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

Accelerating DBSCAN Algorithm with AI Chips for Large Datasets

Zhuoran Ji and Cho-Li Wang

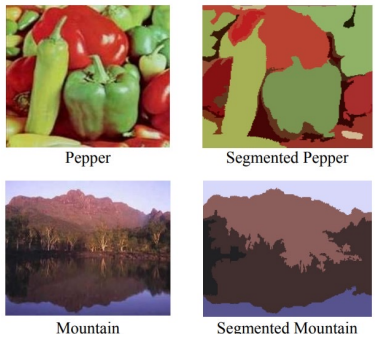
Department of Computer Science

The University of Hong Kong

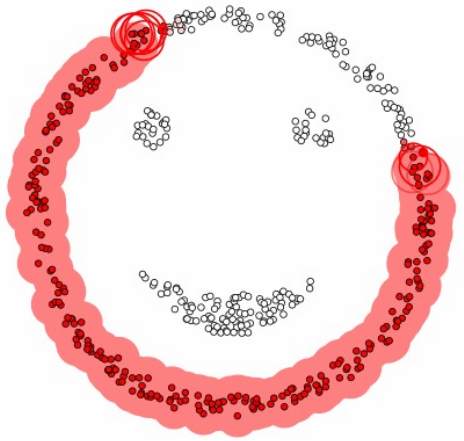
August 11, 2021

DBSCAN

- DBSCAN is a powerful clustering areas
  - E.g., text clustering, astronomy, geography, image processing
- Advantage:
  - No need to specify the # of clusters to be found
  - Can detect cluster with any shape
- But: high computational complexity
  - distance between every two points

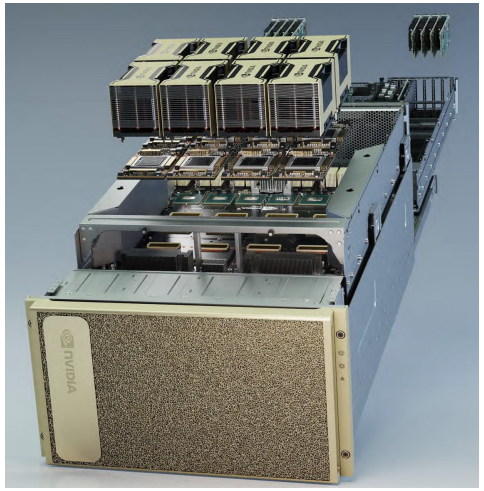


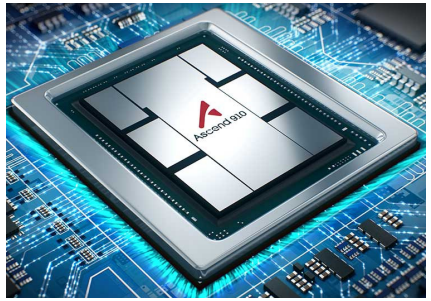
images from google

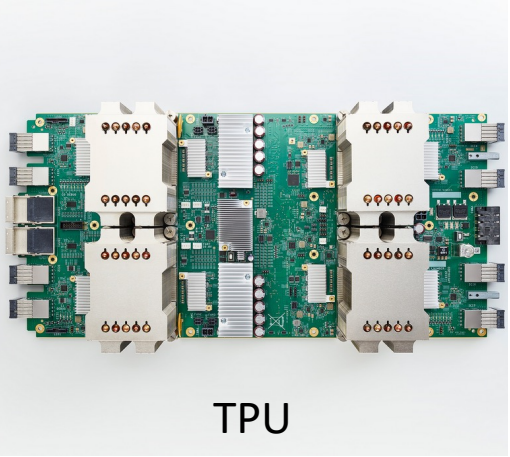


AI Chips

- Driven by DL, many AI chips are proposed
  - Nvidia GPU, Huawei Ascend, Alibaba Hanguang, Google TPU
- **Their marquee feature is TCUs (Tensor Core Units): dedicated matrix multiplication hardware units**
- Provide very efficient MM algorithm
  - RTX3090: 71 vs 35.6 TFLOPS
- However, other components, such as vector units and scalar units are usually
  - less powerful
  - highly specialized for AI apps







