

# Distributed/Cluster Computing for Data Stream Mining: Draft Notes

Vladimir Petko

February 2, 2015

## Abstract

The thesis is focused on elucidating GPU computing feasibility for clustering tasks

## Introduction

In real world applications such as industrial monitoring, sensor networks, financial data generate large unbounded streams of data which has to be processed with pre-defined response time. The processors capabilities limit the bandwidth of the stream which can be processed. Parallelizing processing algorithm will increase maximum bandwidth while maintaining the response time requirement.

## Previous Work

### General Purpose GPU Computing Frameworks

The tree of the GP-GPU technologies is presented in the Figure 1.

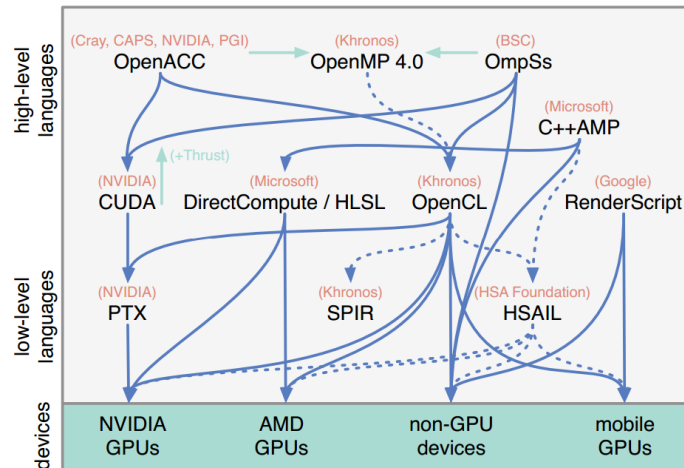


Figure 1: GP GPU technologies tree [13]

**GPU-specific Languages** GPU-specific languages provide a programming model consistent with the GPU hardware implementation. Modern GPUs implement SIMT (Single Instruction - Multiple Thread) execution model (NB. AMD/NVIDIA desktop GPUs) The memory on GPU is divided in 3 tiers

- *private/register* - private to the current thread

- *local* - shared within a *threadblock*
- *global* - accessible by every thread

Each of the listed languages provides following abstractions:

- *Kernel* - a unit of execution
- *Thread* - a single unit of processed data
- *Threadblock* - a group of *threads* sharing same *kernel* and *local* memory.

The unit of scheduling is called *wavefront* in AMD terminology or *warp* in NVIDIA and typically consists of 32 threads on NVIDIA and 64 on AMD hardware. The GPU chip is equipped with a number of SIMT cores which execute same instruction for each *warp*. Divergence of control results in underload of the processing units and reduces performance. The branching should be reduced to wavefront granularity to avoid wasting execution cycles[15](better citation?). Should be noted that the wavefront size is a hardware specific feature which should be specified at the runtime.

- CUDA - A programming language for NVIDIA hardware
- OpenCL - Open standard from Khronos group
- RenderScript - Android GPU computing component
- DirectCompute/HLSL - Microsoft Parallel Computing.

### Low-Level Languages

- PTX - NVIDIA CUDA Assembly
- SPIR - Standard Portable Intermediate Representation from Khronos group. OpenCL assembly.
- HSAIL - HSA Foundation assembly language

**High Level Languages** OpenACC and OpenMP are high level parallel programming frameworks that specify a set of annotations, environment variables and library routines for shared memory parallelism in C/C++ and Fortran programs[?][?]. Microsoft C++ AMP[1] is a C++ library which enables parallel computations for CPU and GPUs (using Microsoft DirectX Shading Language) Rootbear GPU compiler provides a transparent compilation of Java code into CUDA[?]. Aparapi provides a way to generate OpenCL kernel code from Java, theoretically allowing code which can be executed on CPU and offloaded to GPU if needed[?]. Project Sumatra is a OpenJDK project which focuses on development of the Hotspot virtual machine capable of offloading JDK 8 Stream API[?] computations to the GPU[?].

## Limitations

**Input Size** The massively parallel nature of GPU platforms require a certain amount of data to be passed to the kernel to achieve maximum performance. Table 1 shows execution time of a kernel which assigns index to each array element  $X_i = i$  on AMD A8-7600. The execution time starts to increase when input size is above 1024 and remains constant for lower values. To maximum performance on AMD A8-7600 will be achieved when input size will exceed 1024 elements.

Global Size	Execution Time ( $\mu$ sec)
256	8
512	8
768	8
1024	8
1280	9
2560	9
2816	10
3072	10
3584	11
4608	11
4864	12

Table 1: Input Size and Execution Time

**GPU Memory Size and Host-GPU Transfer** The discrete GPU requires transfer of data from the host to the GPU memory which adds additional overhead to the computations and requires task partitioning according to the memory specification of GPU[14]. Memory transfer is a bottleneck for Aparapi and its developers allow explicit memory management[?]. This effectively reduces framework which promises CPU-GPU interoperability to the Java wrapper of the OpenCL API.

