INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING
ICPP/2021/CHICAGO/USA
acm In-Cooperation
sighpc
AUGUST 9–12, 2021

EXPLORING HW/SW CO-OPTIMIZATIONS FOR ACCELERATING LARGE-SCALE TEXTURE IDENTIFICATION ON DISTRIBUTED GPUS

Junsong Wang[1], Xiaofan Zhang[2], YuBo Li[1], Yonghua Lin[1],
[1]V-Origin Technology,
[2]University of Illinois at Urbana-Champaign

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

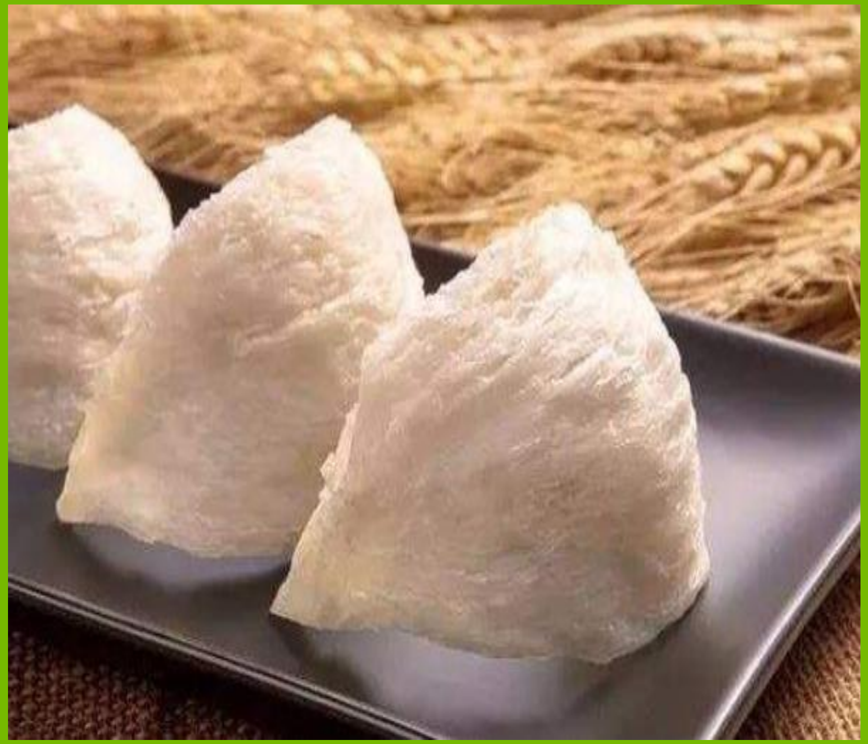
All the label based technologies are only protecting the label (either BarCode/QR code or electronic tag), not the product itself.

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

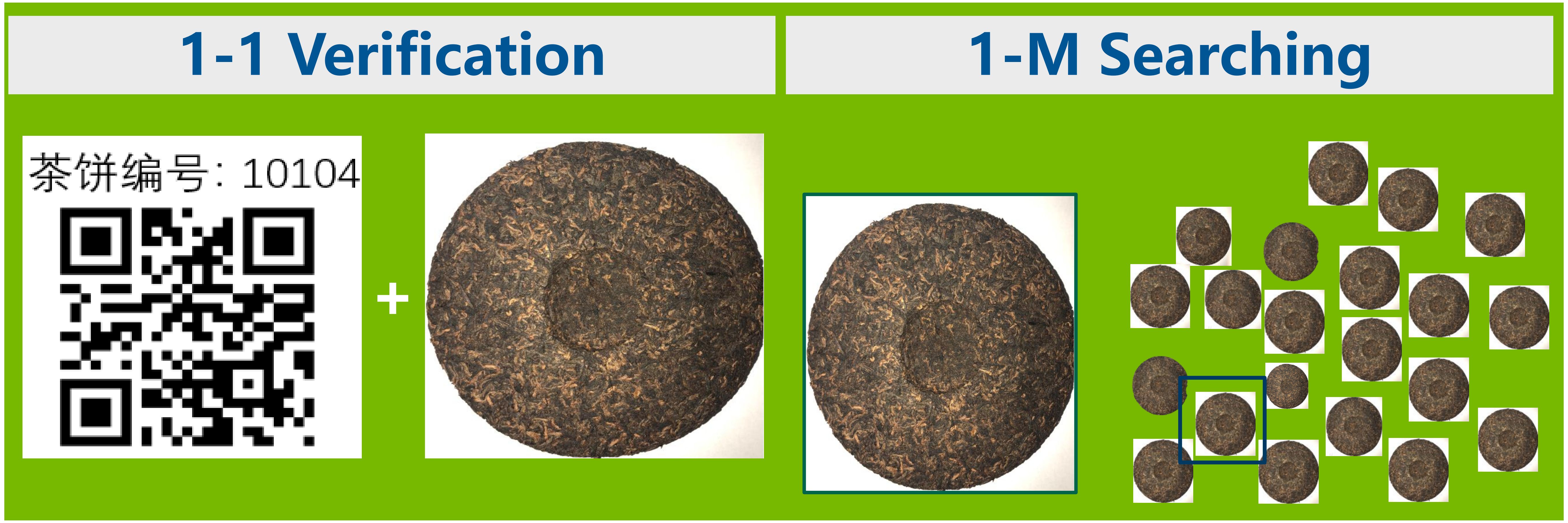


Ham

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 Recognition



1-1 Verification

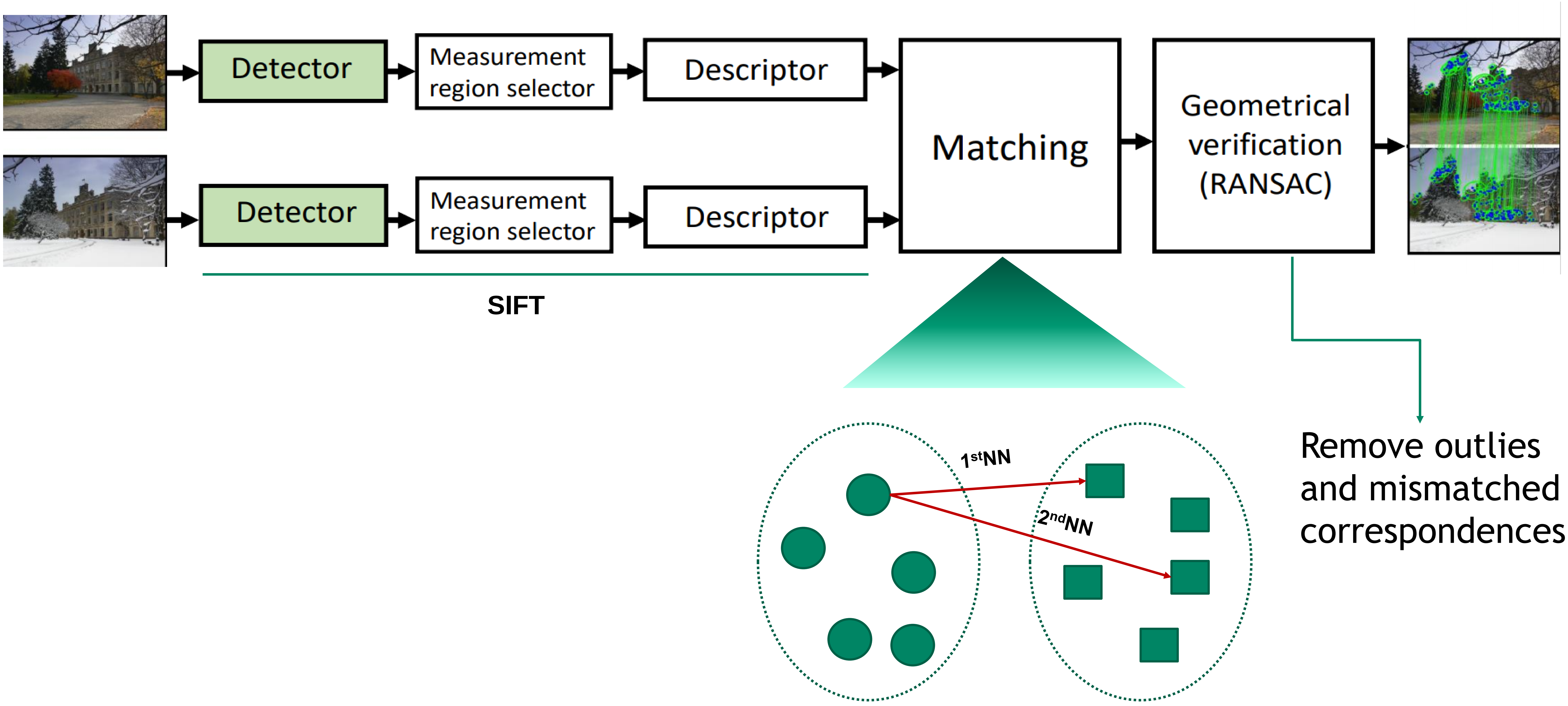
1-M Searching

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

*We have demonstrated that local feature extraction (SIFT) and nearest neighbor matching algorithm achieves the verification accuracy of 99.6% and the top-1 searching accuracy of 98.9%, respectively.

