We share our experience of enabling a popular machine learning (ML) algorithm: K-nearest neighbor (KNN) using a recently proposed task-based target-agnostic heterogeneous open source streaming library (hStreams, IPPDFW 2016) that supports task-concurrency across heterogeneous platforms.

Contributions:
- Opens up the breadth of algorithms implemented using hStreams.
- So far, hStreams has been used to implement Matrix multiplication, Checkeds, LU factorization, and many other important iVS apps.
- This paper: the first example, showcasing hStreams' ability to enable ML algorithms.
- hStreams enabled KNN achieves the best performance achievable by either Xeon or Xeon Phi by utilizing both platforms simultaneously and selectively.

hStreams

What is hStreams and how it works [1]:
- Recently proposed open source streaming library that supports task-concurrency on heterogeneous Intel® based platforms (Xeon® and Xeon Phi™ families, potentially FPGAs).
- Task means serial or parallel function that works on some data. The task concurrency can be across nodes as well as within a node.
- hStreams automatically overlaps independent computations with communications, relieving the user from the complexity of pipelining, thread affinity, asynchronous scheduling, and memory management.
- Offers a platform-agnostic interface of communicating with the sink side: promotes portability.
- The main execution units are called streams: can be considered as FIFO queues holding tasks to be executed.
- Supports offload programming model: a stream has a source side and a sink side (receiving end of an offload operation). Tasks are enqueued at source and executed at sink.
- Divides available system resources (compute and memory) into domains to be used by those streams by default.
- Computation, communication, and synchronization tasks are enqueued in the streams and executed in FIFO order, except that if a communication task does not depend on a preceding computation task, these gets automatically overlapped.

Two independent tasks need to be enqueued on two different streams to make them run in parallel.

Tasks
- Streams on host/Xeon®, domain 1
- Streams on sink/XKN/KNL, domain 2
- Streams on sink2/XKN2/LKN domain 3

KNN Training

The K-nearest neighbors (KNN) Problem

Definition: Given n labeled training data points, and a query data point, find k data points from the given training data points who are closest to the query data point. Then find the prospective label for query data point based on the majority votes from the discovered k nearest neighbors.

Applications: ML, data mining, computer vision, text processing, scientific computing such as computational biology, astronomy, physics, and in other areas.

The Naive way to solve KNN

1. Take each query point, compute distances from the query point to all n training data points.
2. Pick the k training points closest to the query point.
3. Label the query point taking the majority labels of the k neighbors.
4. Requires $O(nk)$ time: computationally expensive.

The KNN Evaluation process

Enabling hStreams to solve KNN (base algorithm)

- Base code taken from highly optimized (vectorized, parallelized) shared-memory implementation of KNN developed by Intel® Parallel Computing Lab (Published in IPPDFW 2016 [2]).
- Divides KNN into two phases: Training and Classification.

Training:
- Uses high-dimensional kd-tree data structure [2, 3], which arranges the labeled data set into tree format by recursively partitioning them across each feasible feature space one by one, until it reaches a point where the data set size in a given tree node is small enough in terms of computation cost. Overall cost $O(n \log n)$.

Classification:
- For each query data point, k nearest neighbors are found by traversing the kd-tree: embarrassingly parallel w/t all query points. Overall cost $O(k \log k)$.

Performance trends of the base algorithm:

Systems Spec: (shows the systems we used to run all experiments)

- Xeon® 24-core CPU E5-2670 @ 2.70GHz, KNC-61-core Knights Corner GPU (Xeon Phi® 7120A, KNL-72-core Knights Landing D-Wiles-5MCDRAM, Compiler: Intel® icc 2016, hStreams 1.0 available at https://github.com/Pconstantinescu-hStreams/hStreams-1.02)
- Dataset: training data: 5M points, each with 16 dimensions, 500k queries

- Scalability: Training algorithm does not scale with hyper threads (best performance on Xeon® achieved at 24 cores, on KNC at 61 cores and on KNL at 71 cores).
- Classification algorithm is embarrassingly parallel and highly scalable (best performance on Xeon® achieved at 48 threads, on KNC at 243 threads, and on KNL at 247 threads).
- Runtime:

<table>
<thead>
<tr>
<th>Dataset</th>
<th>KNN Training</th>
<th>KNN Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M data points, 15 features k = 51, 500k query</td>
<td>25.27 ms</td>
<td>4 min 0.29 s</td>
</tr>
<tr>
<td>2M data points, 15 features k = 51, 500k query</td>
<td>33.16 s</td>
<td>4 min 0.29 s</td>
</tr>
</tbody>
</table>

Enabling KNN using hStreams

Naive way to solve KNN

- To achieve the best of both worlds, keep training phase on the host and use hStreams to offload bulk of the classification phase to the sink side.
- hStreams gives a unique interface, enabling the sink-side (receiving side of an ofcall offload) to be a range of targets (host, KNC, KNL, FPGA or even a GPU) using unified API interfaces. At this point supported targets are Xeon® and KNC.
- Expected performance improvement:
  - Since the original non-hStreams base classification code is already highly optimized and is able to use all resources, we do not see any performance improvement in classification compared to a native KNC run of the original base code.
  - Since the communication cost is asymptotically lower than the computation cost, we also do not expect to see any performance improvement with the computation communication overlap either.
  - However, using hStreams (that uses both Xeon® and Xeon Phi™) we expect to achieve the best of both worlds at the same lowest run times for Training and Classification.
- Enabling techniques:
  - Keep the entire training part on the host and transfer the kd-tree (the model built from the training data) to KNC at the end of the construction.
  - Offload the classification part (either partially or completely) to KNC.
  - Overhead for hStreams set up, memory allocations and sending the entire kd-tree data to the KNC is O(1).s - 0.6 s.

KNN Classification using hStreams:
- Divide the list of query data points into segments (similar to tiling)
- Create $s$ streams (task execution units consisting of multiple cores / threads) to run the K-nearest neighbor problem for those segments of query points in a round-robin fashion
  - $s$ segments of query points to get executed on stream $s i d$
- hStreams automatically overlaps communications with computations.

Using hStreams, we get the best of both worlds

- Systems Spec: Same as before, we used hStreams version 1.0 at available at https://github.com/Pconstantinescu-hStreams.
- Performance summary
  - In training phase, Xeon® is orders of magnitude faster than Xeon Phi™. Bigger and efficient Xeon® cores execute sequential, recursive and complex control flow of kd-tree construction and classification efficiently.
  - In classification phase Xeon Phi™ is 2% – 47% faster than Xeon®
  - Highly scalable and simple code, good for small and simple Phi cores
- Being able to utilize both Xeon 0 and Xeon Phi™ at proper phase, automatic overlapping of computation, communication and efficient exploitation of task-concurrency, hStreams based KNN implementation achieved the best performance achievable by any of these platforms.
- The training phase as well as data can also be distributed across host and sink, as shown in [2]. Pipelining of computation and communication is expected to improve performance even more there, especially for large/big datasets.
- Usually other algorithms, which are known to be benefited from pipelining and overlapping of computation and communication can get benefits from using hStreams.

Conclusion

hStreams is in its early stage of development with possibilities for many exciting features:

- Pipelining, streaming, task concurrency
- Overlaps of communication and computation
- Automatic dependency analysis
- Dynamic load-balancing

- hStreams complements CUDA Streams® available from NVidia.
- hStreams has demonstrated substantial performance improvement in dense, deep learning application (DGMM, LU, Cholesky, and tree BS codes [1]).
- This poster shows an example of hStreams being able to enable machine learning application.
- Contributions from open developer and user community would greatly improve hStreams ability to serve our need.
- We welcome users’ input and contributions in this effort. hStreams can be downloaded from https://github.com/Pconstantinescu-hStreams.
- Some names are the property of others

References


Acknowledgement

Thanks to Chris J. Newburn, Marcin Sasnawski for all their supports with hStreams. Thanks to Mostofa Ali Patwary for sharing KNN code and data.

Questions? Email: Jesmin.Jahan.tithi@intel.com

Optimizing K-nearest neighbor algorithm using a heterogeneous streaming library: hStreams

Jesmin Jahan Tithi

Platform Architecture and Performance Group, Intel Corporation, Santa Clara 95054, jesmin.jahan.tithi@intel.com

Intel

Platform Architecture and Performance Group, Intel Corporation, Santa Clara 95054, jesmin.jahan.tithi@intel.com