Enabling K-nearest Neighbor Algorithm Using a Heterogeneous Streaming Library: hStreams

Jesmin Jahan Tithi
Intel Corporation, Santa Clara, California. jesmin.jahan.tithi@intel.com

ABSTRACT

hStreams is a recently proposed (IPDPSW 2016) task-based target-agnostic heterogeneous streaming library that supports task concurrency over heterogeneous platforms. We share our experience of enabling a non-trivial machine learning (ML) algorithm: K-nearest neighbor using hStreams. The K-nearest neighbor (KNN) is a popular algorithm with numerous applications in machine learning, data-mining, computer vision, text processing, scientific computing such as computational biology, astronomy, physics, and others. This is the first example of showcasing hStreams' ability to enable an ML algorithm. hStreams enabled KNN achieves the best performance achievable by either Xeon® or Xeon Phi™ [1] by utilizing both platforms simultaneously and selectively.

Keywords

hStreams, heterogeneous streaming library, K-nearest neighbor, KNN, task concurrency, asynchronous tasking

1. INTRODUCTION

HeteroStreams [1] (hStreams for short) is an Intel® open source (available at [4]) streaming library that aims to support task-concurrency on Intel-based heterogeneous platforms (CPU, and Xeon Phi™ families, potentially FPGAs). Here, task means a serial or parallel function that works on some data. The task concurrency can be exploited across nodes as well as within a node. hStreams automatically overlaps independent computations with communications, (e.g., computations on independent tiles in a tiled solution), relieving the user from the complexity of asynchrony, offloading, pipelining, thread affinitization, and memory management.

In hStreams, the main execution units are called *streams*. These streams can be considered as FIFO queues holding tasks to be executed. Each stream has a source side (where tasks get enqueued) and a sink side (where tasks get executed) and hStreams offers a platform-agnostic interface[2] to communicate with the sink side promoting portability. The computation, communication, and synchronization tasks are enqueued in the streams and executed in the order they have been enqueued except that if a communication task does not depend on of a preceding computation task, those get automatically overlapped. Upon initialization, hStreams automatically divides available system resources (compute and memory) into domains to be used by the streams created by the program. To run two tasks in parallel, one need to place them in two different streams.

So far, hStreams has been used to implement Matrix multiplication, Cholesky, LU factorization, and a number of important ISV apps [1]. This poster opens up the breadth of algorithms implemented using hStreams. We present the first hStreams enabled algorithm with applications in Machine learning and Bioinformatics: the *K-nearest neighbor (KNN)*. KNN is a popular machine learning algorithm that solves the following problem: given n labeled training data point, and a query data point, find k data points from the given training data points which are closest to the query data point in terms of euclidian distance in the pertaining feature space. Then, label the query data point by taking majority votes from the labels of the discovered k nearest neighboring points.

One can naïvely solve the KNN problem by taking each query point and computing distances from the query point to all n training data points, and then picking the k points closest to the query point. This naïve algorithm requires $\Theta(kn^2)$ computation cost and $\Theta(n+k)$ space, suggesting that KNN is a compute-bound application. In practice, many KNN algorithms use a high-dimensional tree data structure (such as kd-tree or buffer kd-tree [2, 3]) to arrange the training data spatially based on their distances in the feature space, which asymptotically reduces the cost of KNN search.

2. THE BASE ALGORITHM FOR KNN

In this section, we briefly explain the base implementation that we used to create our hStreams-based KNN app. We started with an implementation originally developed by the Intel® Parallel Computing Lab [2]. We used a highly optimized (well vectorized and parallelized) shared-memory version of this code that uses a kd-tree data structure to arrange the training data set into a tree format by recursively partitioning the data across each feasible feature space one by one, until the data set size in a given tree node becomes small enough in terms of computation cost. The tree construction phase is considered as the *training* phase that takes $\Theta(n \log n)$ time where n is the size of the data set.

Once the training phase ends, the classification phase starts where for each query point, k nearest neighbors are found by traversing the kd-tree which takes $\Theta(k \log k + \log n)$ where the $\Theta(k \log k)$ comes from the creation and manipulation of

[1] Intel, Xeon and Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries

[2] While the hStreams interface is platform-agnostic, the current implementation of hStreams available at [4] only supports Xeon® processor and Xeon Phi™ coprocessors.

SC '16 Nov 13–18, 2016, Salt Lake City, UT, USA

DOI: 10.475/123_4

a k-heap to store k-nearest neighbors. The classification task is embarrassingly parallel with respect to all query points.

2.1 Performance

The following section presents some performance analyses[3] of the base implementation. It sets up our expectations regarding how an hStreams-enabled version might perform compared to the original highly-optimized shared-memory implementation. We used 24-core 2.70GHz Xeon® CPU E5-2697 v2, 61-core Knights Corner C0(QS-7120A)(KNC), 72-core Knights Landing D-30tiles-8MCDRAM (KNL) to run the programs. Intel® icc 2016 was used to compile the programs, and MIC_USE_2MB_BUFFERS was set to 64k.

Run time.

- Training: The base KNN training code runs 1.7× faster on KNL than on KNC, but 3.6× slower than 24-core Xeon®.
- Classification: The base KNN classification code runs 1.32× faster on KNL than on KNC and 1.27× faster than 24-core Xeon®.

Scalability.