**Kernel Launch Time** There is a constant time needed to setup kernel launch which might offset any gain from parallelization if the data can be processed sequentially faster.(NB. Amdahl's law)

## HSA Platform

Heterogeneous System Architecture platform specification introduces a set of requirements that allow both GPUs and CPU share same memory space and synchronize execution[2]. The introduction of shared memory addresses memory transfer limitation and allows to use all available host memory as a kernel input data. This makes HSA platform an interesting choice for the implementation

of online algorithms as it is supposed to have lowest latency of all GP GPU frameworks. *Software Available:* At the moment (Feb 2014) there is OpenCL-HSAIL compiler available[?] and a Linux-based runtime environment[?].

## HSA Memory Model *TODO*

**HSA Queues** Similiar to OpenCL[?] HSA uses queues to schedule code execution. A HSA *queue* is a ringbuffer which contains *packets* with either call or synchronization parameters. The queue maintains two indexes - read index and write index. Write index is modified by the user and used to submit packets to the queue. The read index is updated by the packet processor whenever the packet is taken for execution. As soon as packet is written to the queue the ownership is taken by the HSA packet processor and it may change packet contents at any time[2]. *TODO: Continue*

**HSA Signals** HSA uses *signals* to perform synchronization between host and kernels being executed or to signal completion of the task. *TODO: Continue*

**Implementation Notes** Scenario: submit packet, wait for completion, submit another one yields 8  $\mu$ sec per packet and submit  $N$  packets with a shared completion flag, wait for the completion flag become updated by  $N$  results in 193  $\mu$ sec per packet. There is a constant time needed to setup kernel launch, e.g. for AMD A8-7600, it is 6  $\mu$ sec using HSA.

## Machine Learning and General Purpose GPU Computing

### k-Nearest Neighbours

k-Nearest Neighbours method is a non-parametric method used for the classification and regression. It computes a given instance distance to the examples with the known label and either provides a class membership for the classification which is a class most common among nearest neighbours or an object property value which is an average of the nearest neighbours. The error rate bound by the twice the Bayes error if the number of examples approaches infinity[?]. The naive approach computes distance to each example and has computational complexity  $O(N^d)$  where  $N$  - number of examples and  $d$  - cardinality of the example. [?] The method optimizations deal with organizing the search space to reduce number of distance calculations. Examples would be branch and bounds [?] methods - [list trees], and approximate methods, e.g. - Locality sensivity hash.[?] In relation to data stream classification there are two problems:

- Forward k-NN - for an given sliding window of examples find classification of the variable query
- Reverse k-NN - for a given fixed query form a window of examples nearest to it. This approach is discussed in Efficiently Processing Continuous k-NN Queries on Data Streams[5].

There is a number of implementations of the offline k-NN algorithm for GP-GPU: brute force approach[6][9][8][14], kd-tree[7][17], approximate [11][10]. The brute force implementation consists of distance calculation and sorting phase. The distances to the query are computed as a vector-matrix multiplication or if several queries are processed at once as a matrix-matrix multiplication. GPU implementation of those routines is available as a part of cuBLAS library[?]. Sorting phase finds k nearest of all the computed distances[14]. Sismanis et. al[14] provide time complexity of reduced sort algorithms and evaluates their performance on GPU, proposes to interleave distance calculation and sorting phases to hide latency - the data for the distance calculation should be offloaded to GPU while it performs the sorting phase. The input data in the brute-force approach is partitioned according to the GPU memory capabilities and does not use examples's spatial information. The kd-tree approach presented by Gieske et. al focuses on parallel execution of nearest neighbour queries in a lazy fashion. The query points are accumulated in the leaf nodes of the kd-tree until enough of them is present and then processed as batch. This solves an issue of the GPU underutilization and low performance if leaf nodes are processed sequentially for each example[7]. The parallel kd-tree construction is explored by Zhou et. al[17].

*TODO: Continue*

## Problem Statement

*TODO*

## Algorithm Implementation

### k-Nearest Neighbours

**Brute-Force Approach** The algorithm maintains a sliding window of examples, calculates distance to the query point for each example and sorts them according to the least distance selecting nearest  $k$  neighbours.

**Sliding Window** The sliding window is implemented as a FIFO cyclic buffer. The OpenCL implementation uses partial mapping of the buffer to reduce memory transfers.