TPU

Motivation

- Distance metric for DBSCAN:
  - cosine similarity, dot product, Euclidean distance
- Cosine similarity:

$$d_{[i,j]} = \frac{\mathbf{P}_i \cdot \mathbf{P}_j}{\|\mathbf{P}_i\|\|\mathbf{P}_j\|} = \frac{\sum_{k=1}^{dim} \mathbf{P}_i[k]\mathbf{P}_j[k]}{\sqrt{\sum_{k=1}^{dim} \mathbf{P}_i[k]^2}\sqrt{\sum_{k=1}^{dim} \mathbf{P}_j[k]^2}} \xrightarrow{l2 \text{ normalized}} d_{[i,j]} = \frac{\sum_{k=1}^{dim} \mathbf{P}_i[k]\mathbf{P}_j[k]}{1 \times 1} = \mathbf{P}_i \cdot \mathbf{P}_j$$

- Distance matrix:

$$\begin{vmatrix} d_{[\mathcal{F}_0,0]} & \cdots & d_{[\mathcal{F}_0,n]} \\ \vdots & \ddots & \vdots \\ d_{[\mathcal{F}_m,0]} & \cdots & d_{[\mathcal{F}_m,n]} \end{vmatrix} = \begin{vmatrix} P_{\mathcal{F}_0} \\ \vdots \\ P_{\mathcal{F}_m} \end{vmatrix} \times \begin{vmatrix} P_0^T & \cdots & P_n^T \end{vmatrix}$$

- Natural to use TCUs to accelerate distance calculation

Challenge: Identify ε-neighbors on AI chips

- However, identify and count ε-neighbors on AI chips is challenging
  - involve compare-and-select (CMPSEL) operation
  - weakly supported by many AI chips

```
procedure IDENTIFY_NEIGHBORS(dis_matrix)
    adj[i][j] = dis_matrix[i][j] ≤ ε ? 1 : 0
    numNeighbors[i] = REDUCE(+, adj[i][0:j])
    isCore[i] = numNeighbors[i] ≥ minPts ? true : false
    isNeighbor[j] = REDUCE(||, adj[0:i][j], isCore[i])
    return isNeighbor[0:j]
end procedure
```

- Some AI chips do have dedicated ReLU units
  - y = x > 0 ? x : 1
  - BUT, not allow an arbitrary value for the positive input

Challenge: Device Memory is Limited

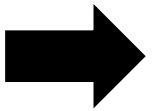
- GPUs and other AI chips' device memory: 10x GBs
- The size of the datasets can be 100x GBs, exceeding the capacity
- Space complexity
  - Dataset: O(DN)
  - Adjacency matrix: O(MN)
  - N is the # of points in the dataset

Identify ε-neighbors on AI chips

- The essence of ε-neighbor identification is to find a mapping function so that
  - # of ε-neighbors <- $REDUCE\big(op_1, MAP(f, distance[i][0{:}j])\big)$
  - whether j is an ε-neighbors <- $REDUCE(op_2, MAP(f, distance[0{:}i][j]))$

- We can then generalize the ε-neighbors identification kernel as

procedure IDENTIFY_NEIGHBORS(*dis_matrix*)
  adj[i][j] = *dis_matrix*[i][j] ≤ ε ? 1 : 0
  numNeighbors[i] = REDUCE(+, *adj*[i][0:j])
  isCore[i] = *numNeighbors*[i] ≥ *minPts* ? true : false
  isNeighbor[j] = REDUCE(||, *adj*[0:i][j], *isCore*[i])
  return *isNeighbor*[0:j]



indicators[i][j] = ***f***(*distance*[i][j], ε)
internalNum[i] = REDUCE(***op*₁**, *indicators*[i][0:j])
isCore[i] = **CHECK_CORE_POINT**(*internalNum*[i], *minPts*)
isNeighbor[j] = REDUCE(***op*₂**, *indicators*[0:i][j], *isCore*[i])
return *isNeighbor*[0:j]

Mapping Function for AI Chips

- The key of the mapping function is to
  - differentiate the behaviors of ε-neighbors and non-ε-neighbors during reduction
  - unify the behavior of the points within the same category
- For example, the unit step function maps
  - ε-neighbors to 1
  - non-ε-neighbors to 0
  - satisfies the two requirement

