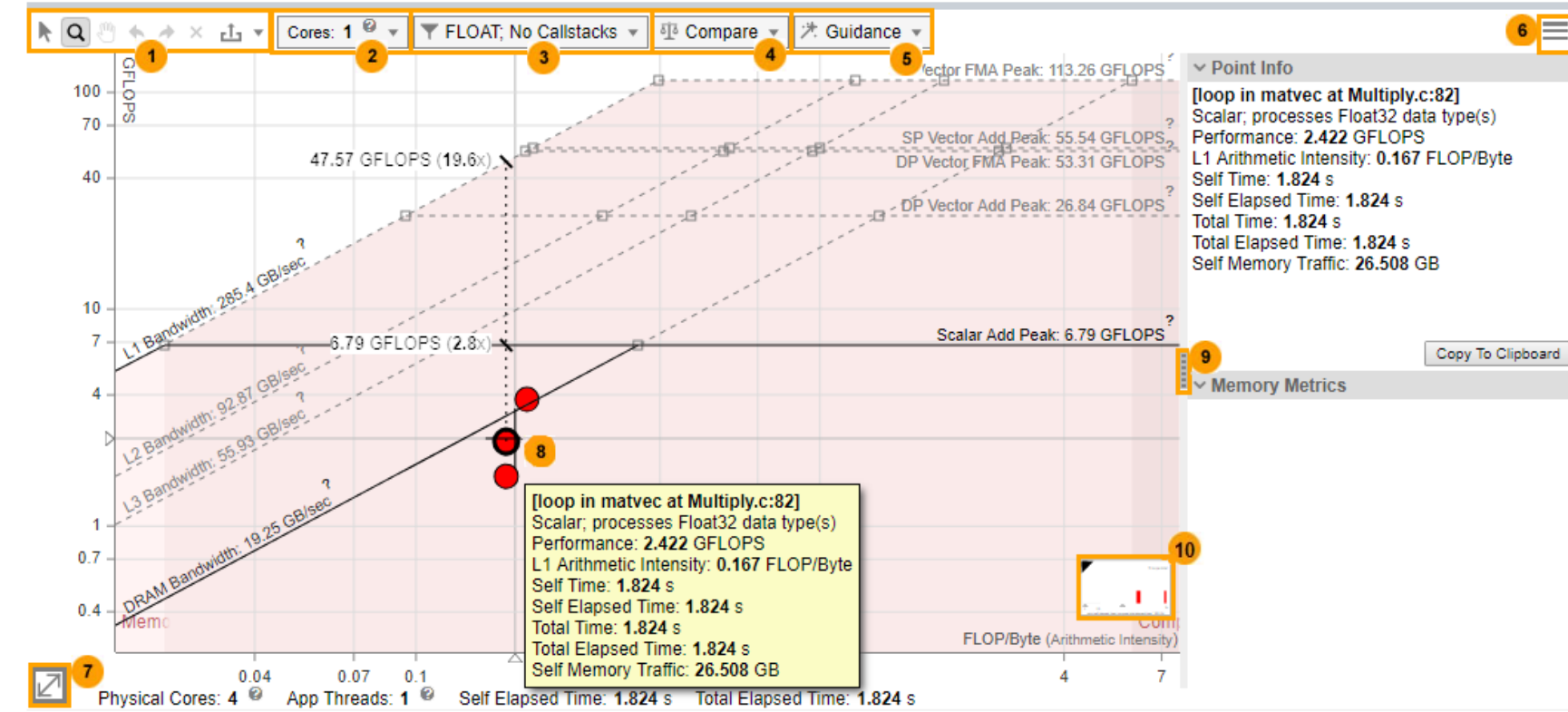
The  $2-A$  and square root could be performed in place to minimize the data movement.