**Distance Calculation** The distance calculation between query vector and sliding window is a vector by matrix multiplication operation. For the dense matrices the naive implementation performs a serial computation of each distance:

```
// input - query vector
// samples - sliding window, matrix of window-size instances
// ranges - min/max values for each attribute
```

```

// result - resulting distance vector
// window_size - size of the window
// element_count - number of attributes in each instance
// numerics_size - number of numeric attributes
distance(double* input, double* samples,
{
    forall result_offset ( 0 < result_offset < window_size) do in parallel
        int vector_offset = element_count * result_offset;
        double point_distance = 0;
        double val;
        double width;
        int i;
        for (i = 0; i < numerics_size ; i ++ )
        {
            double2 range = ranges[i];
            width = ( range.y - range.x);
            val = width > 0 ? (input[i] - range.x) / width
- (samples[vector_offset + i] - range.x)/width : 0;
            point_distance += val*val;
        }

        for (; i < element_count; i ++ )
        {
            point_distance += isnotequal( input[i] , samples[vector_offset +
        }
        result[result_offset] = point_distance;
    }
}

```

The optimal implementation depends on the size of the window and number of attributes present[16]. For the small instance size ( $\leq 100$ ) and windows less than  $10^4$  elements naive implementation will provide the best solution. Best all around distance calculation should apply different strategies depending on the window size and number of attributes.[16].

**Selection** Alabi, et.al evaluated different selection strategies based on bucket sort algorithm and Merrill-Grimshaw implementation of radix sort[3].

This implementation provides only select based bitonic sort[4] algorithm which is suboptimal as we can only truncate sorting at the last stage of the algorithm.

*TODO:* The work needs to provide several alternative selection strategies. While Merrill's algorithm may be too complex for implementation, we might use k-bucket Selection[3] to provide alternative selection strategy.

**KD-Tree based k-Nearest Neighbours Search** The KD-Tree structure was implemented as a sequentially updated structure. The distance calculation for the leaves of KD-Tree was offloaded to GPU. The OpenCL implementation

required transfer of the leaf nodes contents to the GPU memory and has shown performance worse than serial implementation due to the transfer overhead *Need table*

Some ideas to try: when updating tree do not remove instances - each instance has a timestamp and distance calculation assigns max distance when instance is too old. Old instances are replaced by the new ones (see FIFO buffer in brute-force implementation). The NN-Search can be batched - to reduce kernel launch overhead accumulate N query instances at the leaf node and only then perform distance calculation.[?]

**LHS based k-Nearest Neighbours Search** *TODO*

## **Stochastic Gradient Descent**

The implementation relies on the fact that the weights vector can be updated without locking since the training instances are sparse and each instance contributes to a different part of the weights vector[12].

## **Experimental Results**

*TODO* Good benchmarks: OpenCL brute-force kNN, HSA brute-force kNN  
Bad benchmarks: OpenCL KD-Tree

### **k-Nearest Neighbours**

*TODO*

### **Stochastic Gradient Descent**

*TODO* Good benchmarks: HSA Hogwild! with batching Bad benchmarks: HSA Hogwild! without batching

## **Conclusions**

*TODO*

## **Future Work**

*TODO*



# Bibliography

- [1] C++ amp overview.
- [2] Hsa platform system architecture specification, 2014.
- [3] Tolu Alabi, Jeffrey D. Blanchard, Bradley Gordon, and Russel Steinbach. Fast k-selection algorithms for graphics processing units. *J. Exp. Algorithms*, 17:4.2:4.1–4.2:4.29, October 2012.
- [4] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.
- [5] C. Bohm, Beng Chin Ooi, C. Plant, and Ying Yan. Efficiently processing continuous k-nn queries on data streams. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 156–165, April 2007.
- [6] A. Dashti. Efficient computation of k-nearest neighbor graphs for large high-dimensional data sets on gpu clusters.
- [7] Fabian Gieseke, Justin Heinermann, Cosmin Oancea, and Christian Igel. Buffer kd trees: Processing massive nearest neighbor queries on gpus. In *Proceedings of The 31st International Conference on Machine Learning*, page 172180, 2014.
- [8] Kimikazu Kato and Tikara Hosino. Solving k-nearest neighbor problem on multiple graphics processors. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 769–773, Washington, DC, USA, 2010. IEEE Computer Society.
- [9] Quansheng Kuang and Lei Zhao. L.: A practical gpu based knn algorithm. In *In Proceedings of the Second Symposium on International Computer Science and Computational Technology (ISCST 09) (Dec. 2009)*, Academy Publisher, pages 151–155.
- [10] Tieu Lin Loi, Jae-Pil Heo, Junghwan Lee, and Sung-Eui Yoon. Vlsh: Voronoi-based locality sensitive hashing. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 5345–5352, Nov 2013.

- [11] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.
- [12] Feng Niu, Benjamin Recht, Christopher R, and Stephen J. Wright. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.
- [13] C. Nugteren. Improving the programmability of gpu architectures, 2014.
- [14] N. Sismanis, N. Pitsianis, and Xiaobai Sun. Parallel search of k-nearest neighbors with synchronous operations. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–6, Sept 2012.
- [15] T. Aila S.Laine, T. Karras. Megakernels considered harmful: Wavefront path tracing on gpus.
- [16] Hans Henrik Brandenborg Sørensen. High-performance matrix-vector multiplication on the gpu. In *Proceedings of the 2011 International Conference on Parallel Processing, Euro-Par’11*, pages 377–386, Berlin, Heidelberg, 2012. Springer-Verlag.
- [17] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):126:1–126:11, December 2008.