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

Marcel Breyer

Marcel.Breyer@ipvs.uni-stuttgart.de
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
Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

<table>
<thead>
<tr>
<th>hash value</th>
<th>points</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

<table>
<thead>
<tr>
<th>hash value</th>
<th>points</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

<table>
<thead>
<tr>
<th>hash value</th>
<th>points</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

<table>
<thead>
<tr>
<th>hash value</th>
<th>points</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>⋄</td>
</tr>
<tr>
<td>2</td>
<td>⋄</td>
</tr>
<tr>
<td>3</td>
<td>⋄</td>
</tr>
<tr>
<td>4</td>
<td>⋄</td>
</tr>
</tbody>
</table>
Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

<table>
<thead>
<tr>
<th>hash value</th>
<th>points</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><img src="0" alt="points" /></td>
</tr>
<tr>
<td>2</td>
<td><img src="0" alt="points" /></td>
</tr>
<tr>
<td>3</td>
<td><img src="0" alt="points" /></td>
</tr>
<tr>
<td>4</td>
<td><img src="0" alt="points" /></td>
</tr>
</tbody>
</table>

$k = 1$

Marcel Breyer, University of Stuttgart, IPVS, SC: Performance-Portable Distributed k-Nearest Neighbors using Locality-Sensitive Hashing and SYCL
Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

→ too many points per bucket
→ use multiple hash functions:
\[ g(\vec{x}) = h_1(\vec{x}) \circ \ldots \circ h_m(\vec{x}) \]

<table>
<thead>
<tr>
<th>hash value</th>
<th>points</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>⬤ ⬤</td>
</tr>
<tr>
<td>2</td>
<td>⬤</td>
</tr>
<tr>
<td>3</td>
<td>⬤</td>
</tr>
<tr>
<td>4</td>
<td>⬤</td>
</tr>
</tbody>
</table>

\[ k = 1 \]
Locality-Sensitive Hashing (proposed by Piotr Indyk and Rajeev Motwani)

→ too few points per bucket
→ use multiple hash tables

<table>
<thead>
<tr>
<th>hash value</th>
<th>points</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

$k = 1$
Random Projections (proposed by Mayur Datar et al.)
Random Projections (proposed by Mayur Datar et al.)

\[ h(\vec{x}) = \vec{a} \cdot \vec{x} + b \]

\( \vec{a} \in \mathbb{R}^d \): 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.)

\[ \vec{a} \in \mathbb{R}^d : \] independently chosen from the normal distribution

\[ q_1 \leq \vec{a}_i \leq q_2 \]

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

\[ h (\vec{x}) = \begin{cases} 
0 & h' (\vec{x}) \leq q_1 \\
q_1 < h' (\vec{x}) \leq q_2 & \text{} \\
q_2 > h' (\vec{x}) & \end{cases} \]
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 chosen from the normal distribution
Entroopy-Based Hash Functions (proposed by Qiang Wang et al.)

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

\[ r = 3 \]

\[ \vec{a} \in \mathbb{R}^d: \text{ independently choosen from the normal distribution} \]
Entropy-Based Hash Functions (proposed by Qiang Wang et al.)

\[ \vec{a} \in \mathbb{R}^d: \text{ independently chosen from the normal distribution} \]

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

\[ r = 3 \]

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

Frontier: AMD CPUs + AMD GPUs

Perlmutter: AMD CPUs + NVIDIA GPUs

Aurora: Intel CPUs + Intel GPUs

HPC5: Intel CPUs + NVIDIA GPUs

**SYCL Implementations**

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies.

- **Intel**
  - DPC++
    - Uses LLVM/Clang Part of oneAPI
    - Any CPU
    - Intel CPUs
    - Intel GPUs
    - Intel FPGAs
    - Intel CPUs
    - Intel GPUs
    - Intel FPGAs
    - AMD GPUs
    - (depends on driver stack)
    - Arm Mali
    - IMG PowerVR
    - Renesas R-Car

- **Codeplay**
  - ComputeCpp
    - Multiple Backends
    - Any CPU
    - NVIDIA GPUs

- **Xilinx**
  - triSYCL
    - Open source test bed
    - OpenCL
    - NVIDIA GPUs

- **HipSYCL**
  -CUDA and HIP/ROCM
    - Any CPU
    - NVIDIA GPUs

- **NeoSYCL**
  - SX-AURORA TSUBASA
    - VEO
    - Intel CPUs
    - NEC VEs

**Multiple Backends in Development**
SYCL beginning to be supported on multiple low-level APIs in addition to OpenCL e.g., ROCm and CUDA
For more information: [http://sycl.tech](http://sycl.tech)
Implementation
Distributed Multi-GPU Support using MPI

GPU 0

GPU 1

GPU 2

MPI rank 0

MPI rank 1

MPI rank 2

MPI IO

binary data file

Marcel Breyer, University of Stuttgart, IPVS, SC: Performance-Portable Distributed k-Nearest Neighbors using Locality-Sensitive Hashing and SYCL
Distributed Multi-GPU Support using MPI