# GPU Efficiency is Very Low!

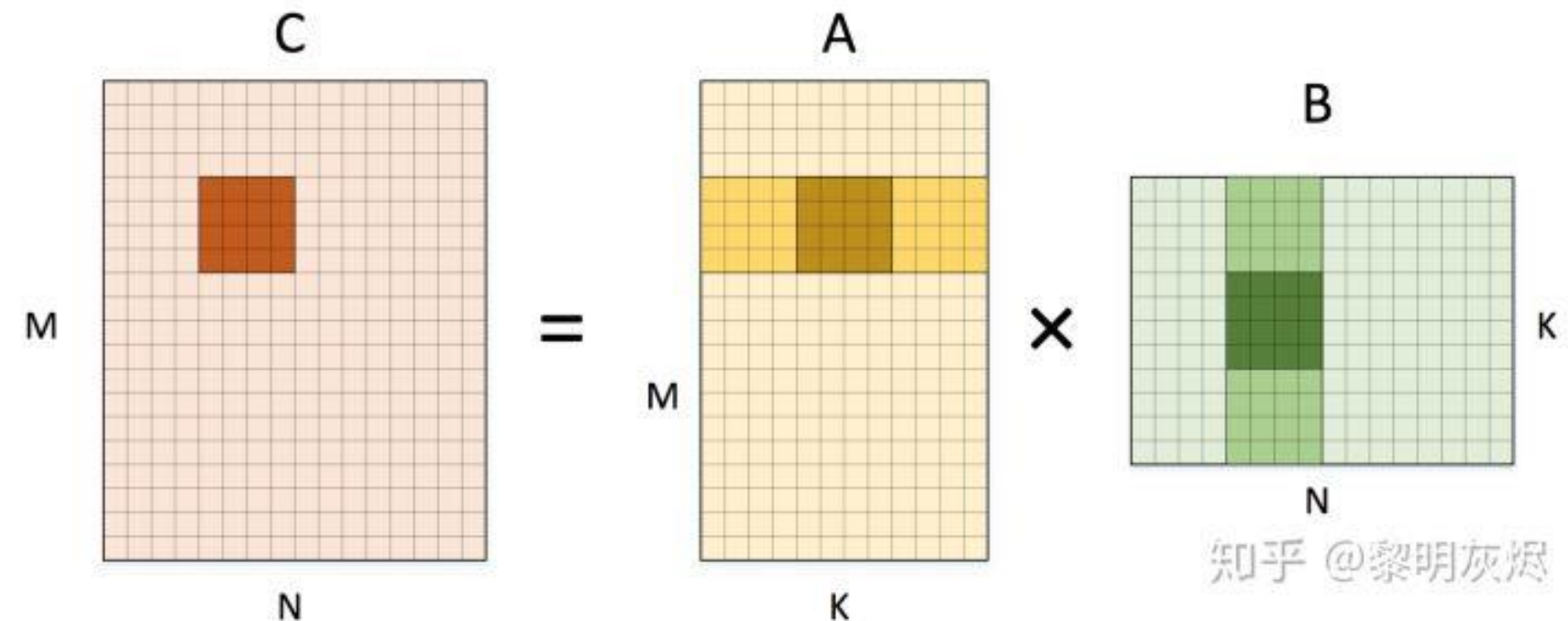
Even with cuBLAS implementation, the actual computation is only 147 Mega FLOPS \* 8620 = 1.27 TFLOS, which is far from the GPU capacity (28 TFLOS(FP16), V100 card), and the major reason is the matrix is too small. The matrix is only 768\*128.



