University of Stuttgart
Germany

Institute for Parallel and Distributed Systems

Scientific Computing

Marcel.Breyer@ipvs.uni-stuttgart.de

Marcel Breyer

Performance-Portable Distributed k-Nearest Neighbors using Locality-Sensitive Hashing and SYCL

Motivation: Data Mining - Classification

- data mining is important in the age of data collection
- classification as one task
- **k-Nearest Neighbors** as one classifier (proposed by Thomas Cover and P. Hart in 1967)

Motivation: Data Mining - Classification

- data mining is important in the age of data collection
- classification as one task
- **k-Nearest Neighbors** as one classifier (proposed by Thomas Cover and P. Hart in 1967)
- naive Brute-Force is infeasible for large data sets

Motivation: Data Mining - Classification

- data mining is important in the age of data collection
- classification as one task
- **k-Nearest Neighbors** as one classifier (proposed by Thomas Cover and P. Hart in 1967)
- naive Brute-Force is infeasible for large data sets

→ Locality-Sensitive Hashing

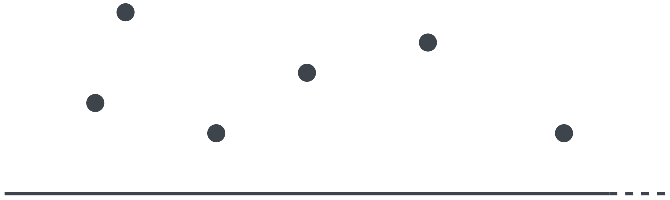
Locality-Sensitive Hashing

1

Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

hash value	points
0	
1	
2	
3	
4	

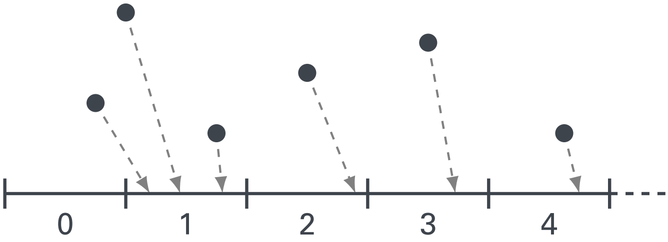
Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)