Mapping Function $f_{-2.0/\pm 0.0}$

- $f_{-2.0/\pm 0.0} = ((\text{distance} - \varepsilon)\ \&\ 0\text{x}8000 \mid 0\text{x}3c00) - 1.0$
- In the form of instructions:

Algorithm 3 Mapping function $f_{-2.0/\pm 0.0}$

1: v_1 = *distance* - ε ▷ ε - *similarity*, if measured as similarity
2: v_2 = reinterpret v_1 as a 16-bits binary
3: $v_3 = v_2$ & 0x8000 ▷ ε-neighbors = 0x8000, others = 0x0000
4: $v_4 = v_3 \mid$ 0x3c00 ▷ ε-neighbors = 0xbc00, others = 0x3c00
5: v_5 = reinterpret v_4 as a *float16*
6: *indicator* = v_5 - 1.0

1. differentiate the behaviors of ε-neighbors and non-ε-neighbors during reduction
2. unify the behavior of the points within the same category

1. +0.0 and −0 ar behaviors for addition a action
2. 0.0 is specia y and division

ε-neighbor: ε – 2.0	–ε	– 2.0	reinterpret as 16-bit bin	0xC000	& 0x8000 → 0x8000 (-0.0)
non-ε-neighbor: ε + 2.0		2.0		0x4000	& 0x8000 → 0x0000 (0.0)

Mapping Function $f_{-2.0/\pm 0.0}$

- $f_{-2.0/\pm 0.0} = ((\text{distance} - \varepsilon)\ \&\ 0\text{x}8000 \mid 0\text{x}3\text{c}00) - 1.0$
- In the form of instructions:

Algorithm 3 Mapping function $f_{-2.0/\pm 0.0}$

1: v_1 = *distance* - ε ▷ ε - *similarity*, if measured as similarity
2: v_2 = reinterpret v_1 as a 16-bits binary
3: v_3 = v_2 & 0x8000 ▷ ε-neighbors = 0x8000, others = 0x0000
4: v_4 = v_3 | 0x3c00 ▷ ε-neighbors = 0xbc00, others = 0x3c00
5: v_5 = reinterpret v_4 as a *float16*
6: *indicator* = v_5 - 1.0

ε-neighbor: ε – 2.0	–ε	– 2.0	reinterpret as 16-bit bin	0xC000	& 0x8000	0x8000 (-0.0)	| 0x3c00	0xbc00 (-1.0)	-2.0
non-ε-neighbor: ε + 2.0		2.0		0x4000		0x0000 (0.0)		0x3c00 (1.0)	0.0

Mapping Function $f_{-128/0}$

- $f_{-128/0} = (\text{distance} - \varepsilon)\ \&\ 0\text{x}8000$
- Same with the first three steps of $f_{-2.0/\pm 0.0}$
- In $f_{-2.0/\pm 0.0}$
  - +0.0 and −0.0 have similar behaviors for addition and subtraction
  - 0.0 is special for multiply and division
- How about we interpret it as int8

ε-neighbor: ε – 2.0	$-\varepsilon$	– 2.0	reinterpret as 16-bit bin	0xC000	$\&\ 0\text{x}8000$	0x8000 (-0.0)	reinterpret as 2 int8	(-128, 0)
non-ε-neighbor: ε + 2.0		2.0		0x4000		0x0000 (0.0)		(0, 0)

Mapping Function $f_{-128/0}$

- Easy to determine whether a point is a ε-neighbor
  - i.e., $op_2 = ||$
- However, cannot count ε-neighbors by integer addition
- The value "−128" is the maximum negative value for int8
- Vector units usually need the inputs and the outputs to have the same data type
  - Overflow almost always occurs
- Two intuitive method
  - Convert the data type to int32
    - e.g., static_cast
    - Casting is costly, reducing int32 is expensive than reducing int16
  - Reduce the magnitude by vector operation
    - e.g., right shift, integer division, or exponential function
    - weakly supported or costly than $f_{-2.0/\pm 0.0}$

Mapping Function $f_{-128/0}$

- Mixed-precision MMA
  - One of the most novel features of TCUs
- Input matrices in low-precision data types
- Accumulates the result in high-precision data types
- E.g.,
  - Nvidia TCUs support FP16 → FP32 and INT4/8 → INT32
  - Huawei TCUs support FP16 → FP32 and INT8 → INT32
- Thus, we use TCUs to accumulate the mapped values