- Classification algorithm is highly scalable, shows improvements with all hyper threads being used.
- Training algorithm has complex control flow and can only utilize the available physical cores efficiently (i.e., does not scale with hyper threads).

Since Xeon® cores are better optimized to efficiently execute complex and sequential control flows (e.g., recursion or depth-first traversal of trees), they are more efficient for executing the recursive kd-tree construction (i.e., training) phase. However, the multicore machine with 24-core Xeon® CPU does not offer enough parallel execution units to exploit all available parallelism that exists in the classification phase of KNN with thousands of query points. On the other hand, Xeon Phi™ families (KNC and KNL) with many small and simple cores are not as efficient in handling complicated training phase with recursive and sequential control flows, and but they offer more parallel execution units and are able to efficiently exploit the available parallelism in the classification phase of KNN. Therefore, one can get the best of the both worlds by using Xeon® for the training and Xeon Phi™ for the classification phase of the KNN problem which can be easily achieved using hStreams.

3. ENABLING KNN USING HSTREAMS

hStreams supports offload model where there is a source side (e.g., mainly a Xeon® (also called host)) and a sink side (receiving end of an offload operation, can be a Xeon® or Xeon Phi™). Although, the current hStreams implementation supports only Xeon and KNC as sink targets, hSreams could be extended to support other sink targets as well.

[3]Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance

In our hStreams enabled KNN algorithm, we executed the training phase (construction of kd-tree) on Xeon® and offloaded the classification part (either partially or completely) to the sink side. The constructed kd-tree (i.e., model built from the training data) was transferred to the sink side before the classification starts. The overall memory allocations and data transfer to the sink side cost additional 0.6 to 1 second with hStreams.

Since we can run the K-nearest neighbor search problem for each query data point in parallel, we distributed the total work among the streams in hStreams by dividing the list of query data points into segments (similar to tiling). We created s streams to run this classification task on those segments in a round-robin fashion and the execution responsibility of segment i was assigned to the stream $i\%s$. hStreams automatically overlapped computations with communications which helped in hiding data transfer latency.

4. USING HSTREAMS, WE GET THE BEST OF BOTH WORLDS

Table 1 shows some performance results of the hStreams enabled version of KNN. We used same systems as we used in section 2.1. The table shows both the runtime for native runs on Xeon® (host) and Xeon Phi™ (KNC) and performance using hStreams when only one KNC card is used for executing the entire classification phase, or when a small fraction of the classification task is executed on host (Xeon as a sink) and the rest on one KNC card. The reasoning presented to explain performance behavior in section 2.1 holds here too. By choosing host for training and KNC (or KNC + host) for classification, hStreams was able to finish the entire program in the shortest amount of time. Table 1 shows that the hStreams heterogeneous implementation gets speedup in the classification phase and the speedup increases with the training data size. The training phase can also be distributed across the host and sink using hStreams, following a similar approach as shown in [2], especially for massive training data. Automatic overlapping of computation and communication is likely to improve performance in those case significantly which is a future possible extension of this work. Being open-source, contributions from open developer and user community would greatly enhance hStreams's ability to serve our need.

Dataset 1: 10M data points, 15 features k = 51, 500K query				
Phase\ Platform	Native		hStreams	
	KNC (240 thd)	Xeon (24 thd)	KNC (240 thd) as sink	Xeon (22 thd) + KNC (240 thd) as sink
Training	64.5s (**4.6**s with 61thd)	**1.01**s	**1.02**s(on host 24thd)	**1.06**s on host 24thd)
Classification	**37.9**s	116.7s (**54.2**s with 48thd)	**36.7**s (on sink)	**36.7**s (on host + sink)
Dataset 2: 2M data points, 15 features k = 51, 500K query				
Training	62.5s (**1.6**s with 61thd)	**0.19**s	**0.18**s (on host 24thd)	**0.19**s (on host 24thd)
Classification	**7.3**s	21.0s (**7.1**s with 48thd)	**7.1**s(on sink)	**7.0**s (on host + sink)
Dataset 3: 5M data points, 18 features k = 51, 500K query				
Training	63.8s (**3**s with 61thd)	**0.5**s	**0.5**s (on host 24thd)	**0.5**s (on host 24thd)
Classification	**11.2**s	31.1s (**10.8**s with 48thd)	**11.2**s (on sink)	**10.9**s (on host + sink)

Figure 1: Performance comparison of KNN implementations (with/without hStreams) on Xeon® and Xeon Phi™. Number of query points for all datasets was 500K.

Acknowledgment

Thanks to Chris Newburn, Marcin Sasinowski for their supports with hStreams. Thanks to Mostofa A. Patwary for sharing the original KNN code and data.

5. REFERENCES

[1] C. J. Newburn, G. Bansal, M. Wood, L. Crivelli, J. Planas, A. Duran, P. Souza, L. Borges, P. Luszczek, S. Tomov, and others. Heterogeneous streaming. In IPDPSW, 2016.

[2] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Liu, P. Sadowski, E. Racah, S. Byna, C. Tull, B. P. Wahid, and P. Dubey. PANDA: Extreme-Scale Parallel K-Nearest Neighbor on Distributed Architectures. In IPDPS, 2016.

[3] F. Gieseke, J. Heinermann, C.E Oancea, C. Igel. Buffer kd Trees: Processing Massive Nearest Neighbor Queries on GPUs. In ICML, 2014.

[4] hStreams repo: https://01.org/hetero-streams-library.