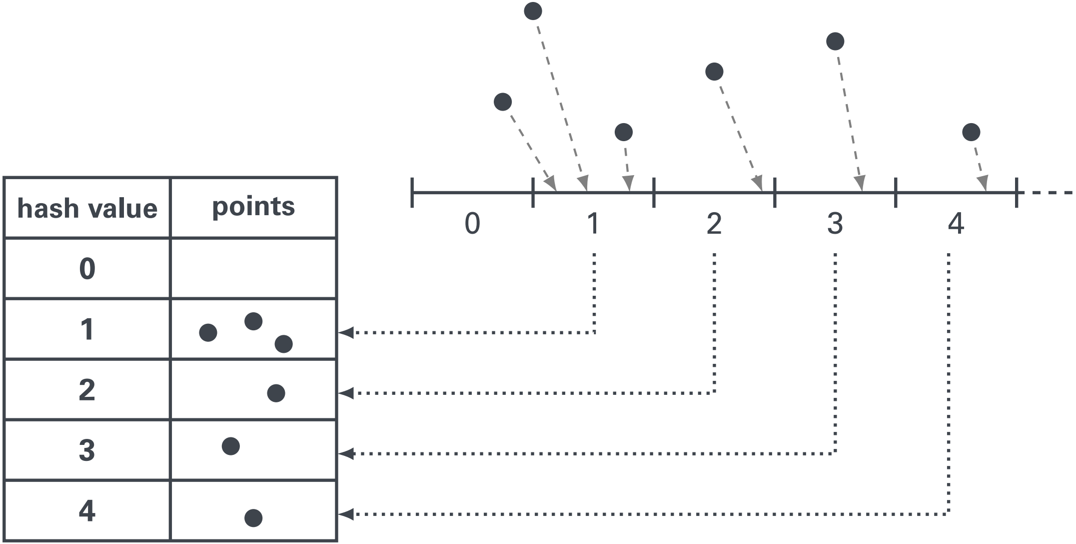
hash value	points
0	
1	
2	
3	
4	

Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

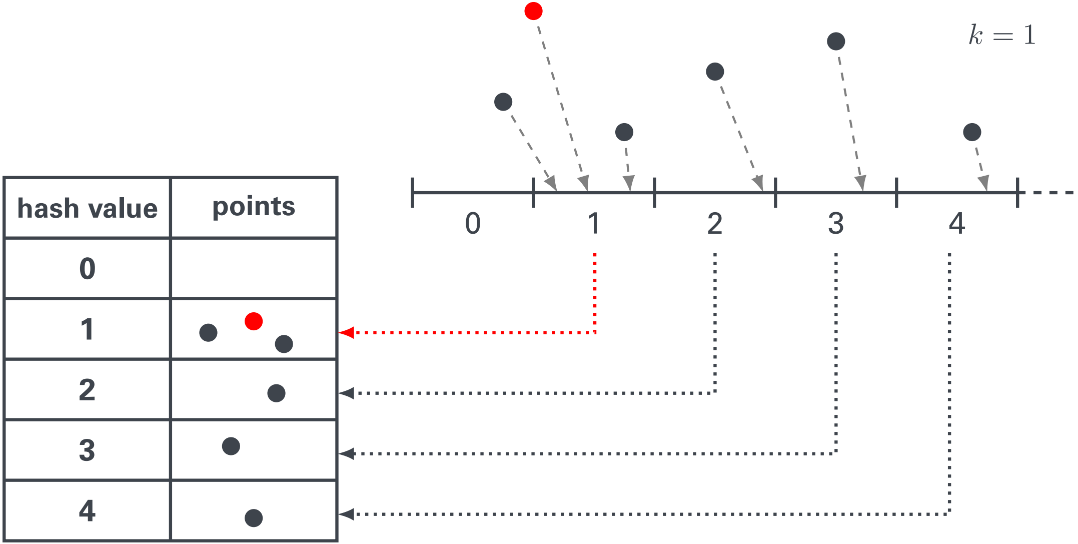
hash value	points
0	
1	
2	
3	
4	



Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)



Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

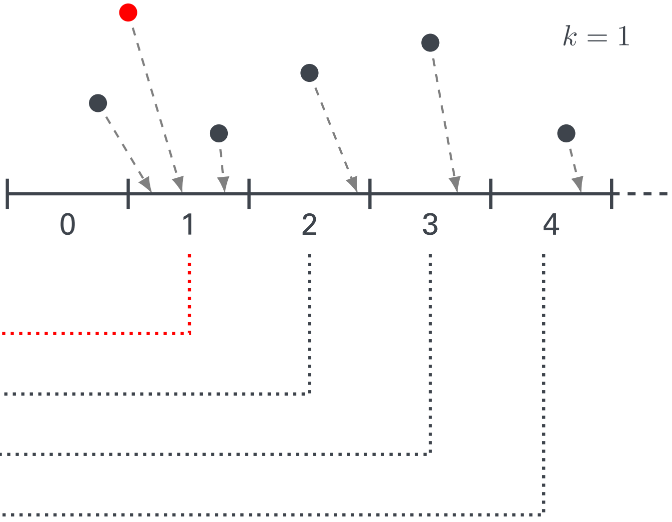


Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

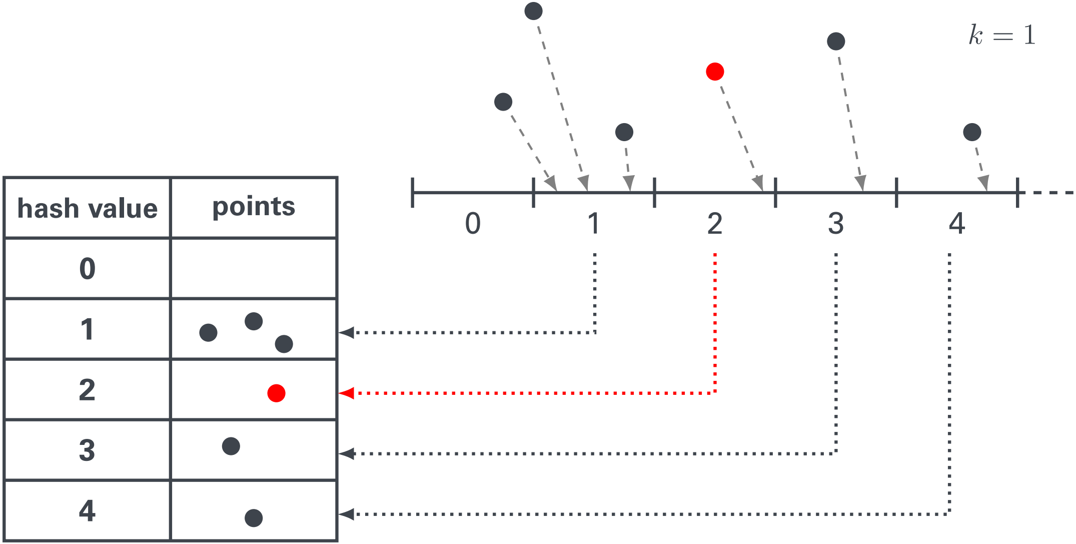
→ too many points per bucket
→ use multiple hash functions:

$$g(\vec{x}) = h_1(\vec{x}) \circ \ldots \circ h_m(\vec{x})$$

hash value	points
0	
1	
2	
3	
4	

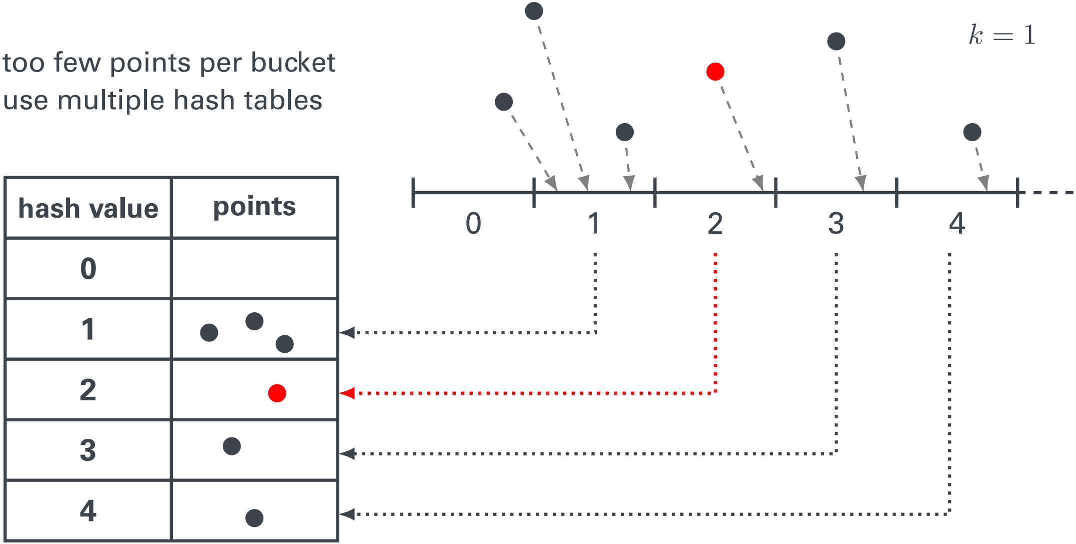


Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

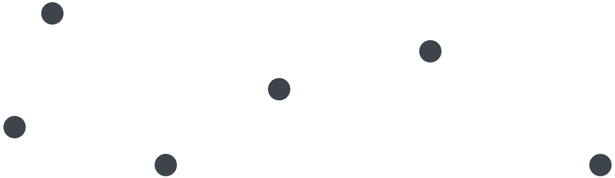


Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

→ too few points per bucket
→ use multiple hash tables

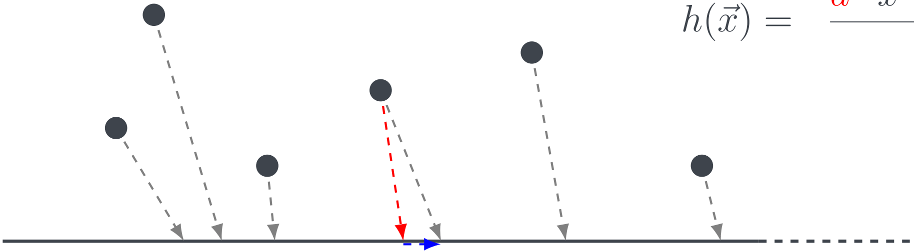


Random Projections (proposed by Mayur Datar et al.)



Random Projections (proposed by Mayur Datar et al.)

$$h(\vec{x}) = \frac{\vec{a} \cdot \vec{x} + b}{\;}$$

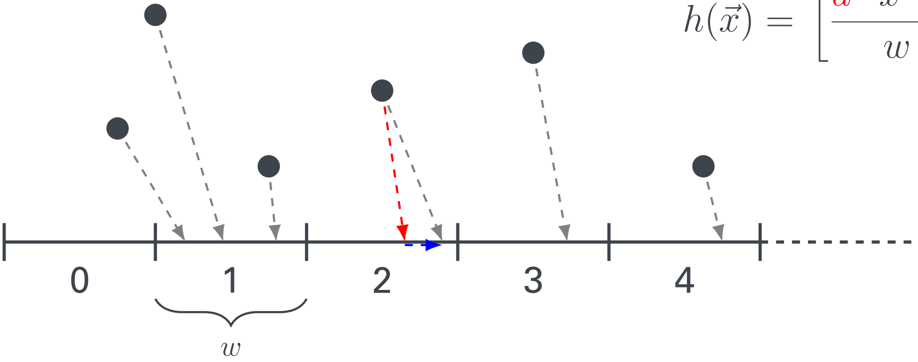


$\vec{a} \in \mathbb{R}^d$: independently choosen from the normal distribution

$b \in \mathbb{R}$: choosen uniformly from $[0, w]$

Random Projections (proposed by Mayur Datar et al.)

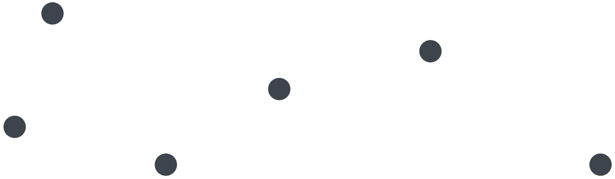
$$h(\vec{x}) = \left\lfloor \frac{\vec{a} \cdot \vec{x} + b}{w} \right\rfloor$$



$\vec{a} \in \mathbb{R}^d$: independently choosen from the normal distribution

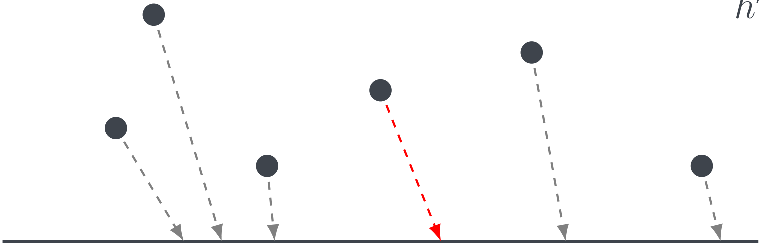
$b \in \mathbb{R}$: choosen uniformly from $[0, w]$

Entropy-Based Hash Functions (proposed by Qiang Wang et al.)