MPI rank 0
GPU 0

MPI rank 1
GPU 1

MPI rank 2
GPU 2
Distributed Multi-GPU Support using MPI

MPI rank 0
GPU 0

MPI rank 1
GPU 1

MPI rank 2
GPU 2

MPI IO

binary data file

Marcel Breyer, University of Stuttgart, IPVS, SC: Performance-Portable Distributed k-Nearest Neighbors using Locality-Sensitive Hashing and SYCL
Distributed Multi-GPU Support using MPI

MPI rank 0
GPU 0

MPI rank 1
GPU 1

MPI rank 2
GPU 2

Marcel Breyer, University of Stuttgart, IPVS, SC: Performance-Portable Distributed k-Nearest Neighbors using Locality-Sensitive Hashing and SYCL
Distributed Multi-GPU Support using MPI
Distributed Multi-GPU Support using MPI

MPI rank 0
GPU 0

MPI rank 1
GPU 1

MPI rank 2
GPU 2

MPI_Sendrecv_replace

Marcel Breyer, University of Stuttgart, IPVS, SC: Performance-Portable Distributed k-Nearest Neighbors using Locality-Sensitive Hashing and SYCL
## Setup

<table>
<thead>
<tr>
<th></th>
<th>argon-gtx</th>
<th>Intel’s devcloud</th>
</tr>
</thead>
<tbody>
<tr>
<td>processors</td>
<td>Intel Xeon Gold 5120</td>
<td>Intel i9-10920X</td>
</tr>
<tr>
<td>number of sockets</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>processor frequency</td>
<td>2.2 GHz</td>
<td>3.5 GHz</td>
</tr>
<tr>
<td>total number of cores</td>
<td>28 (56 threads)</td>
<td>12 (24 threads)</td>
</tr>
<tr>
<td>main memory</td>
<td>754 GB</td>
<td>32 GB</td>
</tr>
<tr>
<td>accelerators</td>
<td>8x NVIDIA GeForce 1080 Ti</td>
<td>Intel Iris Xe MAX</td>
</tr>
<tr>
<td>SYCL</td>
<td>ComputeCpp, hipSYCL, DPC++</td>
<td>DPC++</td>
</tr>
</tbody>
</table>

Marcel Breyer, University of Stuttgart, IPVS, SC: Performance-Portable Distributed k-Nearest Neighbors using Locality-Sensitive Hashing and SYCL
## Setup

<table>
<thead>
<tr>
<th></th>
<th>argon-gtx</th>
<th>Intel’s devcloud</th>
</tr>
</thead>
<tbody>
<tr>
<td>processors</td>
<td>Intel Xeon Gold 5120</td>
<td>Intel i9-10920X</td>
</tr>
<tr>
<td>number of sockets</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>processor frequency</td>
<td>2.2 GHz</td>
<td>3.5 GHz</td>
</tr>
<tr>
<td>total number of cores</td>
<td>28 (56 threads)</td>
<td>12 (24 threads)</td>
</tr>
<tr>
<td>main memory</td>
<td>754 GB</td>
<td>32 GB</td>
</tr>
<tr>
<td>accelerators</td>
<td>8x NVIDIA GeForce 1080 Ti</td>
<td>Intel Iris X&lt;sup&gt;e&lt;/sup&gt; MAX</td>
</tr>
<tr>
<td>SYCL</td>
<td>ComputeCpp, hipSYCL, DPC++</td>
<td>DPC++</td>
</tr>
</tbody>
</table>

**friedman:** 500 000 points in 10 dimensions  
**HIGGS:** 1 000 000 points in 27 dimensions
Evaluation Metrics

\[
\text{true positives} \quad \frac{1}{N} \cdot \sum_{i=1}^{N} \left( \frac{1}{k} \cdot \sum_{j=1}^{k} \frac{\text{dist}_{\text{LSH}_j}}{\text{dist}_{\text{correct}_j}} \right)
\]

\[
\text{relevant elements} \quad S_p = \frac{T_1}{T_p}
\]

Recall

Error ratio

Speedup
Random Projections - friedman

Marcel Breyer, University of Stuttgart, IPVS, SC: Performance-Portable Distributed k-Nearest Neighbors using Locality-Sensitive Hashing and SYCL
Entropry-Based Hash Functions - friedman

Marcel Breyer, University of Stuttgart, IPVS, SC: Performance-Portable Distributed k-Nearest Neighbors using Locality-Sensitive Hashing and SYCL
Entroy-Based Hash Functions - friedman

→ without hash function creation
Scaling - Speedup

**random projections**

**entropy-based**

**entropy-based without hash function creation**

- ComputeCpp *(friedman)*
- ComputeCpp *(HIGGS)*
- hipSYCL *(friedman)*
- hipSYCL *(HIGGS)*
- theoretical speedup

Marcel Breyer, University of Stuttgart, IPVS, SC: Performance-Portable Distributed k-Nearest Neighbors using Locality-Sensitive Hashing and SYCL
Scaling - Runtimes per Round - Random Projections

![Graph showing runtimes per round for different datasets and libraries.](image)

- **friedman**
- **HIGGS**

Marcel Breyer, University of Stuttgart, IPVS, SC: Performance-Portable Distributed k-Nearest Neighbors using Locality-Sensitive Hashing and SYCL
Conclusion
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

Locality-Sensitive Hashing

Random Projections

Entropy-Based Hash Functions

SYCL (DPC++)