** Junsong Wang, Yubo Li, Zhiyong Chang, Haitao Yue and Yonghua Lin, Fine-grained texture identification for reliable product traceability. IEEE International Conference on Multimedia and Exop (ICME), Virtual, July 2021*

Image Matching Pipeline



Detector

Measurement region selector

Descriptor

Detector

Measurement region selector

Descriptor

Matching

Geometrical verification (RANSAC)

SIFT

1^{st}NN

2^{nd}NN

Remove outlies and mismatched correspondences

1^{st}NN/2^{nd}NN >0.75, drop

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:

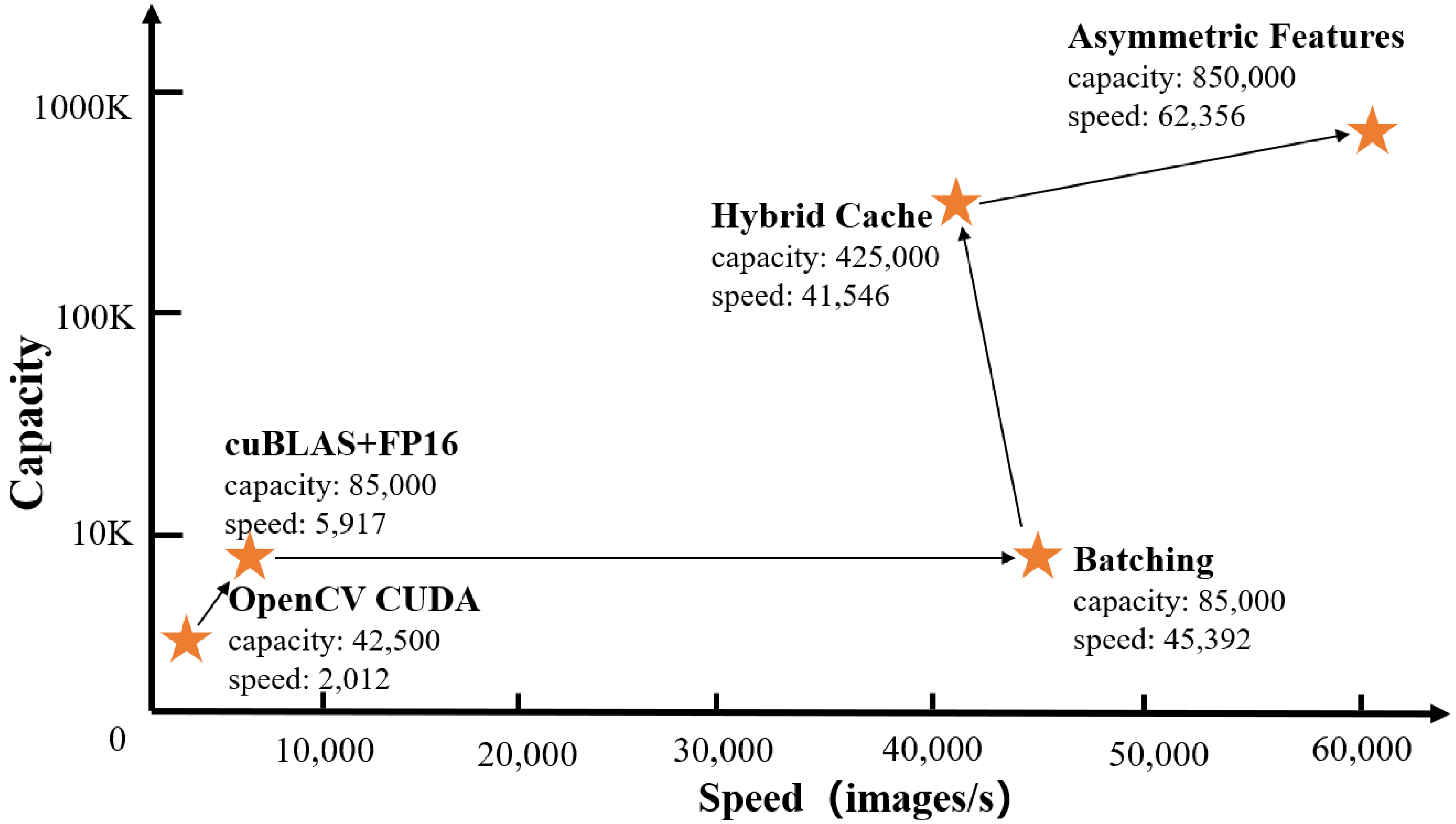
Capacity: how many feature matrices of the reference texture images can be cached in the search system’s memory.

Speed: how many texture image similarity comparisons can be completed in one second.

Optimization Strategies for NN Matching

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

1. A highly optimized **cuBLAS + FP16** implementation dedicated to the 2-nearest neighbors algorithm.
2. A **batch** process for reference feature matrices
3. A **hybrid cache** design to leverage both GPU and host memory space to significantly enlarge the memory capacity for keeping reference feature matrices.
4. An **asymmetric local feature** extraction is adopted for reducing memory footprint.



Capacity improvement: 20×, Speed improvement: 31×

Optimization 1: cuBLAS + FP16 Implementation

cuBLAS implementation:

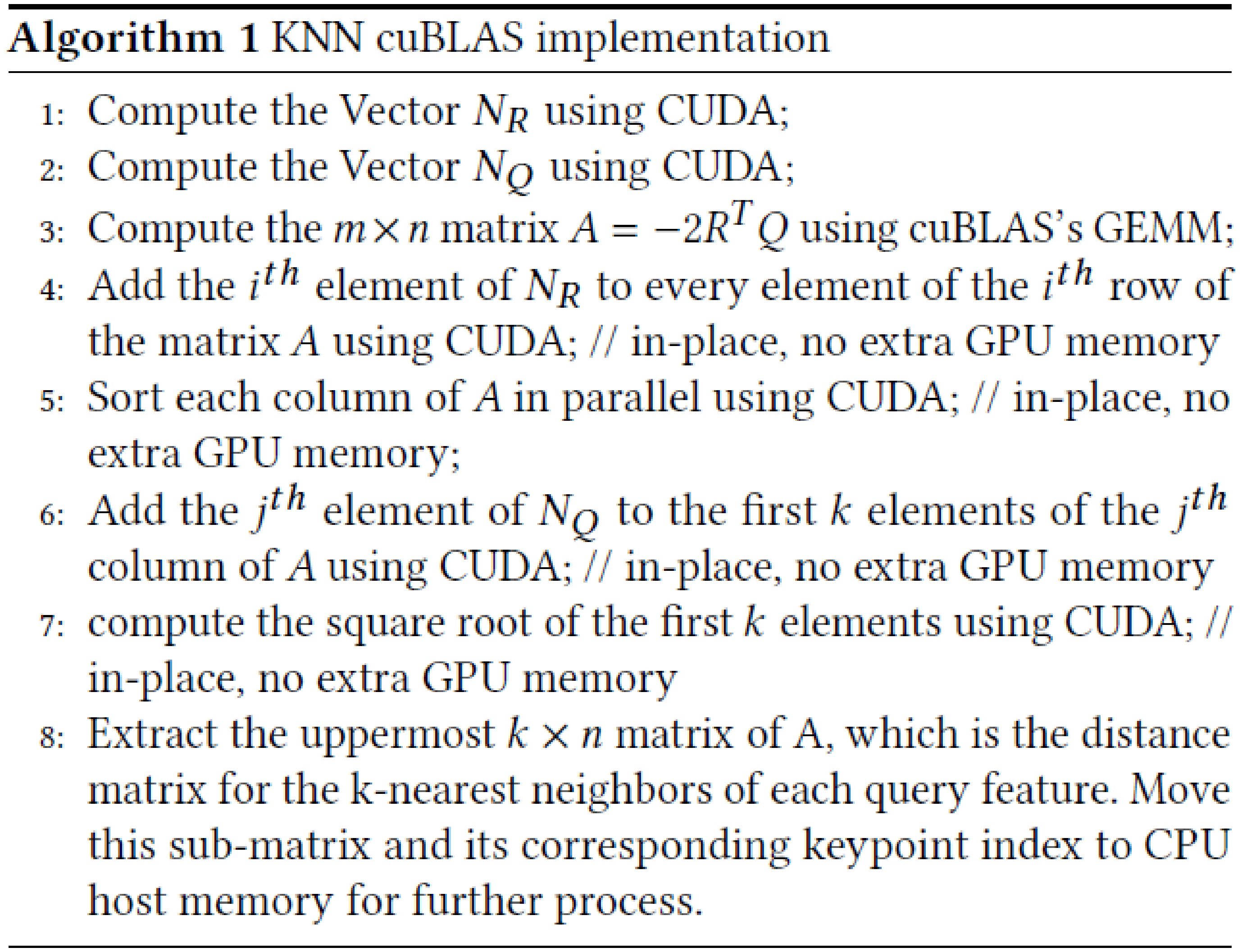
The core computation is two vectors’ Euclidean distance

