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

**A thesis**
**submitted in partial fulfilment**
**of the requirements for the Degree**
**of**
**Master of Science**
**at the**
**University of Waikato**
by

**Vladimir Petko**

**University of Waikato**
**2016**

# Abstract

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

# Acknowledgements

# Contents

# List of Figures

# List of Tables

# 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.

# Chapter 1

# General Purpose GPU Computing

## Introduction

The modern Graphical Processing Units (GPU) greatly outpace CPUs in arithmetic throughput and memory bandwidth for data-parallel tasks. Since 2001 the efforts were made to port data parallel algorithms to GPUs - first using shader languages such as HLSL, then with the release of Nvidia G80 in 2006 using extensions to C programming language - CUDA[12]. Presently there is a number of programming frameworks targetting specifically GPU architecture such as CUDA[39], OpenCL[11], RenderScript[16], DirectCompute[5] and more generic parallel-processing frameworks such as OpenMP[14] and AMP[3] which provide GPU backend as one of the targets. The differences in the hardware architecture between CPU and GPU is reflected in the programming model of the traditional GPU-specific languagages which contain hardware architecture specific language constructs. This chapter provides an overview of GPU architecture, most known programming frameworks, lists limitations of the traditional GP GPU programming,discusses OpenCL 2.0 standard which addresses some of the limitations and Heterogenous System Architecture (HSA) an optimized platform architecture for OpenCL 2.0.

# GPU Architecture

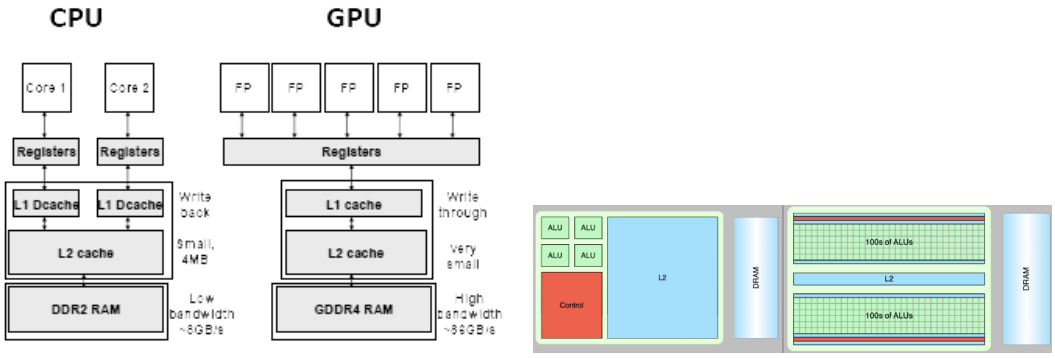


Figure 1.1: CPU versus GPU hardware architecture. Reproduced from NVIDIA GPU ARCHITECTURE and CUDA PROGRAMMING ENVIRONMENT by Alan Tatourian[51]

The main differences between modern CPU and GPU architectures are the level of paralellism and ability to directly address tiered memory. Modern CPU with 2 hex-cores support maximum of 12 threads (24 with hyperthreading) and minimal unit of execution for the NVIDIA GPU (called *wavefront*) is 32 threads. Modern GPUs implement SIMT (Single Instruction - Multple Thread) execution model (AMD/NVIDIA desktop GPUs) first introduced by NVIDIA in G80 model[12]. The single unit of scalar instructions called *kernel* is scheduled to execute in blocks of data-parallel threads on SIMT hardware. Each instruction in a block is executed in a lock-step. The control divergence is emulated by *masking* - the device executes instructions from both branches of the conditional statement[17][39]. The CPU thread is a heavy-weight entity which is centered around execution of a specific task for an extended period of time. Whenever CPU needs to preempt the thread, the register state is stored and another thread takes over. This makes a context switch a costly operation and operating systems attempt to minimize number of context switches per second. The GPU context switch is an extremely lightweight operation and is routinely used for the latency-hiding - whenever the wavefront is waiting on data, the GPU schedules another wavefront for execution. The GPU registers are private for each thread and are not reallocated until thread execution completes.