Mini Batch DBSCAN

- DBSCAN can process the data in a mini-batch manner
  - much like the deep neural network training
  - but high data transfer overhead

```
1:  for batch_i in # of mini-batches do
2:      transfer dataset[batch_i] to device mem
3:      calculate the distance matrix disMat[batch_i]
4:      identify ε-neighbors adjMat[batch_i]
5:      accumulate the # of ε-neighbors of adjMat[batch_i]
6:      transfer adjMat[batch_i] to host mem
7:  determine corePoints with # of ε-neighbors
8:  for batch_i in # of mini-batches do
9:      transfer adjMat[batch_i] to device mem
10:     merge ε-neighbors of corePoints to isNeighbor[batch_i]
11:     transfer isNeighbor[batch_i] to host mem
```

Mini Batch DBSCAN: Ping Pong Buffer

- Ping-Pong buffers can overlap- ping computation and data transfer
  - time should be comparable
  - otherwise, overlap is negligible.

- Distance matrix calculation
  - computation & data transferring time is comparable
- BUT, for merging ε-neighbors
  - overlap is negligible

```
1:  for batch_i in # of mini-batches do
2:      transfer dataset[batch_i] to device mem        O(DN)
3:      calculate the distance matrix disMat[batch_i]  O(MDN)
4:      identify ε-neighbors adjMat[batch_i]
5:      accumulate the # of ε-neighbors of adjMat[batch_i]
6:      transfer adjMat[batch_i] to host mem           O(NM)
7:  determine corePoints with # of ε-neighbors
8:  for batch_i in # of mini-batches do
9:      transfer adjMat[batch_i] to device mem         O(MN)
10:     merge ε-neighbors of corePoints to isNeighbor[batch_i]   O(NM)
11:     transfer isNeighbor[batch_i] to host mem
```

Speculative Execution

- The control flow depends on # of neighbors: calculated with a loop over mini-batches

```
1:  for batch_i in # of mini-batches do
2:      transfer dataset[batch_i] to device mem
3:      calculate the distance matrix disMat[batch_i]
4:      identify ε-neighbors adjMat[batch_i]
5:      accumulate the # of ε-neighbors of adjMat[batch_i]
6:      transfer adjMat[batch_i] to host mem
7:  determine corePoints with # of ε-neighbors
8:  for batch_i in # of mini-batches do
9:      transfer adjMat[batch_i] to device mem
10:     merge ε-neighbors of corePoints to isNeighbor[batch_i]
11:     transfer isNeighbor[batch_i] to host mem
```

Control Dependency

Speculative Execution

- The control flow depends on # of neighbors: calculated with a loop over mini-batches
- kernels are executed in two loops, even if deal with the same piece of data

```
1:  for batch_i in # of mini-batches do
2:      transfer dataset[batch_i] to device mem
3:      calculate the distance matrix disMat[batch_i]
4:      identify ε-neighbors adjMat[batch_i]
5:      accumulate the # of ε-neighbors of adjMat[batch_i]
6:      transfer adjMat[batch_i] to host mem
7:  determine corePoints with # of ε-neighbors
8:  for batch_i in # of mini-batches do
9:      transfer adjMat[batch_i] to device mem
10:     merge ε-neighbors of corePoints to isNeighbor[batch_i]
11:     transfer isNeighbor[batch_i] to host mem
```

Speculative Execution

- The neighborhood relation is usually symmetric
  - a point has many ε-neighbors, its ε-neighbors usually also have many ε-neighbors
- Based on this, we propose a speculative merging strategy
- Aggressively assumes all processed frontiers are core points

  -> speculatively merges their ε-neighbors in the first loop

```
1:  for batch_i in # of mini-batches do
2:      transfer dataset[batch_i] to device mem
3:      calculate the distance matrix disMat[batch_i]
4:      identify ε-neighbors adjMat[batch_i]
5:      accumulate the # of ε-neighbors of adjMat[batch_i]
10:     merge ε-neighbors of frontiers to isNeighbor[batch_i]
11:     transfer isNeighbor[batch_i] to host mem
```

Speculative Execution

- The neighborhood relation is usually symmetric
  - a point has many ε-neighbors, its ε-neighbors usually also have many ε-neighbors
- Based on this, we propose a speculative merging strategy
- Aggressively assumes all processed frontiers are core points