Increasing CTC by Batching

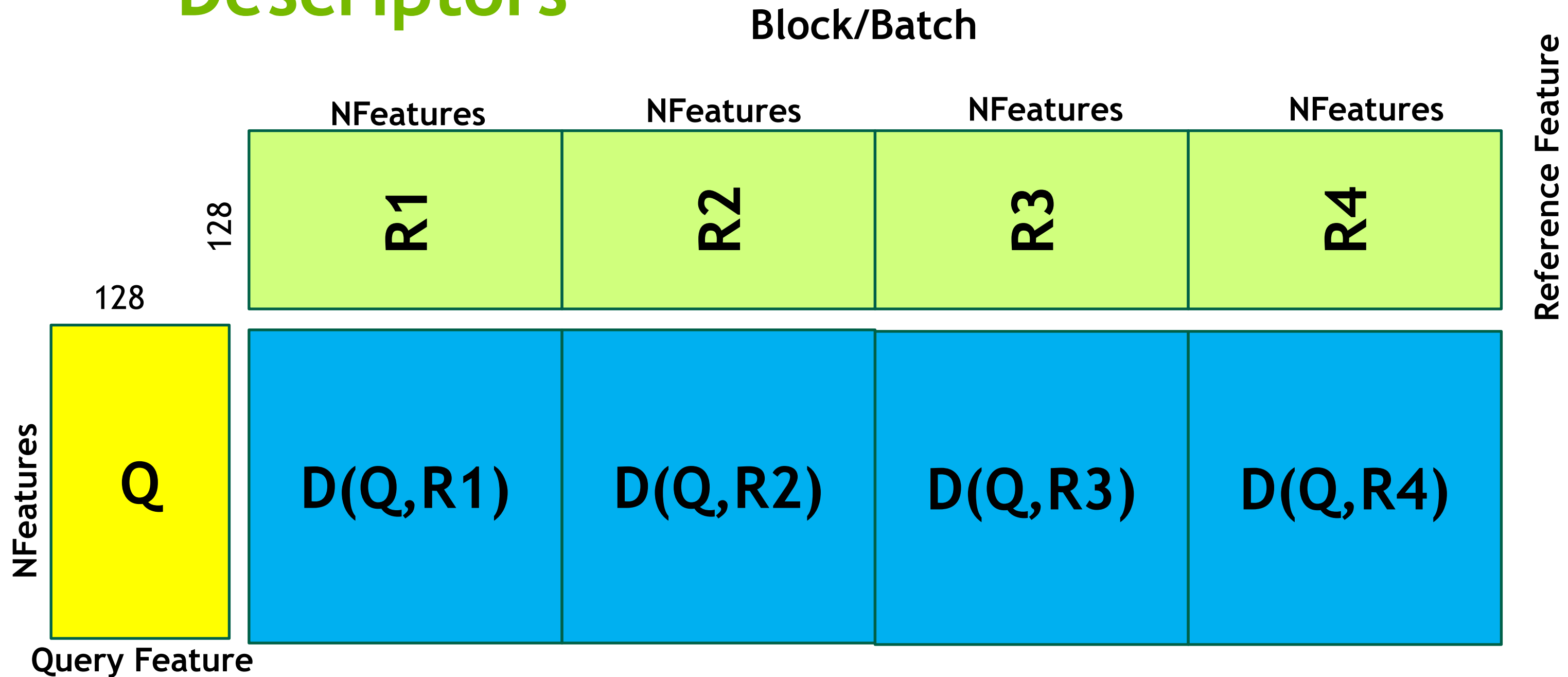


Source: <https://software.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top/optimize-cpu-usage/cpu-roofline-perspective/identify-performance-bottlenecks-on-cpu/cpu-roofline-report-overview.html>





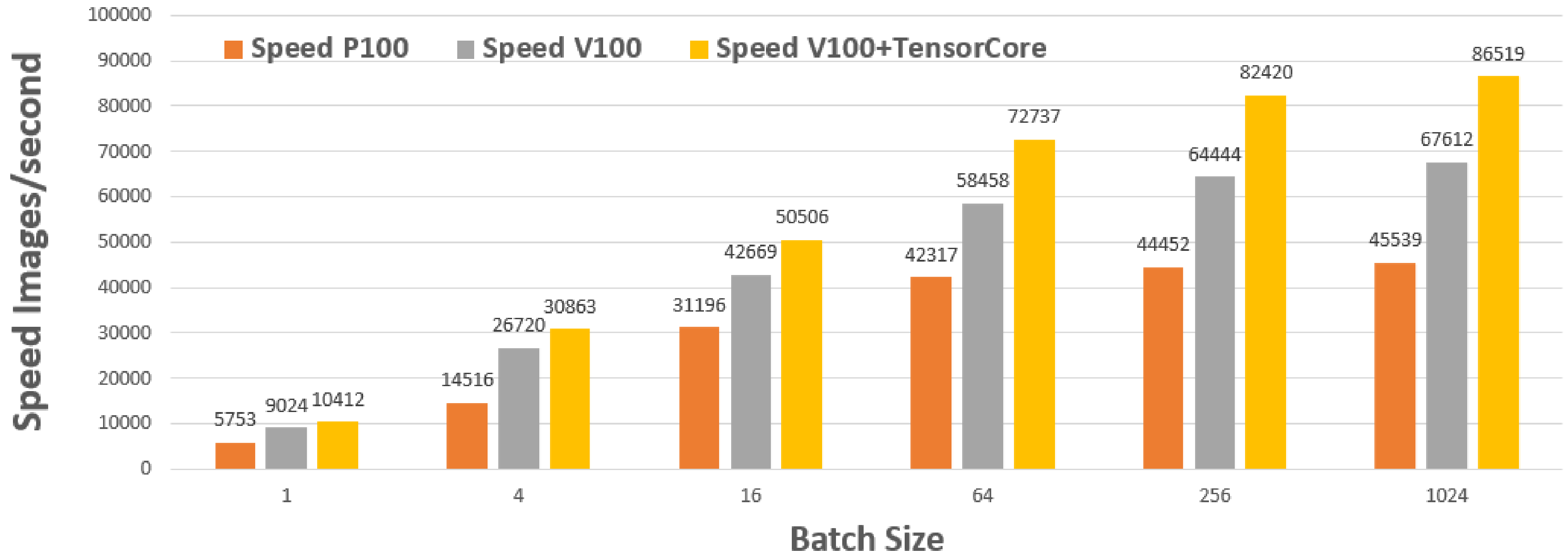
# Optimization 5: Batching the Reference Descriptors