Modern CPUs provide a flat view of the operating system memory while GPUs divide memory in tiers based on the access speed:

- *private/register* - private to the current thread
- *local* - shared within a *threadblock*
- *global* - accessible by every thread

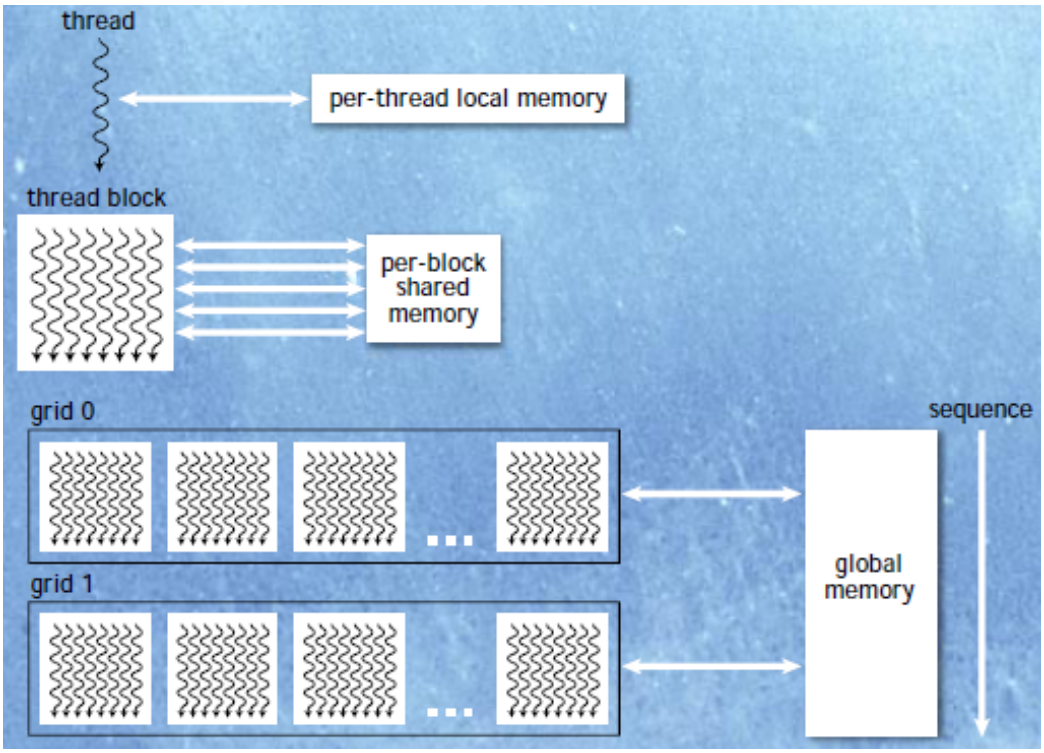


Figure 1.2: GPU Memory Tiers. Reproduced from NVIDIA GPU ARCHITECTURE and CUDA PROGRAMMING ENVIRONMENT by Alan Tatourian[51]

The GPU programming 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[47][39] It should be noted that the wavefront size is a hardware specific feature and the optimization should be performed at the run-time.

## General Purpose GPU Computing Frameworks

Existing General Purpose GPU (GP GPU) computing frameworks can be classified by the level of provided hardware abstraction: high-level frameworks integrate with existing high-level programming language such as Java to provide parallel computing capabilities without exposing any hardware details[13], traditional GPU languages such as CUDA[39] expose task scheduling and memory management giving the expert user fine-tuning capabilities, low level languages provide intermediate binary format compatible with multiple hardware targets. The tree of the GP-GPU technologies is presented in the Figure 1.

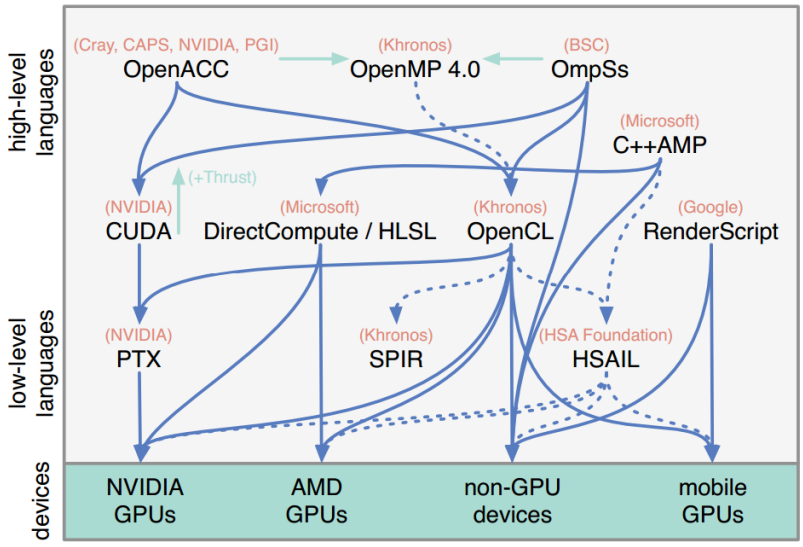


Figure 1.3: GP GPU technologies tree. Reproduced from C. Nugteren, Improving the Programmability of GPU Architecture, p. 21 [41]

## 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[**?**][14]. Microsoft C++ AMP[3] 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[43]. 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[1]. Project Sumatra is a OpenJDK project which focuses on development of the Hotspot virtual machine capable of offloading JDK 8 Stream API[10] computations to the GPU[13].