  -> speculatively merges their ε-neighbors in the first loop
- A correct one: avoids transferring the adjacency matrix
- An incorrect one: incurs $(M-1)\times N$ bitwise operation
  - arithmetic is much cheaper than PCIe data transferring

Speculative Execution

- Another bottleneck is initializing clusters
- Computes the distance **vector** between the potential seed (a single point) and the whole dataset
  - transfer the whole dataset to device mem
  - use TCU to compute matrix-vector multiplication
    - APIs impose limitation, at least $16 \times 16 \times 16$
- Low arithmetic intensity, low TCU utilization
- Thus, speculatively explore the initial points when constructing new cluster(s)

Speculative Execution

- Speculatively explores many unvisited points rather than one
- A call to `explore_points` is reduced if
  - one of these points is not a core point
  - any two belong to different clusters
- More formally
  - x non-core points and y different clusters reduce $x + y - 1$ times calls to `explore_points`
- How many points are explored speculatively?
  - explore 16 points incurs no extra cost, but poor tiling strategy
  - thus, explore as many as possible until exhausting on-chip memory

Evaluation

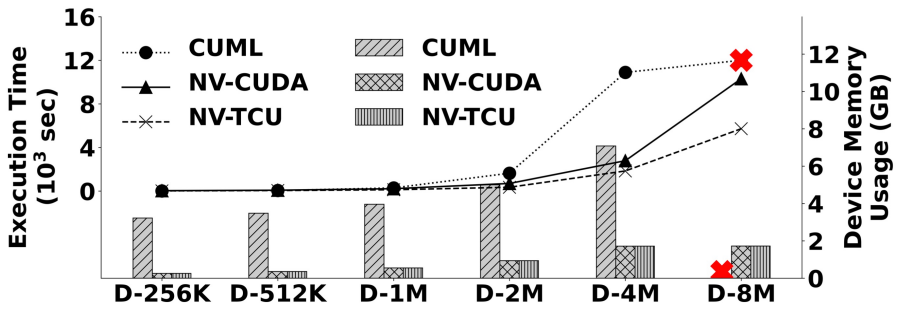
- Evaluate on both Nvidia GPUs and Huawei Ascend 310

Device	Architecture	TCU Size	TCU throughput	Memory Type	Memory Size	Programming Language
Nvidia RTX 2080ti [34]	Turing	$4 \times 4 \times 4$	53.8 TFLOPS	GDDR6	11 GB	CUDA + WMMA
Nvidia RTX 3090 [35]	Ampere	$4 \times 8 \times 4$	71.0 TFLOPS	GDDR6X	24 GB	CUDA + WMMA
Huawei Ascend 310 [23]	DaVinci	$16 \times 16 \times 16$	11.0 TFLOPS	LPDDR4X	8 GB	TIK&TBE DSL [16]

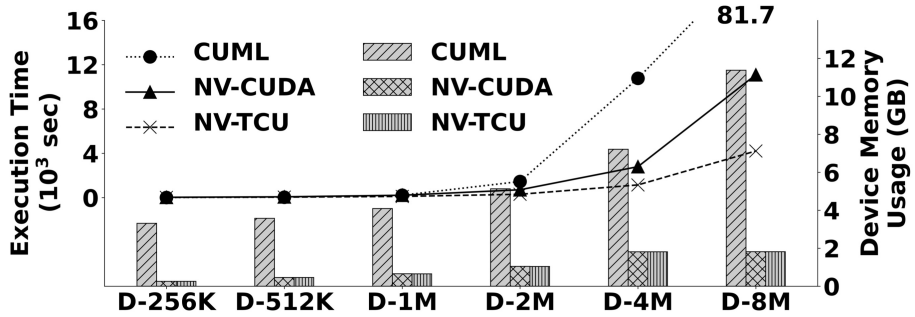
- Datasets are in 128-dimensional space
  - # of points: 256K – 8M
    - sizes range from 100MB to 10GB
  - # of clusters: 0 to 4096
    - 0 indicates all points are noise points
    - D-NORMAL: datasets with a reasonable number of clusters (i.e., 1 – 4096)
    - D-NOISE: dataset without any cluster

Evaluation: Mini Batch Framework

- Compared with CUML library
- NV-CUDA: use CUDA core to compute distance matrix
- NV-TCU: use TCU to compute distance matrix