Entropy-Based Hash Functions (proposed by Qiang Wang et al.)

$$h'(\vec{x}) = \vec{a} \cdot \vec{x}$$

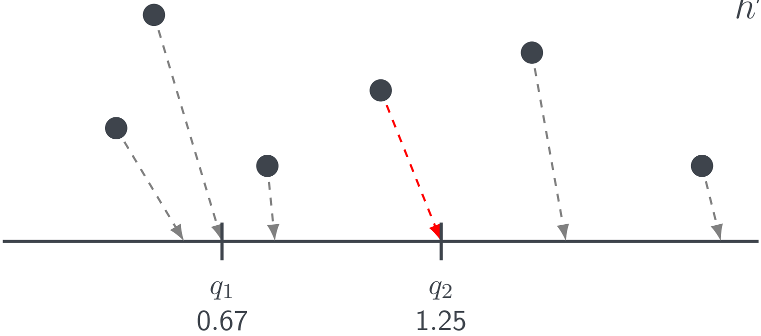


$\vec{a} \in \mathbb{R}^d$: independently choosen from the normal distribution

Entropy-Based Hash Functions (proposed by Qiang Wang et al.)

$$h'(\vec{x}) = \vec{a} \cdot \vec{x}$$

$$r = 3$$

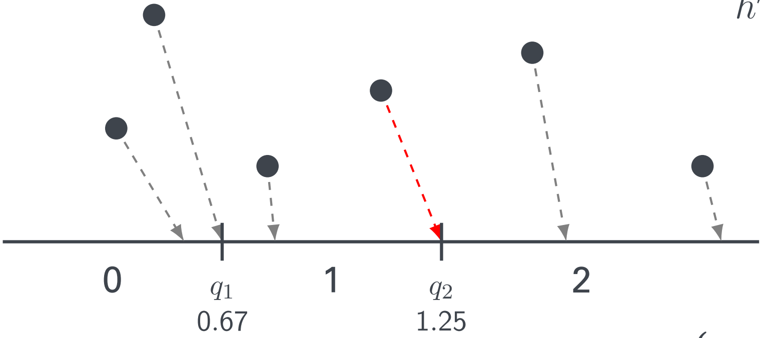


$\vec{a} \in \mathbb{R}^d$: independently choosen from the normal distribution

Entropy-Based Hash Functions (proposed by Qiang Wang et al.)

$$h'(\vec{x}) = \vec{a} \cdot \vec{x}$$

$$r = 3$$



$\vec{a} \in \mathbb{R}^d$: independently choosen from the normal distribution

$$h(\vec{x}) = \begin{cases} 0 & h'(\vec{x}) \leq q_1 \\ 1 & q_1 < h'(\vec{x}) \leq q_2 \\ 2 & h'(\vec{x}) > q_2 \end{cases}$$

SYCL

2

What is SYCL?

- cross-platform abstraction layer for heterogeneous computing
  ➔ can target a variety of different hardware platforms
  ➔ SYCL 1.2.1: build on top of OpenCL
  ➔ SYCL 2020: allows the usage of other backends like NVIDIA's CUDA,
  AMD's ROCm, or Intel's Level Zero

What is SYCL?

- cross-platform abstraction layer for heterogeneous computing
  ➔ can target a variety of different hardware platforms
  ➔ SYCL 1.2.1: build on top of OpenCL
  ➔ SYCL 2020: allows the usage of other backends like NVIDIA's CUDA, AMD's ROCm, or Intel's Level Zero

- combines the concepts, portability, and efficiency of standards like OpenCL with the ease of use of modern C++
  ➔ C++ constructs like templates or inheritance in kernel code explicitly allowed

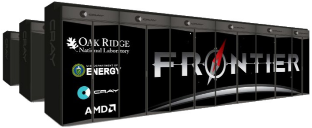
What is SYCL?

- cross-platform abstraction layer for heterogeneous computing
  ➔ can target a variety of different hardware platforms
  ➔ SYCL 1.2.1: build on top of OpenCL
  ➔ SYCL 2020: allows the usage of other backends like NVIDIA's CUDA, AMD's ROCm, or Intel's Level Zero

- combines the concepts, portability, and efficiency of standards like OpenCL with the ease of use of modern C++
  ➔ C++ constructs like templates or inheritance in kernel code explicitly allowed

- Single-Source Multiple Compiler-Passes

Why use SYCL?



(1)

Frontier: AMD CPUs + AMD GPUs

(2)

Perlmutter: AMD CPUs + NVIDIA GPUs

(3)

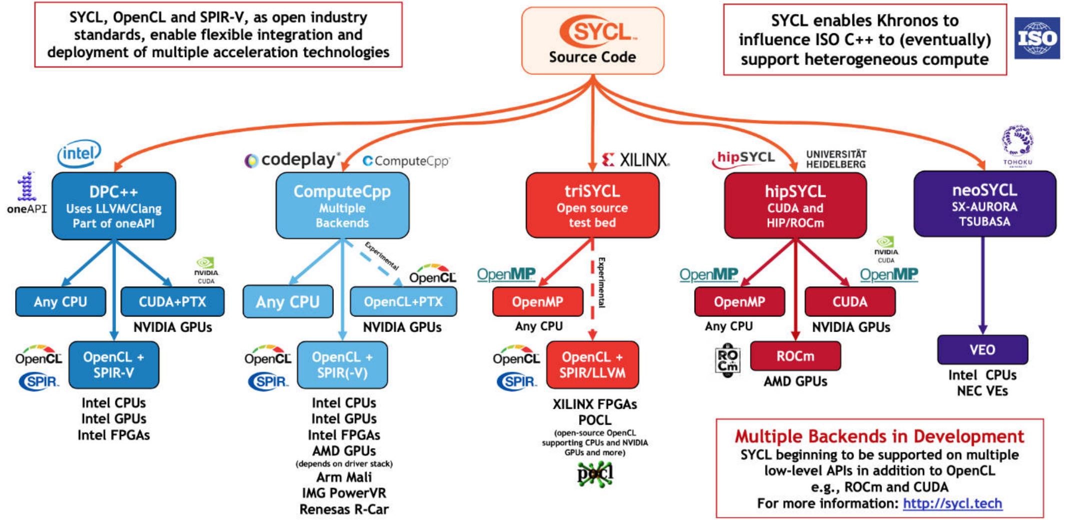
Aurora: Intel CPUs + Intel GPUs

(4)