- With batching the reference features, the size of matrix B is significantly increased, resulting in higher data reuse and CUDA core utilization.
- The following sorting process can also fully utilize the CUDA cores.

# Batching Performance

speed with different batch size, 716 SIFT descriptors. FP16



\* This is the kernel performance (distance calculation and top-2 ranking for NN algorithm) without considering the CPU/GPU memory copy and post-processing cost.

- With batch size=1024 in V100 + TensorCore, the speed is 86519 images/s with an improvement of 8.2x comparing w/o batching, 29.5x faster than OpenCV.

# GPU Efficiency

GPU efficiency with 768 descriptors in SIFT and using FP16, batch size = 1024.

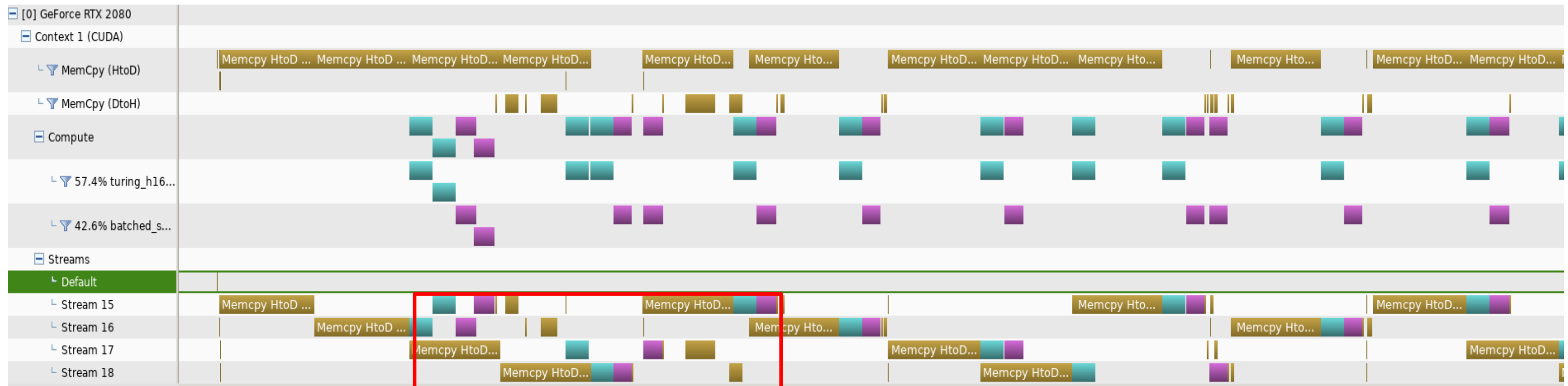
GPU Type	Speed	Achieved TFLOPS	Capability (FP16)	Efficiency
P100	45,540	6.69	18.7	35.8%
V100	67,613	9.94	28	35.5%
V100+TensorCore	86,520	12.72	112	11.4%

As discussed in optimization 3, we use host memory to enlarge the capacity, which will involve huge host->device memory copy.



# Optimization 6: Computation and MemCopy Overlap

Using multiple CUDA streams to overlap the computation and memory copy between the host and GPU.



**Kernel, H2D, D2H are well overlapped**

*Since we use Aliyun cloud GPU VM, we can directly use `nvvp` to tuning the performance. This figure is captured in our local workstation with RTX2080 card.*

# Multiple CUDA Stream Evaluation

Evaluated with Telsa P100 32GB PCIE3.0 GPU, 768 descriptors in SIFT and using FP16, query in 51,200 images, all the features are stored in the host memory. The real time of transferring 51,200 sift features from host to GPU memory costs 1.0758 second, the theoretical speed is **47,592** images/s.