## GPU-specific Languages

GPU-specific languages provide a programming model consistent with the GPU hardware implementation.

- CUDA - A programming language for NVIDIA hardware based on C language. Kernels are expressed as C-functions for one thread with parallelism defined at run-time by specifying dimensions of execution grid and thread blocks[39]

- OpenCL 1.X builds upon ideas implemented in CUDA by adding device management APIs and providing hardware-agnostic programming specification. OpenCL gives *write once-run anywhere* guarantee but does not give any performance consistency guarantees across different hardware[49].

- RenderScript - Android GPU computing component which uses OpenCL with Java binding programming model - C-style kernels and Java-based control code. RenderScript does not provide any APIs for the *work-*

*group* size control in the bid to provide performance portability between different devices[16].

- DirectCompute/HLSL - Microsoft Parallel Computing.

## Low-Level Languages

The low level assembly representation is used to abstract compiler implementation from the actual hardware since each model or even revision may have a different instruction set. The translation is performed by *Just-In-Time* compiler before the kernel execution. Each vendor provides different low level specifications: NVIDIA CUDA uses Parallel Thread Execution and Instruction Set Architecture (PTX ISA)[15], Khronos Group specifies Standard Portable Intermediate Representation(SPIR)[18], and HSA Foundation specifies Heterogenous System Architecture Intermediate Language (HSAIL)[19].

## 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 1.1 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 | 256 | 512 | 768 | 1024 | 1280 | 2560 | 3072 | 3584 | 4608 | 4864 |
|---|---|---|---|---|---|---|---|---|---|---|
| Execution Time ($\mu$sec) | 8 | 8 | 8 | 8 | 9 | 9 | 10 | 11 | 11 | 12 |

Table 1.1: 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 mem-

ory specification of GPU[46]. Memory transfer is a bottleneck for Aparapi and its developers allow explicit memory management[1]. This effectively reduces framework which promises CPU-GPU interoperability to the Java wrapper of the OpenCL API.

**Kernel Launch** 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) It is impossible to schedule kernel execution from within the kernel itself requiring a mix of kernel and host code if several iterations are required.

## OpenCL 2.0

OpenCL 2.0 standard[11] introduces several features which attempt to address limitations of GPU programming:

- Shared Virtual Memory - both host and kernel code share same address space thus either hiding memory transfers (discreet GPU driver stack) or if backed by the hardware architecture such as HSA eliminate its need[19]
- Dynamic Parallelism - OpenCL 2.0 allows scheduling of kernels from within a kernel without host interaction reducing host CPU bottleneck.
- Pipes - pipes feature allows passing data from kernel to kernel without processing the whole input which allows to obtain the results of computation faster.

## HSA Platform

AMD introduced Heterogeneous System Architecture plaftorm as an optimized platform architecture for OpenCL 2.0. Its specifiction introduces a set of requirements that allow both GPUs and CPU share same memory space, synchronize execution using signals and atomics and schedule execution both from

GPU and CPU[19]. Task execution is performed by *agents* which represent CPU or GPU nodes. The task execution is scheduled via *queues* and synchronized using *signals*. HSA memory model guarantees sequential consistency for the correctly synchronized programs.

*Software Available*: At the moment (Feb 2014) there is a OpenCL 2.0-¿HSAIL compiler available[8] and a Linux-based runtime environment[7].

## HSA Queues

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[19]. Compared to traditional dispatch where the execution is scheduled via user-mode and kernel-mode driver layers the HSA dispatch intends to be lightweight and source-agnostic way of scheduling execution. The HSA Queues support work-stealing that is several HSA agents may be attached to the queue to share the workload.

## HSA Signals

HSA uses *signals* to perform synchronization between host and kernels being executed or to signal completion of the task. A *signal* is essentially a shared memory variable modified by the HSA agent. Runtime environment provides a way to check the value of the signal or wait for the specific value.