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

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

The core of cuBLAS is GEMM:

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



Algorithm 1 KNN cuBLAS implementation

```
Compute the Vector $N_R$ using CUDA;
Compute the Vector $N_Q$ using CUDA;
Compute the $m \times n$ matrix $A = -2R^T Q$ using cuBLAS's GEMM;
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
Sort each column of $A$ in parallel using CUDA; // in-place, no
    extra GPU memory;
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
compute the square root of the first $k$ elements using CUDA; //
    in-place, no extra GPU memory
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.
```

Half Precision (FP16):

- 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.

Vincent Garcia, Eric Debreuve, Frank Nielsen, Michel Barlaud, K-NEAREST NEIGHBOR SEARCH: FAST GPU-BASED IMPLEMENTATIONS AND APPLICATION TO HIGH-DIMENSIONAL FEATURE MATCHING

Performance of cuBLAS implementation

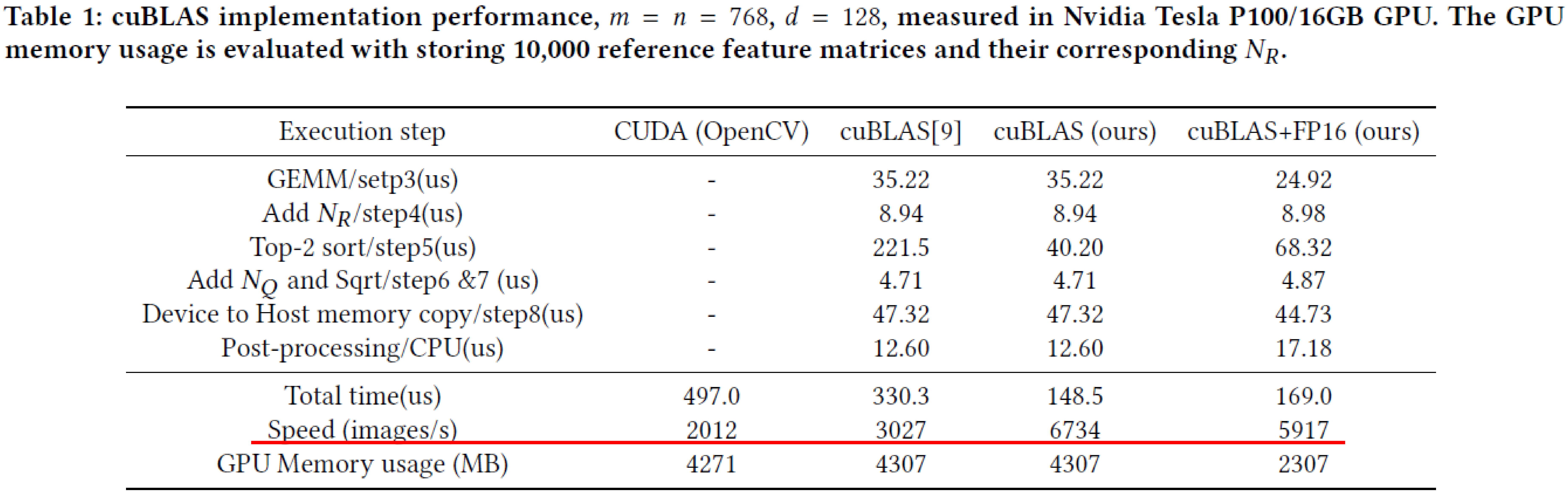


Table 1: cuBLAS implementation performance, $m = n = 768$, $d = 128$, measured in Nvidia Tesla P100/16GB GPU. The GPU memory usage is evaluated with storing 10,000 reference feature matrices and their corresponding N_R.

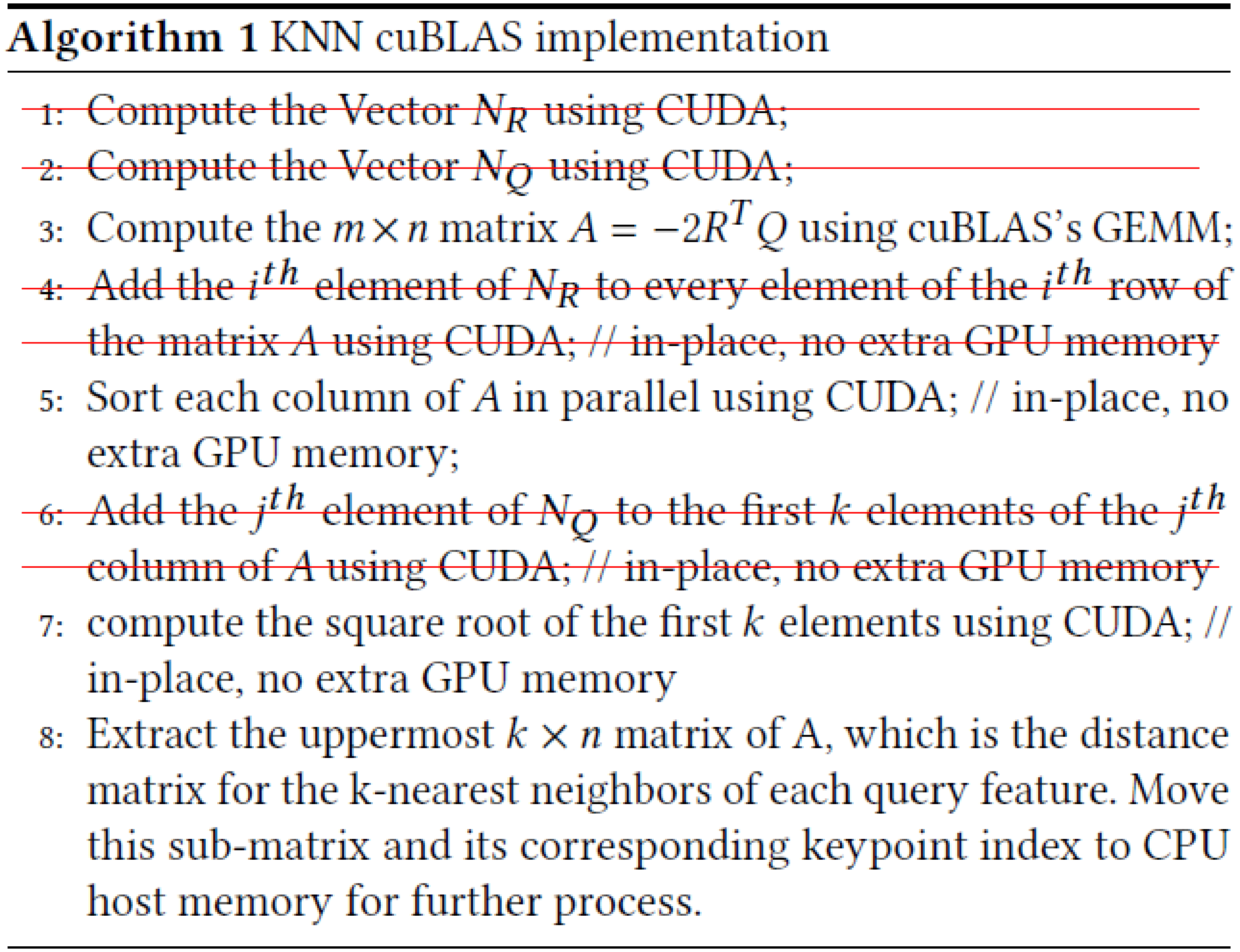
Execution step	CUDA (OpenCV)	cuBLAS[9]	cuBLAS (ours)	cuBLAS+FP16 (ours)
GEMM/setp3(us)	-	35.22	35.22	24.92
Add N_R/step4(us)	-	8.94	8.94	8.98
Top-2 sort/step5(us)	-	221.5	40.20	68.32
Add N_Q and Sqrt/step6 &7 (us)	-	4.71	4.71	4.87
Device to Host memory copy/step8(us)	-	47.32	47.32	44.73
Post-processing/CPU(us)	-	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

- cuBLAS + FP32, the speed is 6,734 images/s with 3.34x speedup comparing to OpenCV baseline.
- cuBLAS + FP16, the speed is 5,915 images/s with 2.94x 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 12.1% 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.