HPC5: Intel CPUs + NVIDIA GPUs

(1): www.hpe.com/de/de/compute/hpc/cray/oak-ridge-national-laboratory.html (01.04.2021)
(2): www.waccpd.org/wp-content/uploads/2019/12/NickWright_Keynote_N9_WACCPD_2019.pdf (01.04.2021)
(3): www.hpe.com/uk/en/compute/hpc/cray/argonne-national-laboratory.html (01.04.2021)
(4): www.eni.com/en-IT/operations/green-data-center-hpc5.html (01.04.2021)

SYCL Implementations

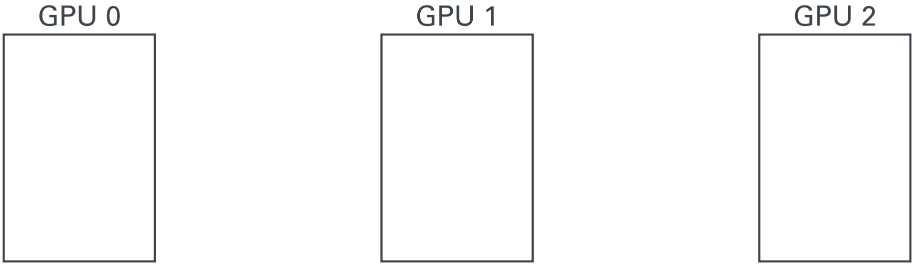


www.khronos.org/sycl/ (01.04.2021)