## HSA Memory Model

The sequential consistency was first defined by L. Lamport as “ ..the result of any execution is the same as if the operations of all the processors were

executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." Modern processors (ARM, x86, Itanium, POWER) introduce relaxed memory model to allow a range of the hardware optimizations to provide better performance by reodering load and store operations[36]. Platform specification states

> The HSA memory consistency model is a relaxed model based around RCsc semantics on a set of synchronizing operations. The standard RCsc model is extended to include fences and relaxed atomic operations. In addition HSA includes concepts of memory segments and scopes.

[19] Similiar to Java Memory Model[**?**] it guarantees sequential consistency for the correctly synchronized programs, that is 'synchronizing operations meet the requirements for sequential consistency within each scope/segment instance'[19]. The specification introduces several memory segments: "

- Global segment, shared between all agents.
- Group segment, shared between work-items in the same work-group in a HSAIL kernel dispatch.
- Private, private to a single work-item in a HSAIL kernel dispatch.
- Kernarg, read-only memory visible to all work-items in a HSAIL kernel dispatch.
- Readonly, read-only memory visible to all agents

" Each particular memory location is always associated with one and only one segment and all operations apply to only one segment with the exception of fence operations[19]. In addition to memory segments HSA memory model introduces *scopes* : wavefront, work-group, component and system. They can be used to reduce visibility of the memory operation compared to the default supported by the segment.The global segment may use any of the specified scopes and group segment is limited to wavefront and workgroup scopes[19]. Different workgroups accessing a global variable with the workgroup scope

will work with different instances of the variable. The write serialization only applies to the operations within the segment/scope that they specify.

### Implementation Notes

Sequential execution of several packets sometimes may be faster than submission of all packets and waiting for the barrier packet. For instance first scenario runs in 8 $\mu$sec per packet on AMD A8-7600 and second results in 193 $\mu$sec per packet. According to AMD support this is caused by the CPU going into power-saving mode while kernel is running. There is a constant time needed to setup kernel launch, e.g. for AMD A8-7600, it is 6 $\mu$sec using HSA.

## Conclusion

The modern specifications such as OpenCL 2.0 and HSA attempt to address some of the latency issues of the GPU programming by introduction of the shared memory and lightweight dispatch and data passing mechanisms. This work will focus on the evaluation of suitability of those technologies for the latency-sensitive data stream processing.

# Chapter 2

# System Architecture

The implemented data stream processing library provides a set of classification algorithms for Massively Online Analysis(MOA)[24]. The library uses existing linear algebra package ViennaCL[52] which allows multiple backends such as CUDA, OpenCL, CPU and extends it with the machine learning algorithms such as nearest neighbours search and stochastic gradient descent.

**MOA interface** The ViennaCL library is implemented in C++ and as such requires Java Native Interface[9] to be used to interface from the Java Virtual Machine. Java Virtual Machine manages its own memory space and garbage collector may move the data at any time. JNI provides two mechanisms to access the array data from the native code. First is copying - the java pointer is locked by the critical section and array content is copied to the native array. Second skips the copying and provides direct access to the java pointer. Both involve entering and exiting a critical section and impose significant performance loss due to the copying and locking overhead. Those costs can not be avoided but can be minimized by moving them to the instance creation/-modification time - the object constructor will call the native method which allocates the native data structures and moves data from the Java storage to the native one.

The alternative solution uses *java.misc.Unsafe* class to manipulate offheap memory directly. The native code allocates GPU shared virtual memory and

passes the pointer to the Java implementation. Java code uses *java.misc.Unsafe* methods to update data in parallel to training. Figure 2.1 shows the training process of the Stochastic Gradient Descent classifier. The instances are accumulated in batches on the main thread of execution, the batches are passed to the data transfer thread that handles CPU-GPU transfer and then the training thread is triggered to run GPU kernels. Whenever the evaluation ( *getVotesForInstance*) is called the threads process the last available batch and meet at the synchronization point stopping execution.