Can the NN cuBLAS approach be simplified?

🙂 SIFT Feature Descriptor Normalization



Algorithm 1 KNN cuBLAS implementation

```
~~Compute the Vector $N_R$ using CUDA;~~
~~Compute the Vector $N_Q$ using CUDA;~~
Compute the $m \times n$ matrix $A = -2R^T Q$ using cuBLAS's GEMM;
~~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~~
Sort each column of $A$ in parallel using CUDA; // in-place, no extra GPU memory;
~~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~~
compute the square root of the first $k$ elements using CUDA; // in-place, no extra GPU memory
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.
```

🙁 Directly normalize the SIFT descriptor will hurt the matching accuracy

Descriptor Normalization

Introduce the RootSIFT(*) implementation.

a) L_1 normalize the SIFT vector;

b) square root each element.

RootSIFT descriptor is L_2 normalized

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

<u>Hellinger kernel</u>: for two L_1 normalized histograms, x and y:

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

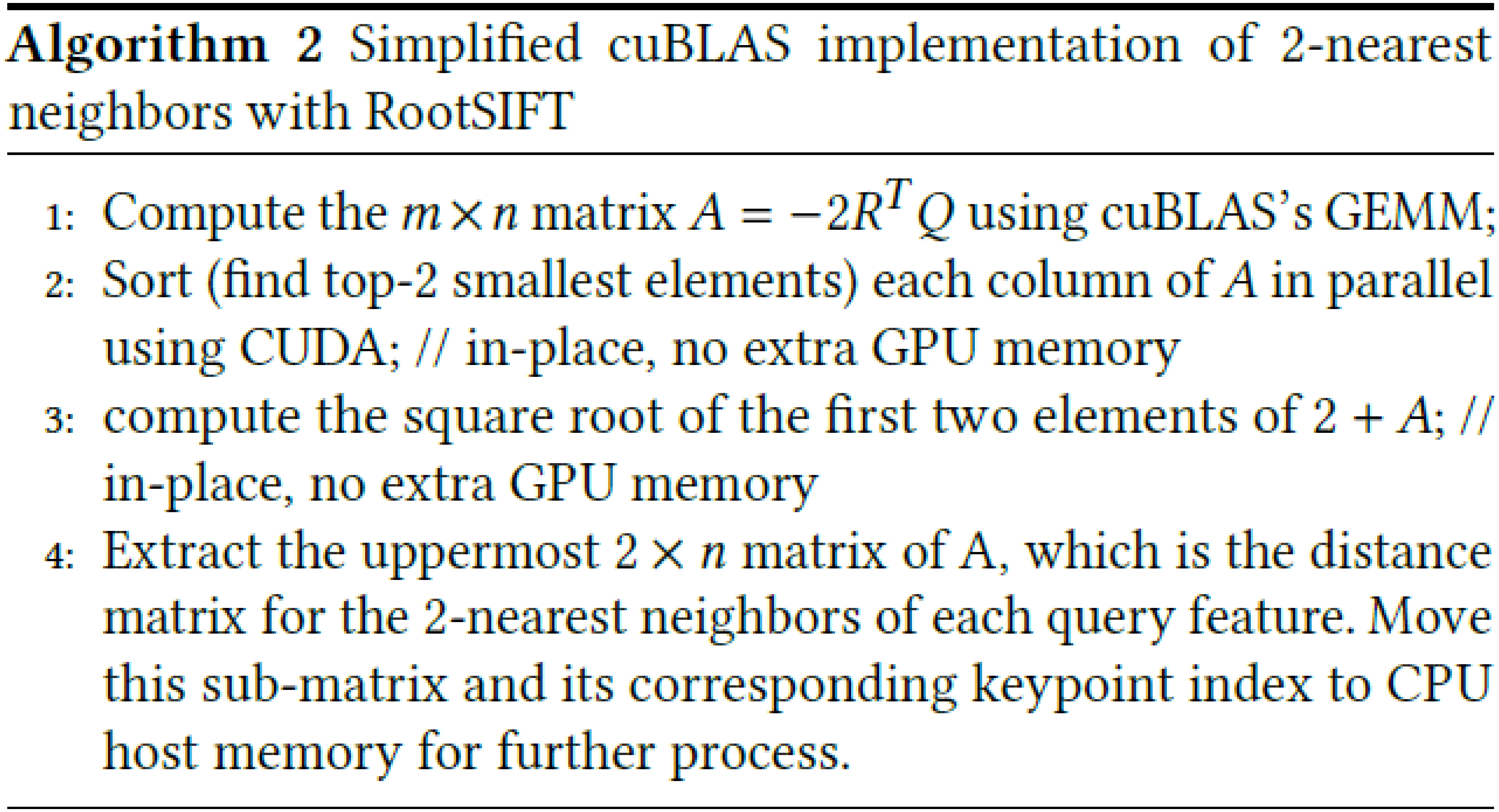
Hellinger distance is used to quantify the similarity between two probability distributions

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

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

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

Simplified Algorithm



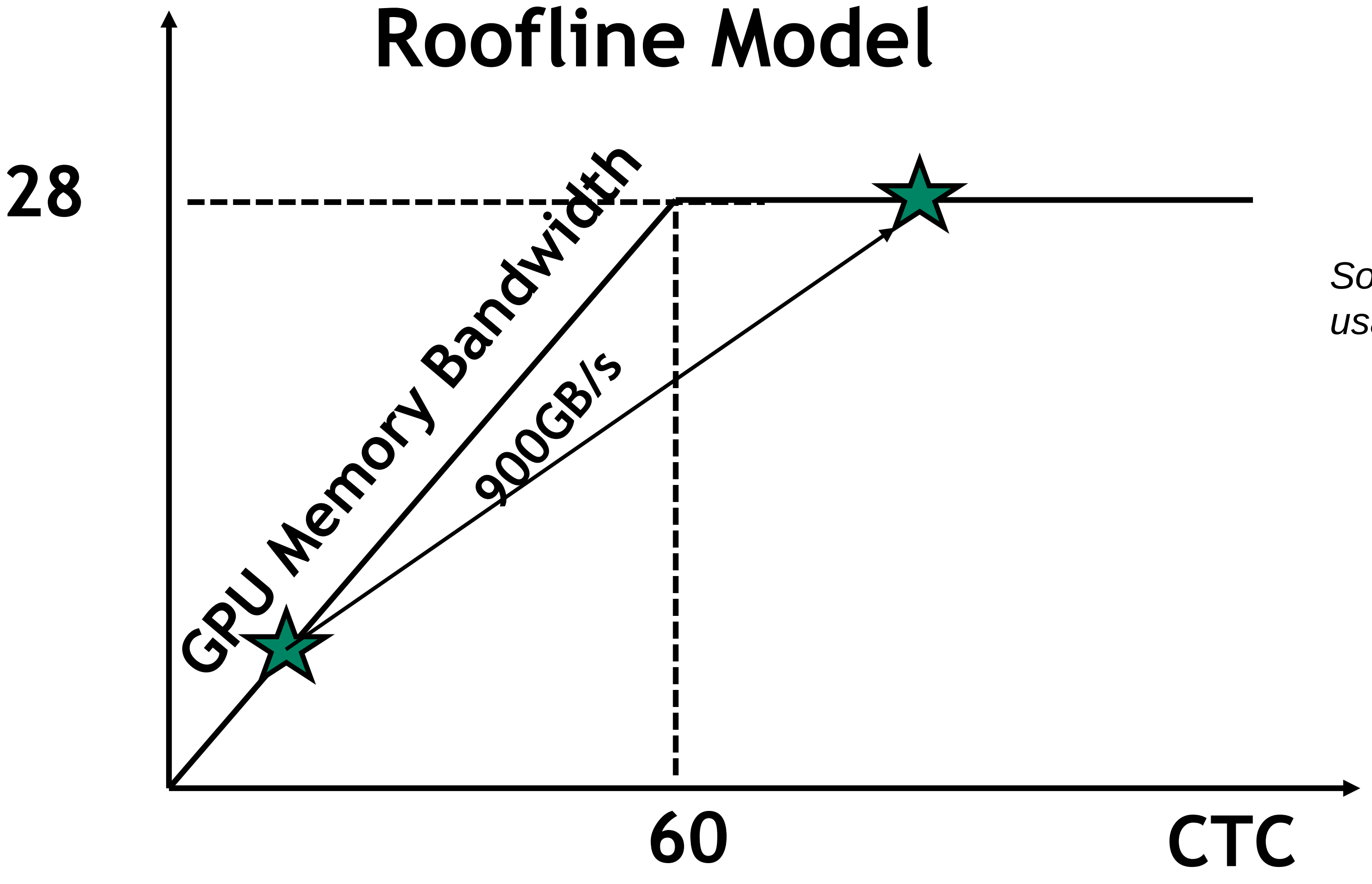
Algorithm 2 Simplified cuBLAS implementation of 2-nearest neighbors with RootSIFT

```
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.
```