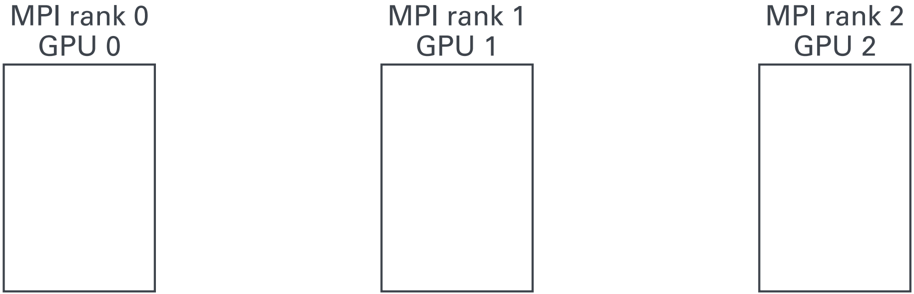
Implemen-tation

3

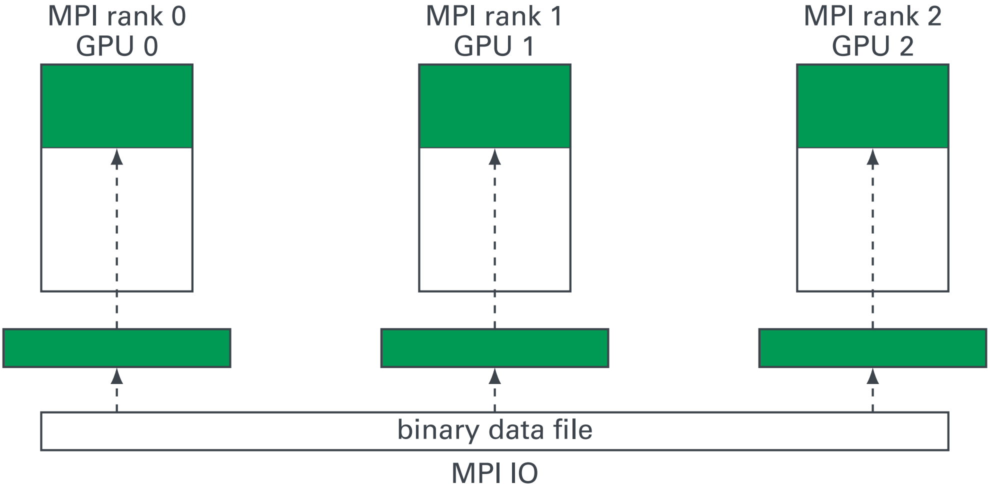
Distributed Multi-GPU Support using MPI



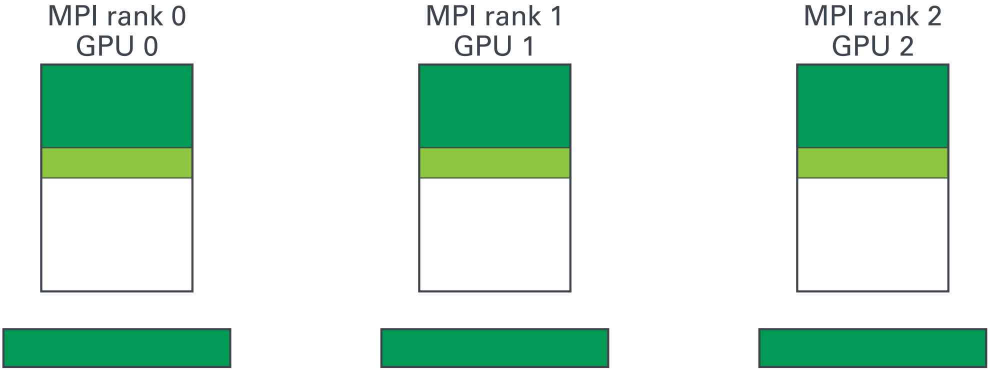
Distributed Multi-GPU Support using MPI



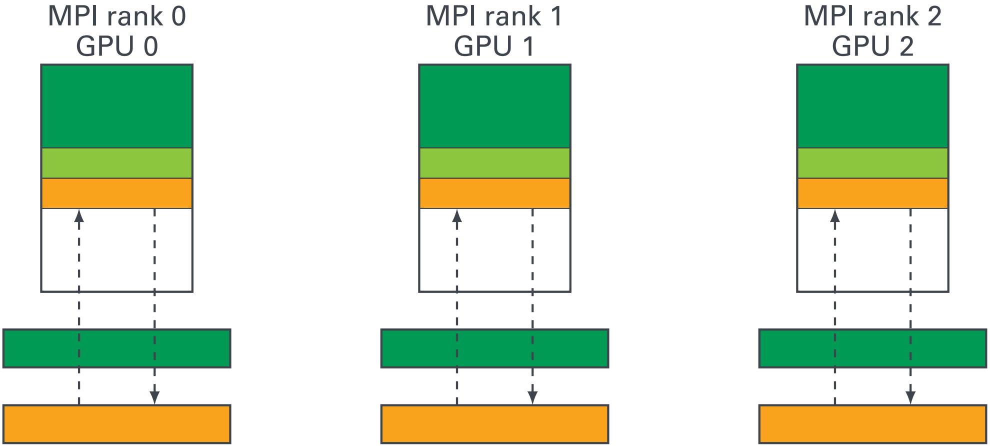
Distributed Multi-GPU Support using MPI



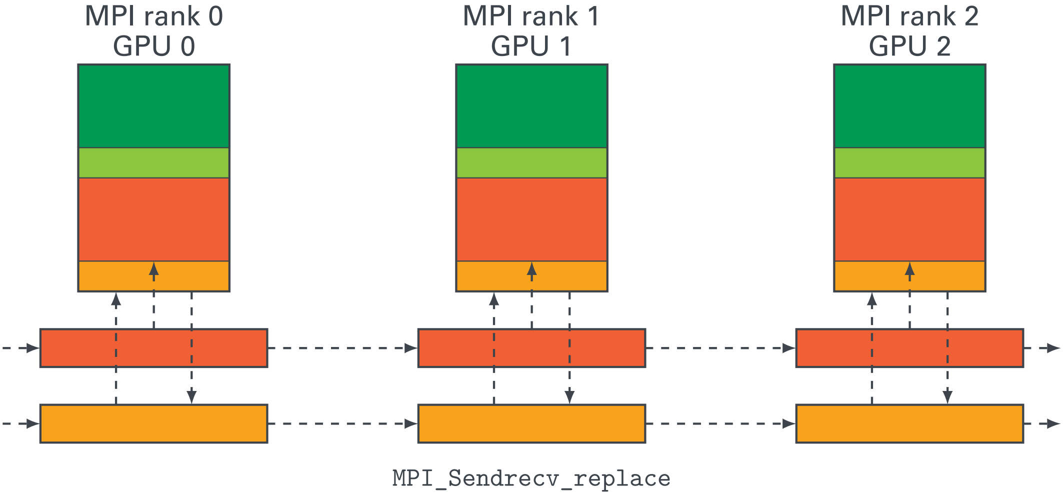
Distributed Multi-GPU Support using MPI



Distributed Multi-GPU Support using MPI



Distributed Multi-GPU Support using MPI



Results

4

Setup

	`argon-gtx`	Intel's `devcloud`
processors	Intel Xeon Gold 5120	Intel i9-10920X
number of sockets	2	1
processor frequency	2.2 GHz	3.5 GHz
total number of cores	28 (56 threads)	12 (24 threads)
main memory	754 GB	32 GB
accelerators	8x NVIDIA GeForce 1080 Ti	Intel Iris X^e MAX
SYCL	ComputeCpp, hipSYCL, DPC++	DPC++

Setup

	argon-gtx	Intel's devcloud
processors	Intel Xeon Gold 5120	Intel i9-10920X
number of sockets	2	1
processor frequency	2.2 GHz	3.5 GHz
total number of cores	28 (56 threads)	12 (24 threads)
main memory	754 GB	32 GB
accelerators	8x NVIDIA GeForce 1080 Ti	Intel Iris X^e MAX
SYCL	ComputeCpp, hipSYCL, DPC++	DPC++

friedman: 500 000 points in 10 dimensions (synthetic)

HIGGS: 1 000 000 points in 27 dimensions (real world)