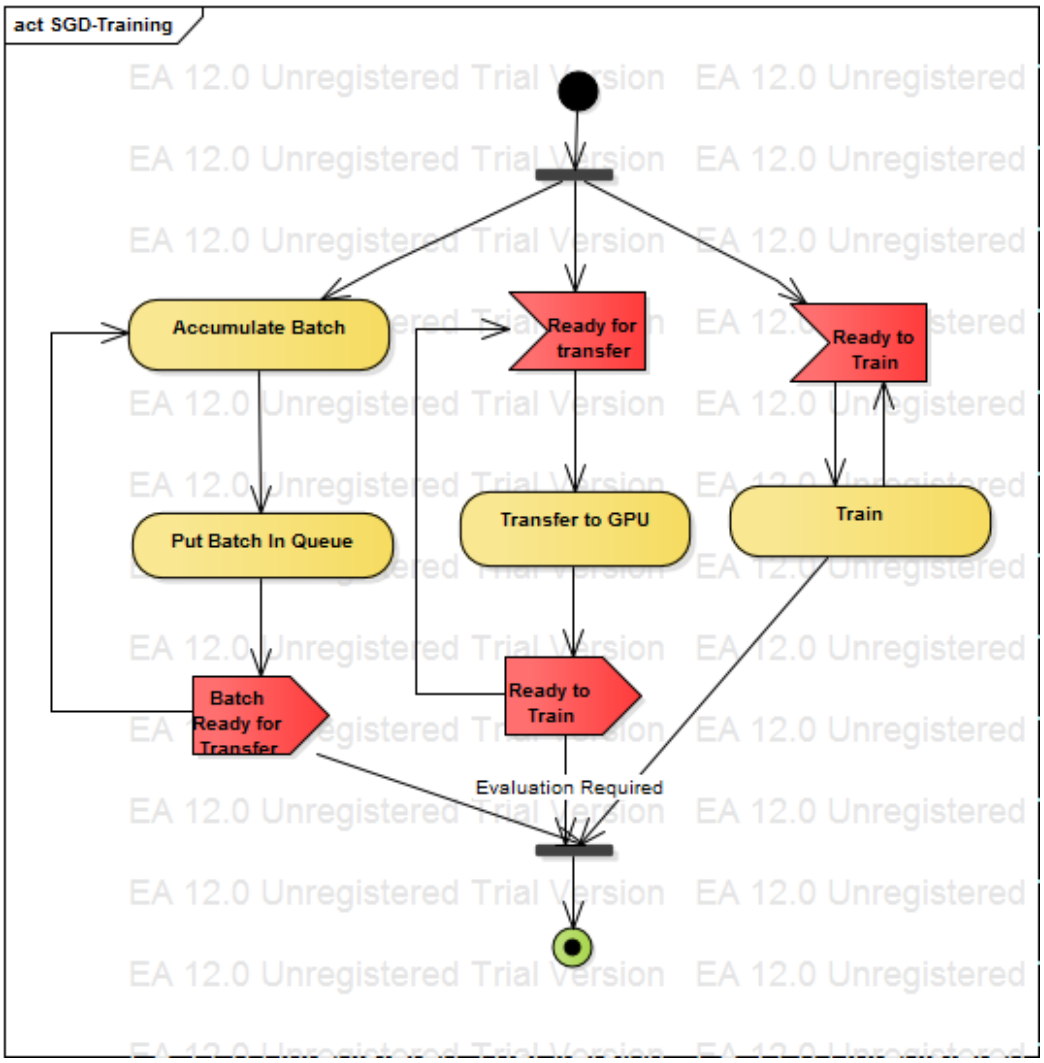


Figure 2.1: SGD classifier training activity.

**GPU Memory Limits** The library implementation stores the instance data as the native ViennaCL types.This implies that for GPU ViennaCL backends the data will be stored in GPU memory that may be insuffient for the larger

problems. In such case the partitioning will be used - the problem data is kept in the Java memory as collection of *weka.core.Instance* objects and offloaded to GPU-backed context on as needed basis. The *weka.core.Instance* class represents the attribute values as a vector of double precision numbers. Modern consumer GPUs provide far better floating point performance than double performance. For instance modern AMD GPUs have 8x scale that is floating point performance is 8x betweer than double one (R390, R290). NVIDIA GPUs have 32x scale *TODO citation*. There are several works that explore using fixed precision numbers to reduce memory requirements of the machine learning tasks[44][33]. The alternate precision implementation will impose the overhead costs of double to fixed/floating point conversion if the GPU classifier will be used for instance as a part of the meta-classifier ensemble.

**HSA Backend** This work adds a new HSA backend to the ViennaCL library based on the HSA Runtime[19]. This implementation is tuned for Kaveri AMD APU and uses the same set of OpenCL kernels as the OpenCL backend. In HSA backend the main system memory is transparently mapped to GPU and vice-versa, allowing to use vector or matrix element-addressing operations without first copying data to the CPU memory space.

**OpenCL 2.0 features** At the time of writing (driver version 1642.5) the OpenCL 2.0 features such as workgroup functions and device enqueue impose significant performance impact. Atomic locks do not result in significant performance impact. The library uses Shared Virtual Memory buffers to facilate easy switch between OpenCL and HSA backends. The ViennaCL types are contructed from *cl_mem* representation of the SVM buffers.