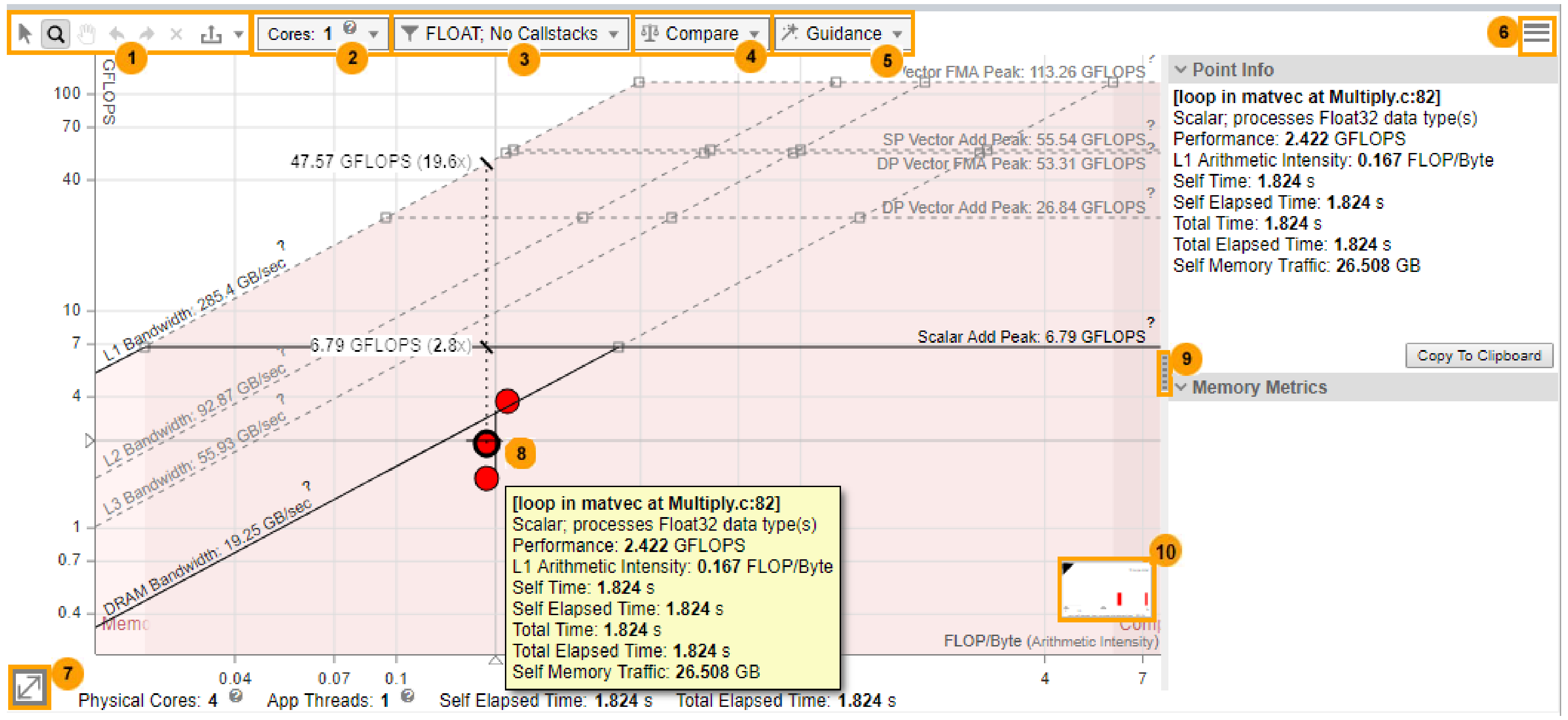
The step 3 2+**A** and square root could be performed in place directly after the step 2 to minimize the data movement.

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 matric is two small. The matrix is only 768*128.



Increasing CTC by Batching

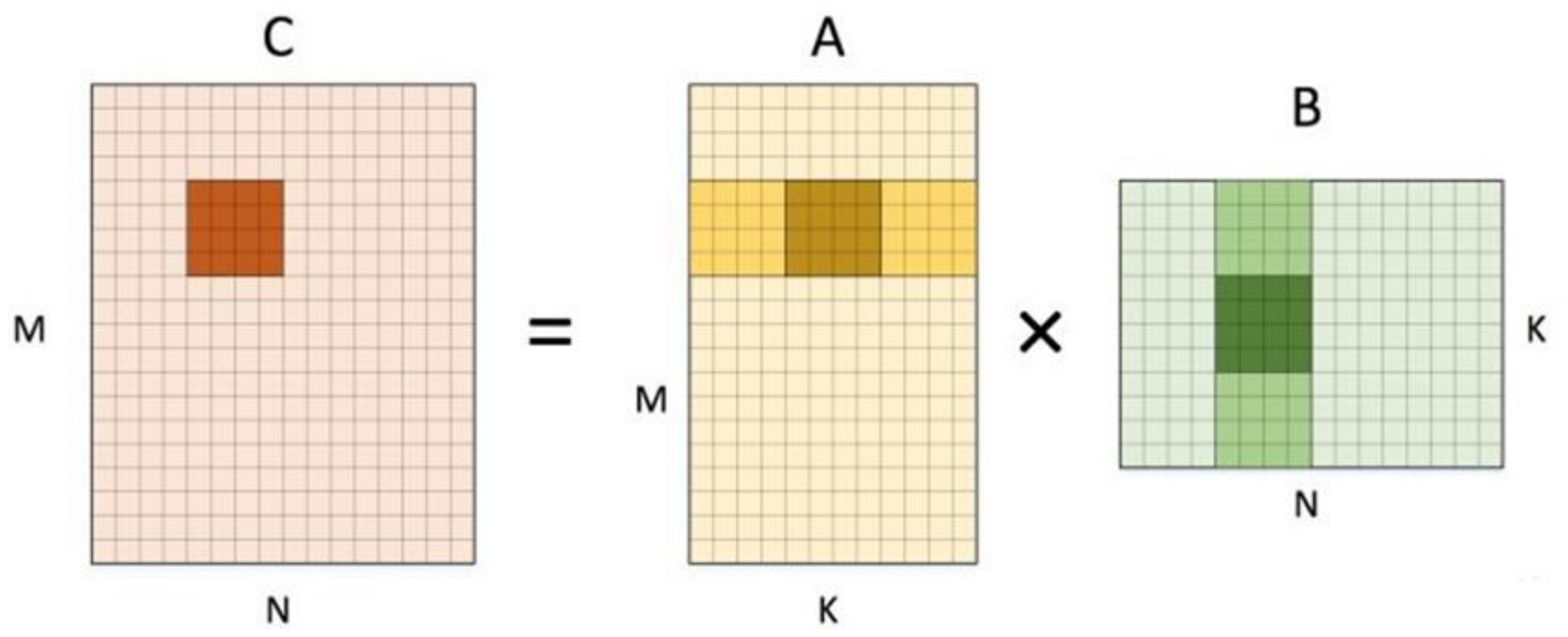

Cores: 1
FLOAT; No Callstacks
Compare
Guidance
1
2
3
4
5
6
7
8
9
10
GFLOPS
100
70
40
10
7
4
1
0.7
0.4
Vector FMA Peak: 113.26 GFLOPS
SP Vector Add Peak: 55.54 GFLOPS
DP Vector FMA Peak: 53.31 GFLOPS
DP Vector Add Peak: 26.84 GFLOPS
Scalar Add Peak: 6.79 GFLOPS
47.57 GFLOPS (19.6x)
6.79 GFLOPS (2.8x)
L1 Bandwidth: 285.4 GB/sec
L2 Bandwidth: 92.87 GB/sec
L3 Bandwidth: 55.93 GB/sec
DRAM Bandwidth: 19.25 GB/sec
Memo
Com
FLOP/Byte (Arithmetic Intensity)
0.04
0.07
0.1
4
7
Point Info
Copy To Clipboard
Memory Metrics