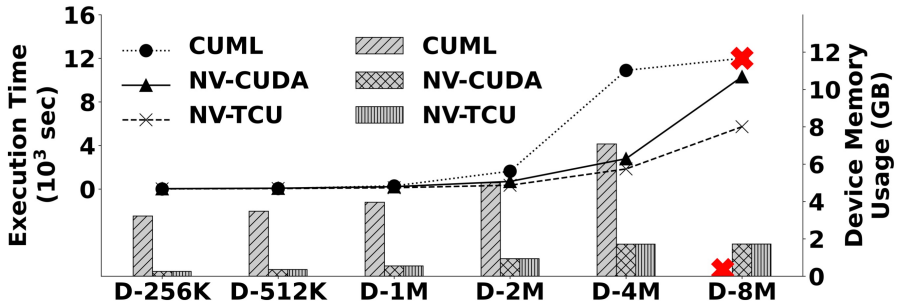
(a) Result on NVidia RTX 2080ti



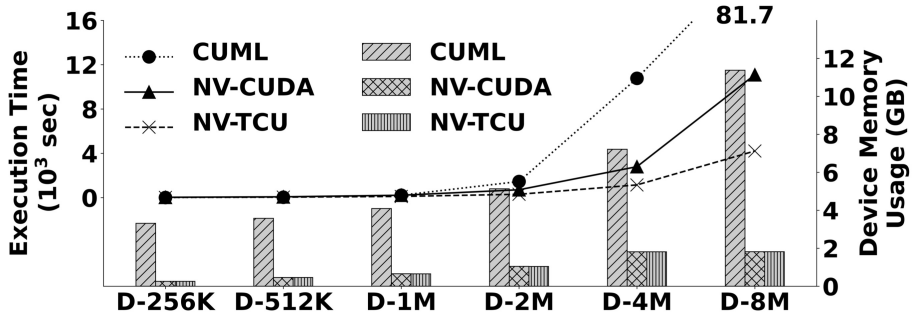
(b) Result on NVidia RTX 3090

- Small dataset: AC-DBSCAN is slightly slower than CUML
- Large dataset: AC-DBSCAN is much faster than CUML

Evaluation: Mini Batch Framework



(a) Result on NVidia RTX 2080ti

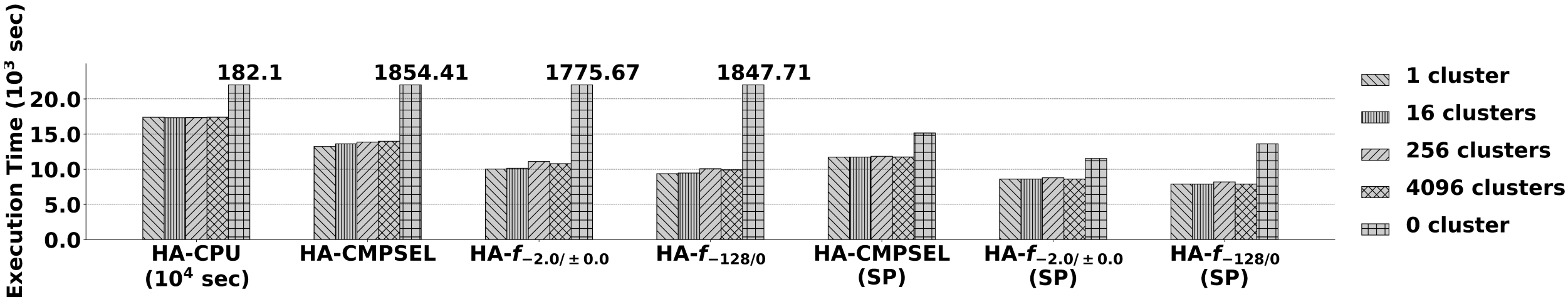


(b) Result on NVidia RTX 3090

- On D-8M: NV-TCU has 19.43x throughput of CUML on RTX 3090
  - CUML use 11.1GB device memory, mem error on RTX 2080ti
- Tensor core vs CUDA core
  - 2.6x speedup on RTX 3090
- Better scalability
  - device memory usage can be controlled by the size of the mini-batch