Evaluation Metrics

$$\frac{\text{true positives}}{\text{relevant elements}}$$

recall

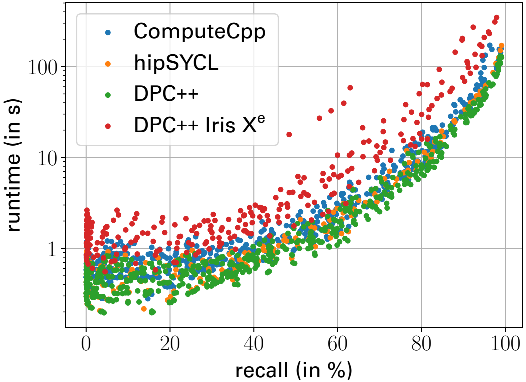
$$\frac{1}{N} \cdot \sum_{i=1}^{N} \left(\frac{1}{k} \cdot \sum_{j=1}^{k} \frac{dist_{LSH_j}}{dist_{correct_j}} \right)$$

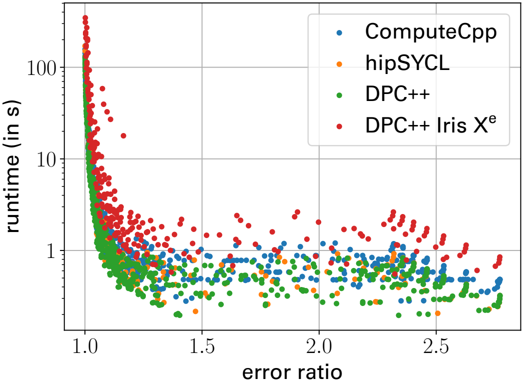
error ratio

$$S_p = \frac{T_1}{T_p}$$

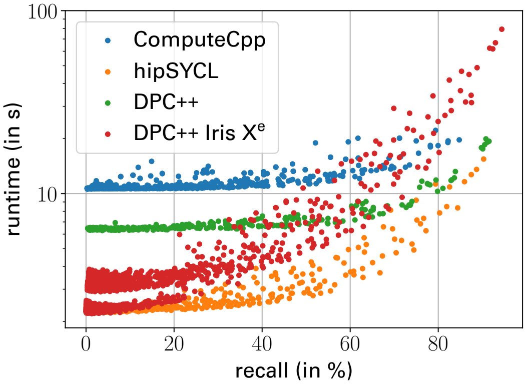
speedup

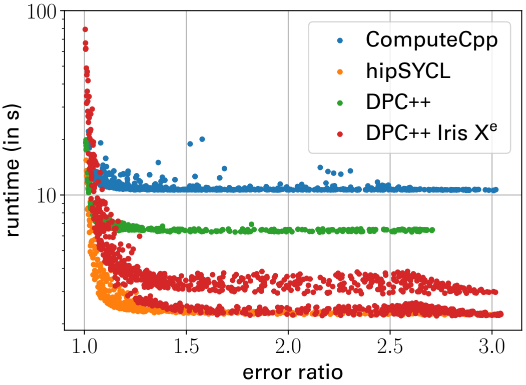
Random Projections - `friedman`



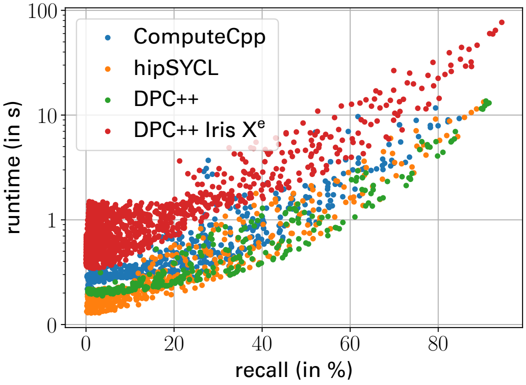


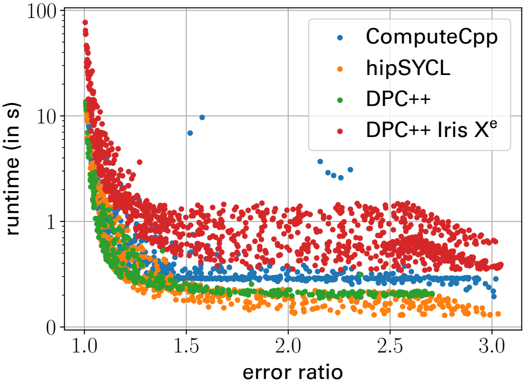
Entropy-Based Hash Functions - `friedman`