[loop in matvec at Multiply.c:82]
Scalar; processes Float32 data type(s)
Performance: **2.422** GFLOPS
L1 Arithmetic Intensity: **0.167** FLOP/Byte
Self Time: **1.824** s
Self Elapsed Time: **1.824** s
Total Time: **1.824** s
Total Elapsed Time: **1.824** s
Self Memory Traffic: **26.508** GB

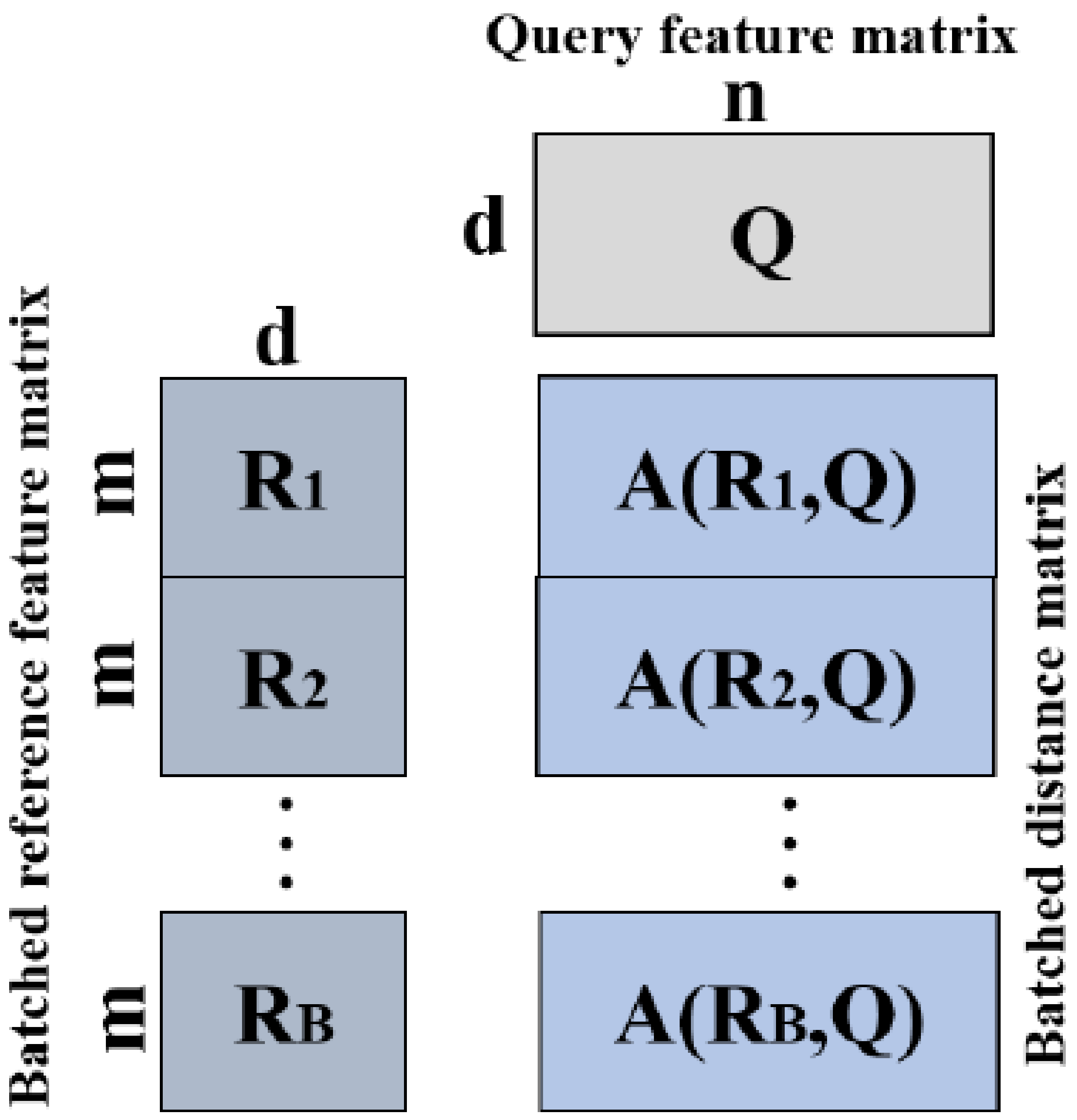
[loop in matvec at Multiply.c:82]
Scalar; processes Float32 data type(s)
Performance: **2.422** GFLOPS
L1 Arithmetic Intensity: **0.167** FLOP/Byte
Self Time: **1.824** s
Self Elapsed Time: **1.824** s
Total Time: **1.824** s
Total Elapsed Time: **1.824** s
Self Memory Traffic: **26.508** GB

Physical Cores: 4   App Threads: 1   Self Elapsed Time: **1.824** s   Total Elapsed Time: **1.824** s