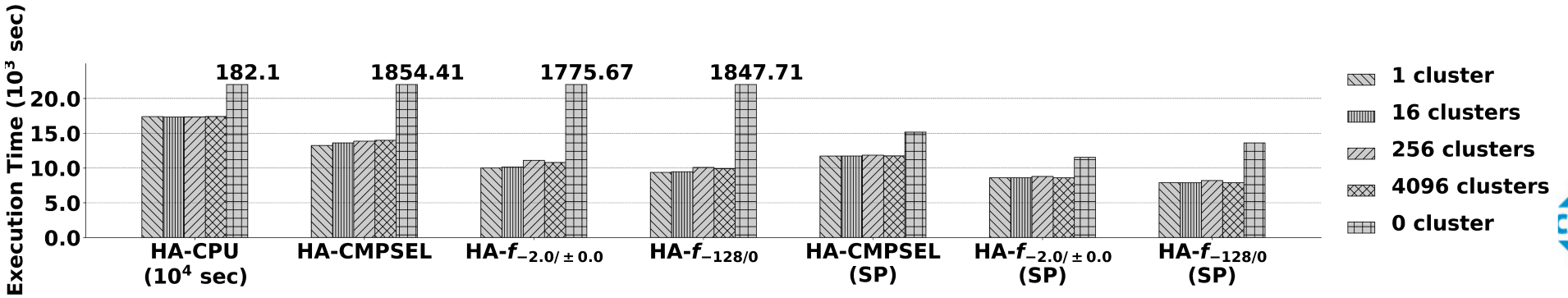
Evaluation: Mapping Functions

- HA-CPU: deploy ε-neighbor identification to CPU
- HA-CMPSEL: CMPSEL func in TBE DSL
- HA-$f_{-2.0/\pm 0.0}$, HA-$f_{-128/0}$: proposed method
- "(SP)" suffix indicates speculative execution is enabled



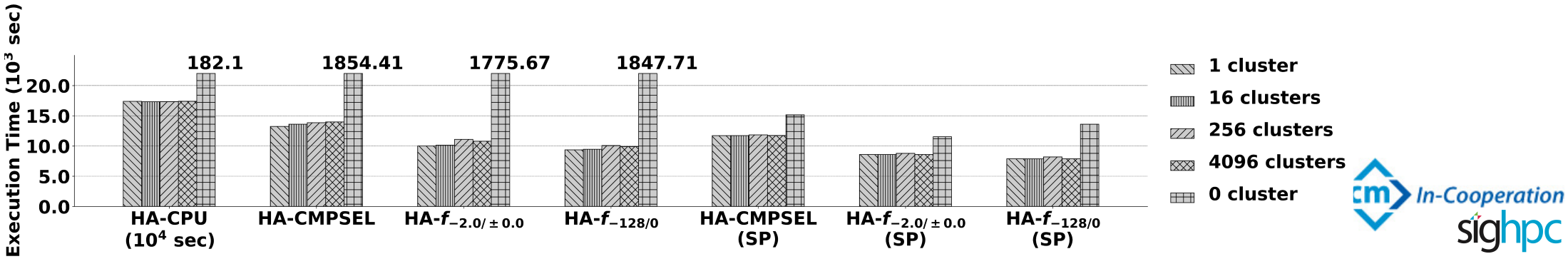
Evaluation: Mapping Functions

- Portability:
  - Our mapping functions are supported by almost all AI chips
    - No need to deploy computation to CPU
    - Avoid transferring distance matrix to host mem
      - much larger than adjacency matrix
    - Vectorized bitwise op is usually faster than CMPSEL on CPUs
  - 16.51x and 17.88x throughput