# Chapter 3

# k-Nearest Neighbours

## Problem Statement

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 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 complexity associated with distance calculation. Examples would be branch and bounds methods such as kd-tree that partition search space, and approximate methods, e.g. locality sensivity hash that simplifies the distance function by mapping instances into lower dimensional space preserving their pair-wise distances within the certain error margin.

**Exhaustive search** The exhaustive search approach consists of distance calculation and selection phase.The distances to the query a 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 avail-

able as a part of libraries implementing BLAS[6][4][52]. The selection phase finds nearest to the query out of all the computed distances. Sismanis et. al[46] 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.

**Space parititioning methods** The space paritioning techinques are widely used to limit number of distance calculations needed for nearest neighbour search. The most famous are $k-d$ tree, ball tree and cover tree.

The $k-d$ tree [**?**, **?**] is a balanced binary tree where each node represents a set of points $P \in \langle p_1 \cdot p_n \rangle$ and its children are disjoint and almost equals sized subsets of $P$. The tree is constructed top-down, the initial set of points is split along the widest dimension or using other criteria until the predefined number of points in child nodes is reached. The tree can be constructed in $O(nlogn)$ time and occupies linear space. Weber *etal*[53] have shown that space partitioning methods are being outperformed by the exact calculation at moderate dimensionality ( $n > 10$ ) and those methods result in full processing of the examples if number of dimensions is large enough. The $k-d$ tree requires $N \gg 2^k$ points to be more effective than exhaustive search.

The listing of the $k-d$ tree construction and nearest neighbours search pseudocode is shown in the Figure 3.1. The parallel $k-d$ tree construction on GPU utilizes breadth-first approach[54][45] - the $k-d$ tree is constructed top-down with the split criteria computed in parallel for all nodes at the specific level. The nearest neighbours search using k-d trees does not benefit much from the GP GPU parallelism due to the branch divergence and irregular memory access patterns[32]. The $k-d$ tree search approach presented by Gieske et. al focuses on parallel execution of nearest neighbour queries in a

```
1   tree_node create_tree(pointList, level)
2   {
3     int dim = select_dim(pointList); // select split dimension according to
4                         // pre−defined criteria, e.g. level mod total_dimensions
5     splitVal = select_split_value( pointList, dim); // select split value
6                                   // according to pre−defined criteria
7                                   // e.g. median value of point[dim]
8     left = {};
9     right = {}
10    for (point : pointList )
11    {
12          if (point[dim] > splitVal)
13              right += point;
14          else
15           left += point;
16    }
17    node = {
18      .location = splitVal,
19      .dim = dim,
20      .left = create_tree(left, level + 1),
21      .right = create_tree(right, level + 1)
22    };
23    return node;
24  }
25
26  void search(Heap nearest_neighbours, tree_node root, point p)
27  {
28    if (root.is_leaf())
29    {
30      nearest_neighbours.update(root);
31    }
32    else
33    {
34      split = root.location;
35      dim = root.dim;
36      if (p[dim] < split ) // search "closest" node
37        search(nearest_neighbours, root.left, p)
38      else
39        search(nearest_neighbours, root.right, p)
40
41      distance_to_split_plane = abs(split−p[dim]);
42      distance_to_point = abs(nearest_neighbours.furtherst_point()[dim]−p[dim])
43      if (distance_to_point >=distance_to_split_plane) // outer radius of NN heap
44      intersects the split plane
45      {
46        if (p[dim] < split )
47          search(nearest_neighbours, root.right p)
48        else
49          search(nearest_neighbours, root.left, p)
50      }
51
52
53    }
54  }
```

Figure 3.1: k-d tree construction and NN-search pseudocode

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 a batch. This solves an issue of the GPU underutilization and low performance if leaf nodes are processed sequentially for each example[32].

Ball tree[**?**, **?**] is a simplest and oldest data structure suitable for data represented in arbitrary metric space - data points $\mathbb{X}$ with defined distance function $d : \mathbb{X} \times \mathbb{X} -> \mathbb{R}^+$. The data points are bounded by the tree hierarchy of hyperspheres as opposed to hyper-rectangles in $k-d$ tree.The hyperspheres are allowed to overlap, the data points are assigned only to one hypersphere. At the time of writing there were no widely known GPU-based ball tree implementation though $k-d$ tree limitations such as branch divergence and irregular memory access for the tree traversal should apply to the ball tree algorithm as well.