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

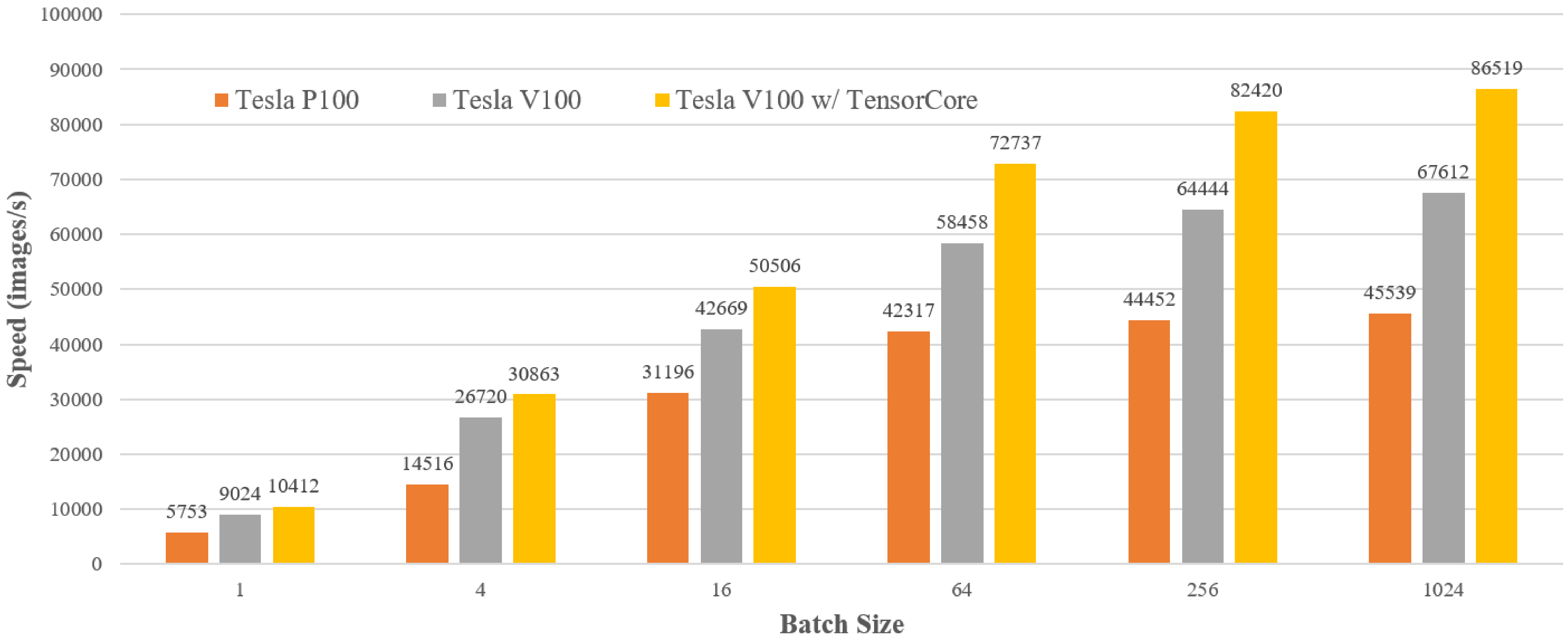


Optimization 2: Batching the Reference Descriptors



- 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



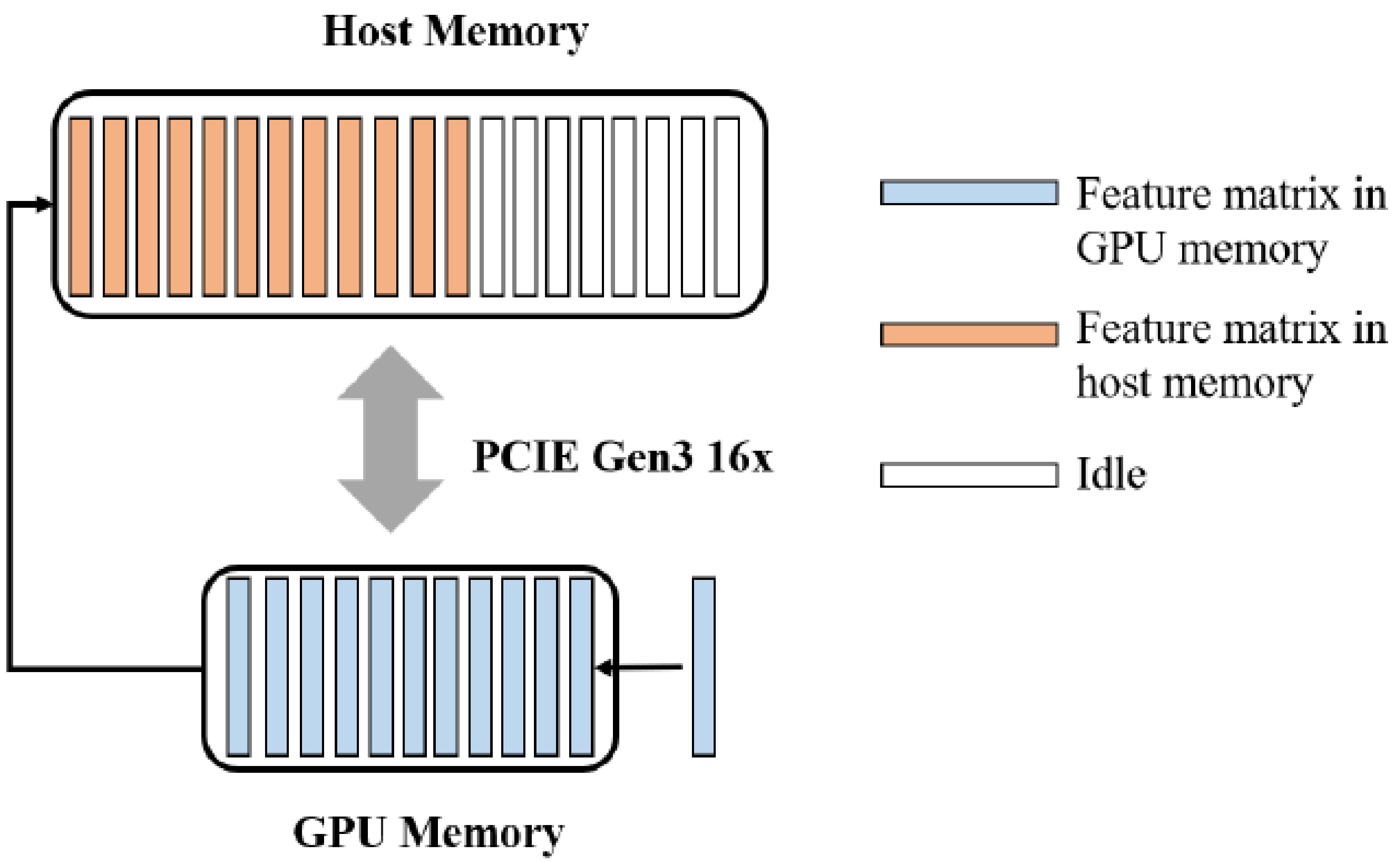
- With batch size=1024, P100 GPU, the speed is improved from 5,753 images/s to 45,539 images/s with a significant speedup of 7.9x.
- With batch size=1024, V100 GPU, the speed is improved from 9,024 images/s to 67,612 images/s with a significant speedup of 7.5x
- w/ Tensor Core in V100 GPU, peak performance is 86,519 images/s which is an additional 1.3x speedup

Optimization 3: Hybrid Caching

Even with half precision, each image's feature descriptors need ~0.2MB GPU memory, and single 16GB 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 decreased from 45,539 to 25,362 images/s (dropped 43.9%) 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

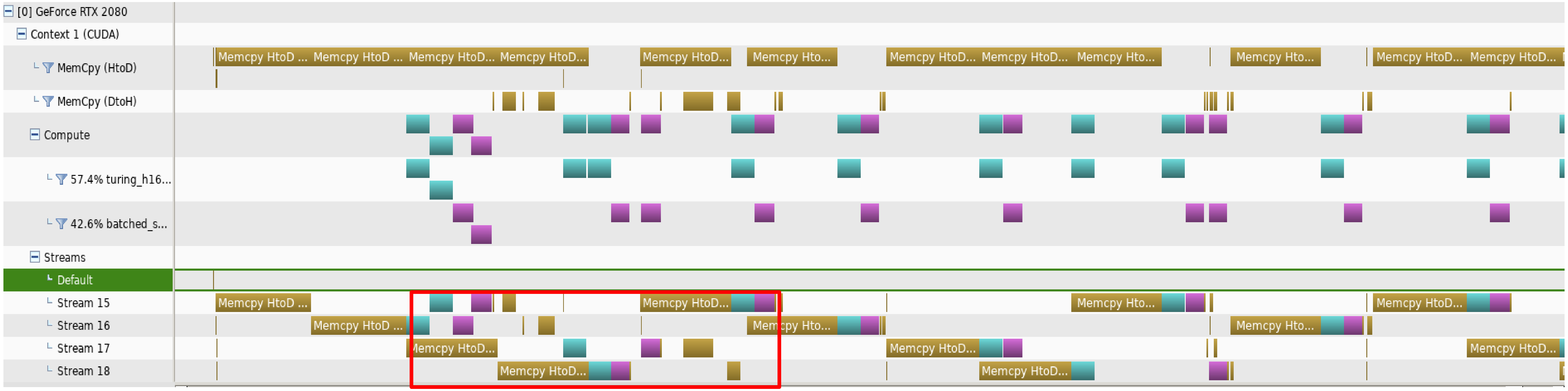


INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

acm In-Cooperation sighpc

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

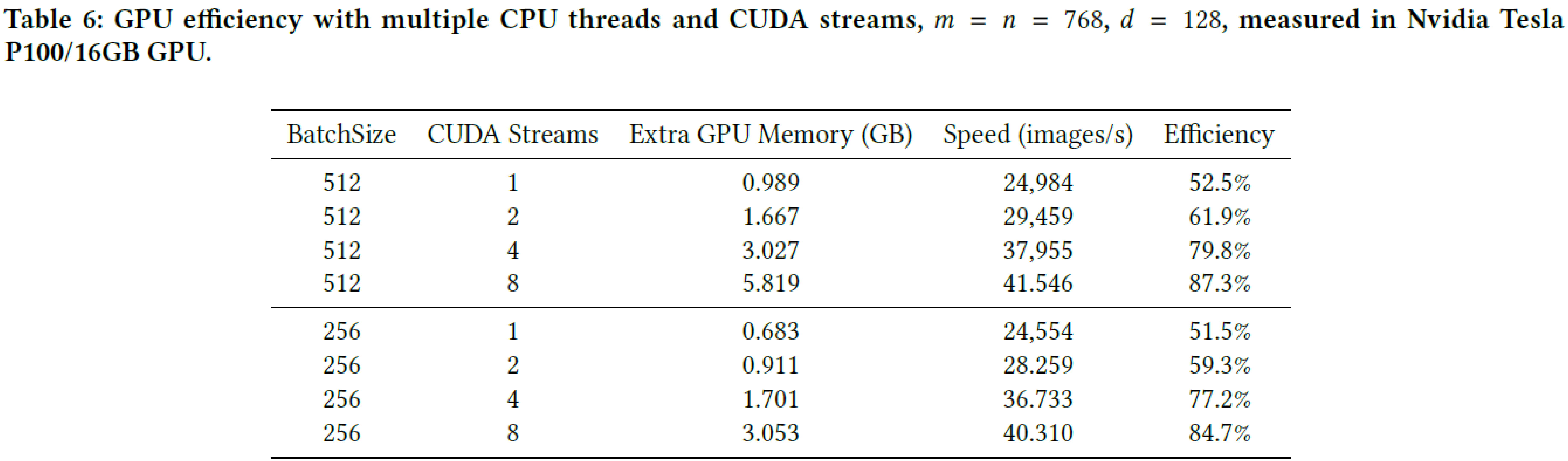


Table 6: GPU efficiency with multiple CPU threads and CUDA streams, $m = n = 768$, $d = 128$, measured in Nvidia Tesla P100/16GB GPU.