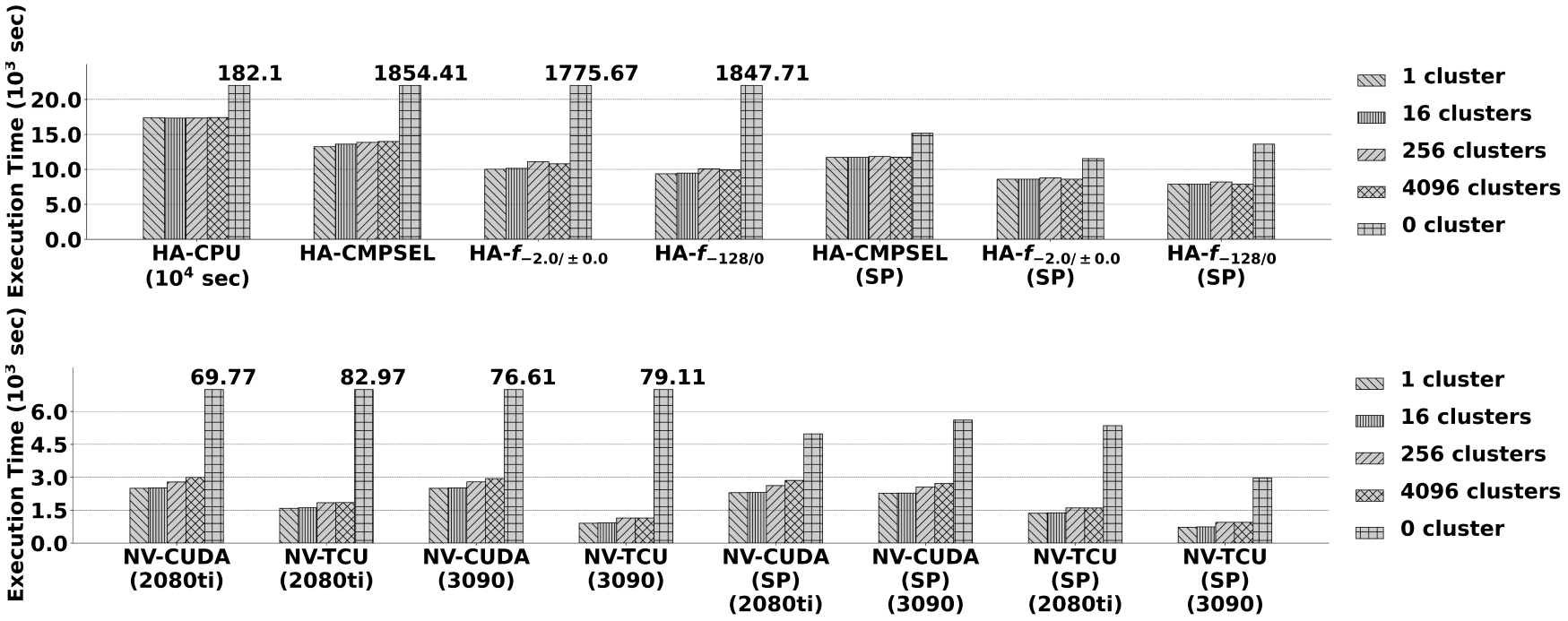
Evaluation: Mapping Functions

- Performance Portability:
  - The performance is also portable
    - i.e., have notional performance on different AI chips.
    - The hardware implementations of bitwise AND/OR are about the same.
    - In contrast, the CMPSEL operation can be implemented in various ways.
  - No need to fine-tune the performance
  - 23.1% and 29.0% higher throughput of HA-CMPSEL



Evaluation: Speculative Execution

- Speculative merging speedup
  - NV-TCU by 16.4% on D-NORMAL
  - HA-$f_{-128/0}$ by 15.8% on D-NORMAL
- Speculative initialization avoid very long execution time caused by abnormal datasets (e.g., D-NOISE)
  - reduces 93.9% on Nvidia GPUs
  - reduces 99.3% on Ascend 310



Future Work

- Extend AC-DBSCAN to multiple AI chips and distributed system
  - the mini-batch manner -> good scalability
  - rethink the bottleneck
    - large capacity SSD is common
    - GPUDirect can transfer SSD -> GPU directly
  - save the whole dataset in each machine's SSD and assign works based on computation?

- Integrating AC-DBSCAN with
  - accelerating index structures
  - approximation algorithms

Conclusion

- This paper presented AC-DBSCAN , a DBSCAN algorithm designed for AI chips.
  - 2.61× throughput by deploying distance calculation to Ampere TCUs
  - With high portability, our ε-neighbor identification kernels can be executed on almost all AI chips
    - 16.20× higher throughput than deploying ε-neighbor identification to CPUs.
  - HA-$f_{-128/0}$ reduces the execution time by 29.0% compared with HA-CMPSEL.
  - The speculative execution further reduces the execution time by 15.1% on D-NORMAL and 99.0% on D-NOISE.

THANK YOU

Q & A