Batch Size	# of CUDA Streams	Extra GPU Memory (GB)*	Speed (images/second)	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	0.991	28,259	59.3%
256	4	1.701	36,733	77.2%
256	8	3.053	40,310	84.7%

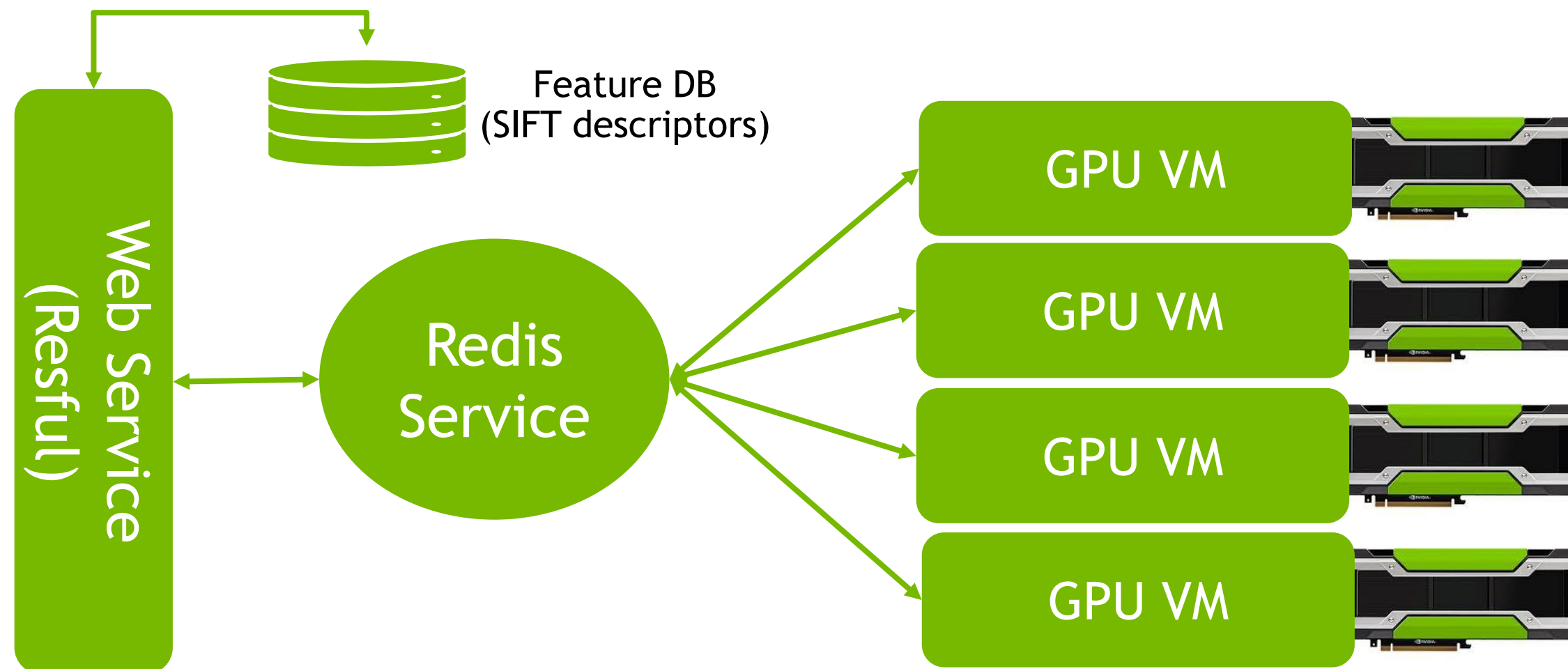


Using multiple CUDA streams can significantly improve the efficiency.

# Optimization 7: Distributed Computing

- For Tesla V100 card, the host->device bandwidth will be the bottleneck, and significant affect the speed,
- For Tesla P100 card, the computation and bandwidth are more balanced, and the computation resource in GPU can be well explored.

By considering the performance and cost, we deployed 14 P100 cards to support our first large scale texture identification use case -- Pu'er tea traceability.



- Each GPU VM has a Tesla P100 16GB PCIE3.0 16x card.
- Each GPU VM has 96GB memory, 64GB is reserved for caching SIFT descriptors (FP16).
- All the SIFT descriptors are equally allocated to 14 GPU VMs.
- Select the Redis as the message queue
- Support ADD, DELETE, UPDATE, SEARCHING functions.

Finally, this GPU Cluster has the capacity of 6 millions texture images, and the speed is 581,633 images/second.