Cover tree[23] is a $N-$ary tree also suitable for the data represented in arbitrary metric space. The tree is defined recursively - at the top level the initial point is arbitrary picked from the data set and covered by the ball with radius $2^i$. The $i$ is picked so that the ball covers all data points. On the next level the ball radius is $2^(i-1)$ and any points outside this radius generate their own cover balls. The initial ball is assigned as their parent. The process continues until each ball contains exactly one point. The cover tree satisfies following invariants:

- Nesting - once point is associated with node at level $i$ any level $j < i$ will have a node associated with this point
- Covering - the parent node at level $i$ covers all its child nodes
- Separation - the distance between centers of two distinct nodes at the same level is more than node radius.

The nearest neighbours query process is iterative. For the query point $p$ and $Q = Children(q) : q \in Q_i$ children of the node $Q_i$ we form a ball that includes its children satisfying following condition $Q_{i-1} = \{q \in Q : d(p, q) \leq$

$min_{q \in Q} d(p, Q) + 2^i\}$ [23].Thus on each step next level query radius to the distance to the center of the nearest ball plus its radius $2^i$. It should be noted that in practice the radius is redefined as $1.3^i$ as it yields better results[**?**]. The cover tree construction can be performed using breadth-first search approach to perform contstruction iteratively similiar to $k - d$ tree iterative construction[**?**]. The *nearest ancestor tree* is a simplification of the cover tree[**?**]. The *simplified cover tree* is a cover tree without nesting invariant. Simplified cover tree explicitly defines *level* invariant - each node has associated integer level and for any child node $q$ with parent $p$ $level(q) = level(p) - 1$. The *nearest ancestor tree* is defined as a simplified cover tree with nearest ancestor invariant - the maximum distance from the parent to the child nodes is minimized - for parent $q_1$ its sibling $q_2$ and a child node $p$, $d(q_1, p) \leq d(q_2, p)$[**?**]. The nearest ancestor tree requires rebalancing when a new node is inserted if this invariant is violated. The nearest ancestor tree construction requires significantly more distance calcuation than cover tree, but this is offset by better NN query in most cases[**?**]. The parallelisation approach for nearest ancestor tree construction - divide the data set, construct individual trees and merge the results.

The hypersphere bounding is used in *Random Ball Cover* method[27]. It provides a single level metric space cover. For the initial set of points $\mathbb{X} = \{x_1, \ldots, x_n\}$ random $O(\sqrt{n})$ points are selected to act as representatives and $L$ nearest points to them are attached to them. Radius $r$ is defined as distance between representative point and furthest point $l \in L$ attached points. The points may belong to more than one representative. The parameter $s$ defines distance needed to put a given point into list $L$. The parallel construction of the data model is simple and essentially consists of point selection and matrix multiplication followed by construction of $L$ via scan and scatter parallel primitives. Reduction is used to find maximum distance for each $L$. The method defines two algorithms:

- one shot - a distance is computed to each representative point, closest

one taken and $k$ points are selected from its list

- exact - *TODO*

*Random Projection Trees TODO*

**Approximate methods** The nearest neighbours search methods in high dimensional space provides little benefit over exhaustive search where an exact distance is computed to each point in the database[53][21]. The approximate methods provide means to overcome this limitation by solving the problem of finding neighbours whose distance from the query point are at most $c > 1$ times greater than distance to the closest neighbour. The approximate solution can be used to find exact one by computing distance to each approximate nearest neighbour and choosing closest ones. Locality Sensivity Hashing[34] is a method that captalizes on the idea that exist such hash functions $h(x), x \in \mathbb{R}^d$ that for points $p, q \in \mathbb{R}^d$, radius $R$ and approximation constant $c$

$$\begin{cases} \|p - q\| \leq R, P[h(p) = h(q)] \geq P_1 \\ \|p - q\| \geq cR, P[h(p) = h(q)] \leq P_2 \end{cases}$$

where probabilty $P_1 > P_2$. The LSH algorithm uses a concatenation of $M \ll d$ such functions to increase difference between $P_1$ and $P_2$[34]. Initially it was proposed to use Hamming distance as this function satisfies required properties[34]. Later it was shown that other families of hash functions such as $l_p$ distance[30], Jaccard coefficient [25][26], angular distance[28] are locally sensitive. The algorithms selects $L$ contactenations of the hash functions and uses them to transform input dataset points $v \in \mathbb{R}^d$ into lattice space $\mathbb{Z}^M$ storing them as $L$ hash tables. The exact query is performed by contactenating contents of the $L$ bins corresponding to the hash codes of the query and computing exact distance. The approximation is obtaining by stopping as soon as $k$ points in $cR$ distance from the query point is found.