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%

- 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.
- More extra GPU memory is used since each stream needs a dedicated GPU memory to store some temporary intermediate data.

Using multiple CUDA streams can significantly improve the speed & efficiency.

Asymmetric Local Feature Extraction

- Reference features are only used for ratio tests to distinguish distinct features from non-distinct features in the query texture image.
- Slightly reducing feature number of reference texture images will not significantly affect the search accuracy.

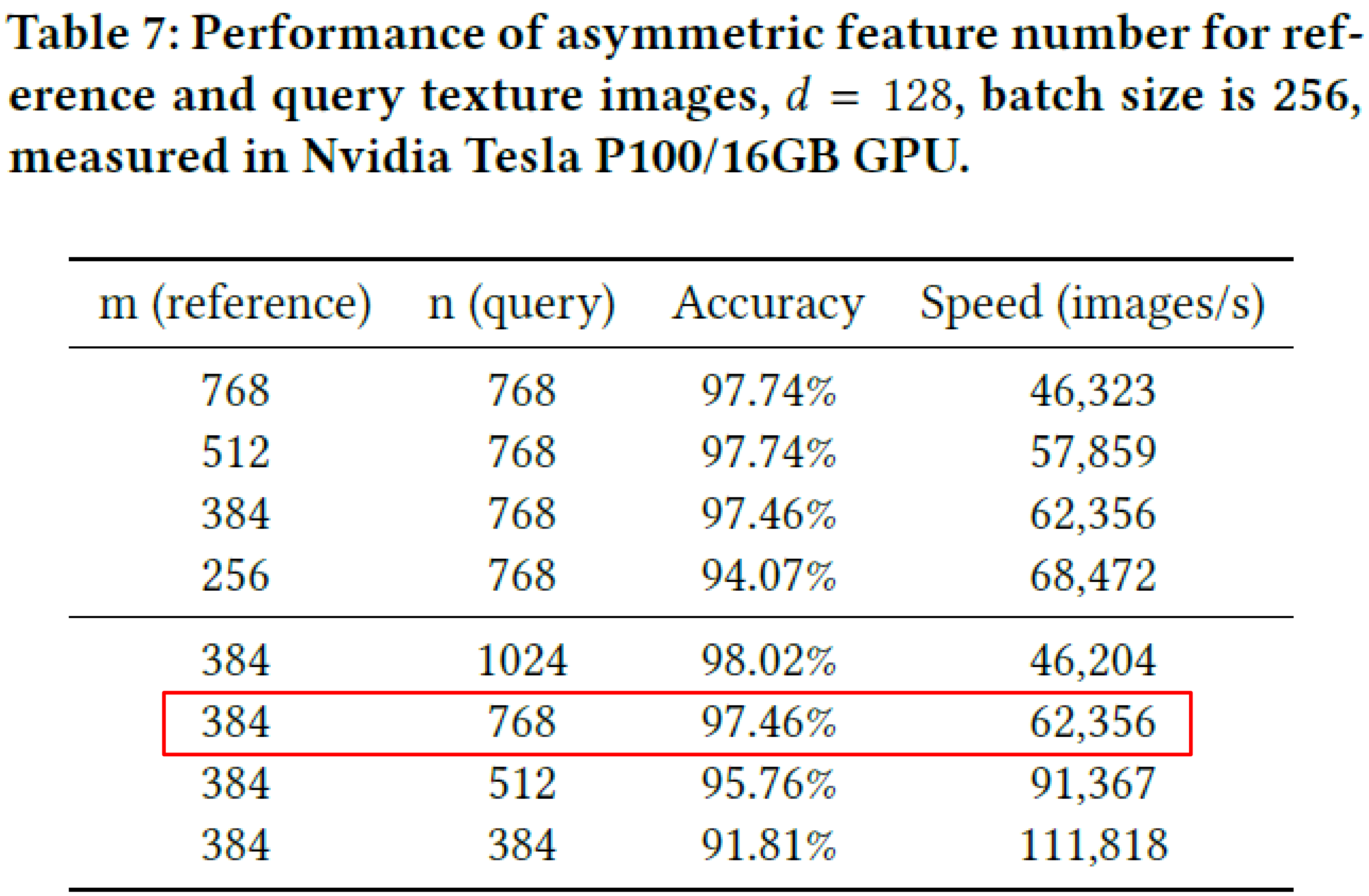


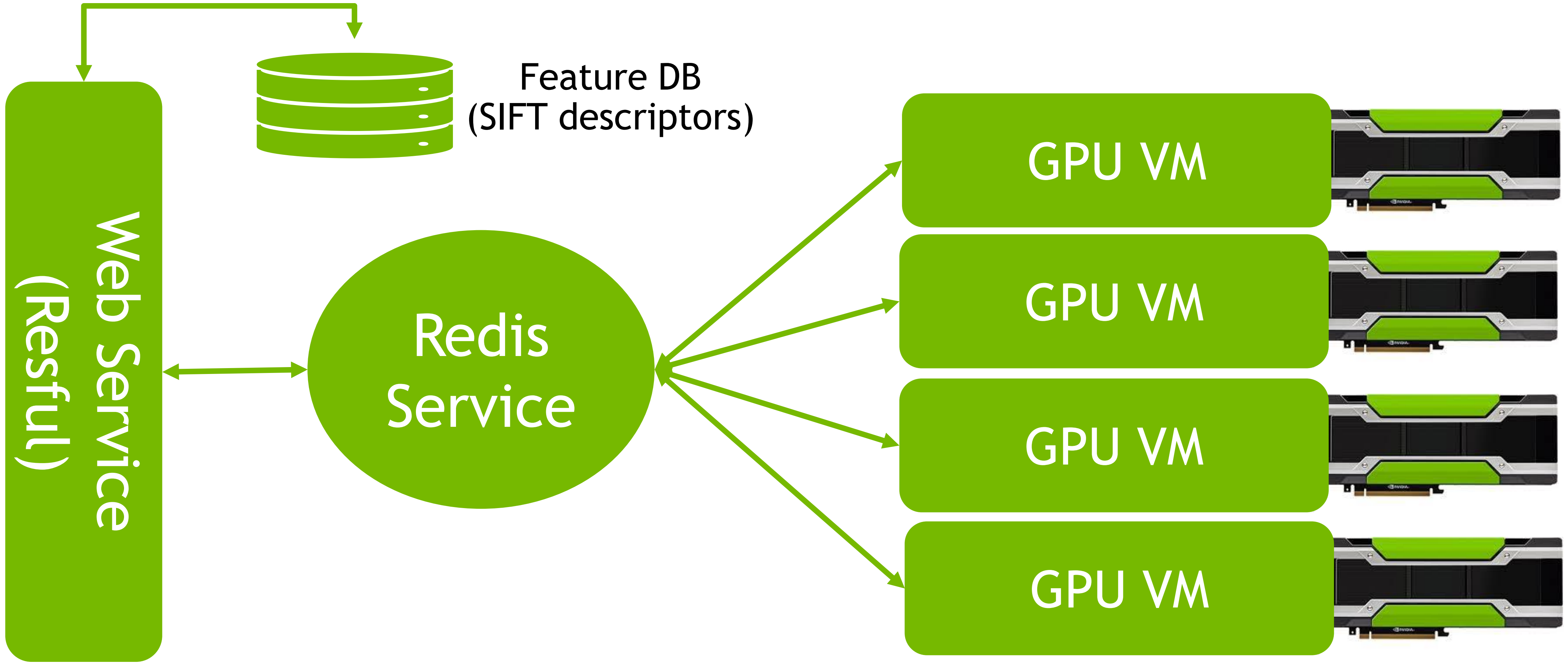
Table 7: Performance of asymmetric feature number for reference and query texture images, $d = 128$, batch size is 256, measured in Nvidia Tesla P100/16GB GPU.

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

- We can observe that the optimal solution is m=384, n=768.
- The accuracy loss is very limited (only 0.28%).
- The speed is improved by 34.6% compared to the baseline with m = 768, n = 768.

Distributed Texture Search System

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 10.8 millions texture images, and the speed is 872,984 images/second.

Thanks