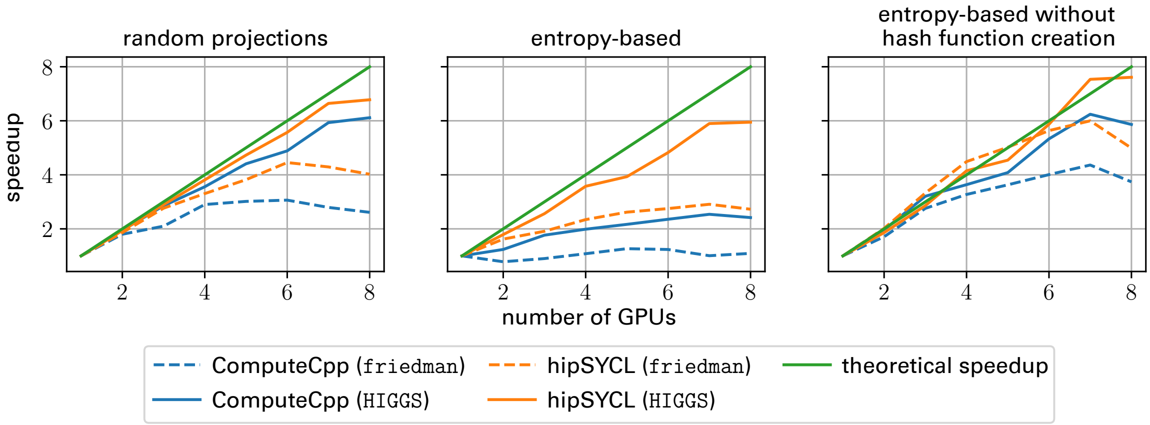
Entropy-Based Hash Functions - `friedman`



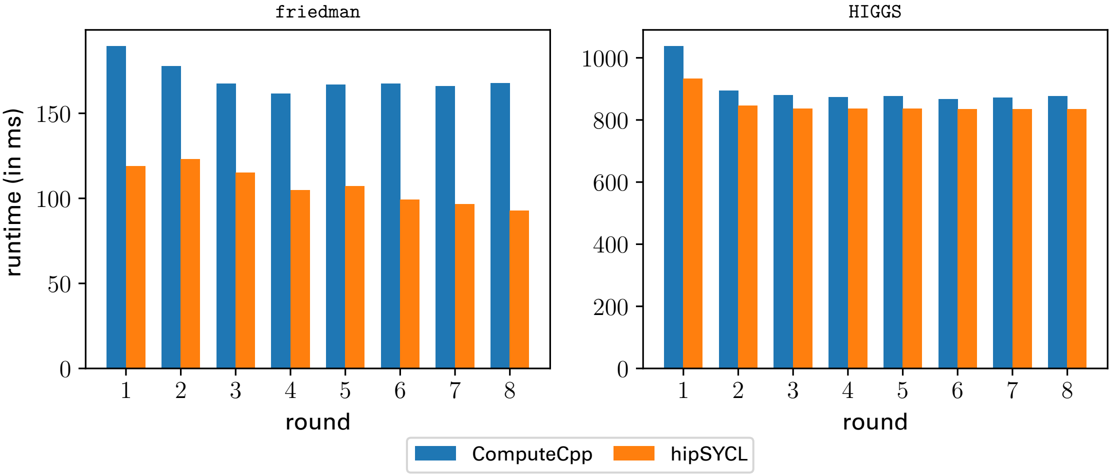


→ without hash function creation

Scaling - Speedup



Scaling - Runtimes per Round - Random Projections



Conclusion

5

Conclusion

- better recalls and error ratios increase runtime
  ➔ if smaller recalls or bigger error ratios are sufficient, the runtime decreases drastically

Conclusion

- better recalls and error ratios increase runtime
  ➔ if smaller recalls or bigger error ratios are sufficient, the runtime decreases drastically

- comparable results for random projections and entropy-based hash functions

Conclusion

- better recalls and error ratios increase runtime
  → if smaller recalls or bigger error ratios are sufficient, the runtime decreases drastically
- comparable results for random projections and entropy-based hash functions
- easily scalable on multiple GPUs
  → parallel speedup of up to 7 using 8 GPUs
  → for short kernel invocations hipSYCL scales better than ComputeCpp because of a smaller static overhead

Conclusion

- better recalls and error ratios increase runtime
  ➔ if smaller recalls or bigger error ratios are sufficient, the runtime decreases drastically
- comparable results for random projections and entropy-based hash functions
- easily scalable on multiple GPUs
  ➔ parallel speedup of up to 7 using 8 GPUs
  ➔ for short kernel invocations hipSYCL scales better than ComputeCpp because of a smaller static overhead
- runtime characteristics are similar for ComputeCpp, hipSYCL, and DPC++
  ➔ except for ComputeCpp and DPC++ when using entropy-based hash functions and NVIDIA GPUs in the hash function creation step

Further Reading

k-Nearest Neighbors as Classifier

Thomas Cover and P. Hart. "Nearest neighbor pattern classification". In: *IEEE Transactions on Information Theory* (1967)

Locality-Sensitive Hashing

Piotr Indyk and Rajeev Motwani. "Approximate nearest neighbors: towards removing the curse of dimensionality". In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 1998, pp. 604–613

Random Projections

Mayur Datar et al. "Locality-sensitive hashing scheme based on p-stable distributions". In: *Proceedings of the twentieth annual ACM symposium on Computational geometry*. ACM Press, 2004

Entropy-Based Hash Functions

Qiang Wang et al. "Entropy based locality sensitive hashing". In: *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2012

SYCL (DPC++)

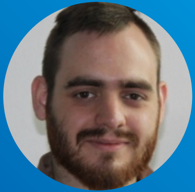
James Reinders et al. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Springer Nature, 2021

University of Stuttgart
Germany

Marcel Breyer

Marcel.Breyer@ipvs.uni-stuttgart.de

+49 711 685-88427



Gregor Daiß

Gregor.Daiss@ipvs.uni-stuttgart.de

+49 711 685-88365

Dirk Pflüger

Dirk.Pflueger@ipvs.uni-stuttgart.de

+49 711 685-70413