A number of parallel LSH implementation has been developed to date[50][42].

*TODO*

**Random Projection and Fast Johnson-Lindenstrauss Transform** Random Projection is a hash function that according to Johnson-Lindenstrauss lemma preserves relative distances between data points.

Naive implementation of the Random Projection involves multiplication of the data point vector by the random projection matrix. The computational complexity of this operation is $O(kd)$. Fast Johnson-Lindenstrauss transform (https://github.com/gabobert/fast-jlt/tree/master/fjlt) achieves logarithmic speed in relation to d. It is well suited to the GP-GPU implementation due to the divide and conquer nature of the algorithm.

**Morton Code computation for variable number of dimensions** *TODO* The local neighbourhood of the data point in N dimensional space can be established via computation of the z-order. The z-order is a space-filling curve computed via sorting points according to their Morton Code. The Morton Code is computed as a bit interleave of the data point coordinates.

The computation of the morton code in original high dimensional space is both impractical due to the curse of dimensionality and computational complexity - the cost of the code computation and comparison is linear.

Thus it is possible to create a faster approximate k-NN algorithm by exploiting both logarithmic complexity of the fast Johnson-Lindenstrauss transform (logarithmic), reduced number of points needed for distance calculation and overall faster computation in lower dimensional space.

- The NN candidates of the query point can be determined by computing a hash function that preserves relative distance between points
- Morton code is a hash function computed by interleaving bits of the feature vector
- Morton code creates a space filling curve (z-order curve) with following property - it never doubles up (definition)
- Pick k nearest neighbour candidates by following z-order curve, compute

bounds of a hypercube and this hypercube will contain all possible NN candidates.

- Experiment and graphics - 2 dimensional task - good result, N¿2 - bad
- try orthogonal projections to limit search space - adaptable kd-tree in a sense

# Algorithm Implementations

**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 implementation performs a serial computation of the distance, each thread working on its own example. Since all the threads process attributes in exactly same order there is no wavefront divergence. This is not the case for the sparse matrices and this approach will cause GPU underutilization.

The optimal implementation depends on the size of the window and number of attributes present[48]. 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 windowsize and number of attributes[48]. The alternatives are presented in the Figure 3.2.

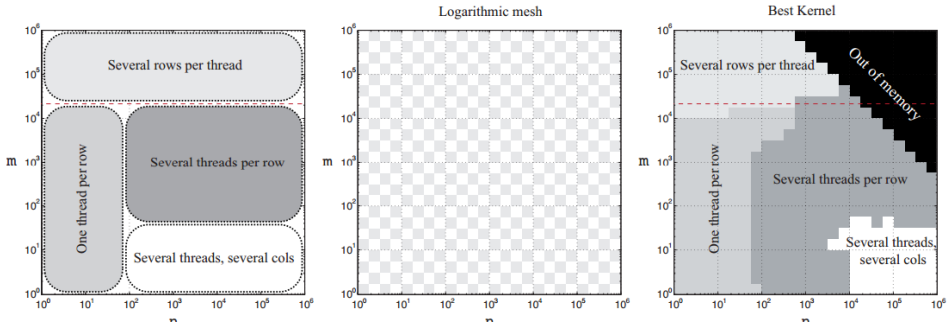


Figure 3.2: Left; Four matrix-vector multiplication kernels designed to perform well at different shapes m n of A. Middle; Tuning mesh. Right; Best kernel in practice. The dashed line indicates the minimum 21504 rows needed in A for full occupancy of the Nvidia Tesla C2050 card in a one-thread-per-row kernel. Note the logarithmic axes.. Reproduced from High-Performance Matrix-Vector Multiplication on the GPU by Hans Henrik Brandenborg Sørensen[48]

**Selection** Alabi, et.al evaluated different selection strategies based on bucket sort algorithm and Merril-Grimshaw implementation of radix sort[20].

The selection phase may be interleaved with the distance calculation to utilize both CPU and GPU cores and benefit from the better sort performance on low window sizes[38].

The work needs to provide several alternative selection strategies such as Merill-Grimshaw radix sort[37] or k-bucket Selection[20] to provide alternative GPU selection strategy.

Figure 3.3 shows measured performance of different selection strategies - merge sort from AMD Bolt library[2], bitonic sort similiar to reference AMD implementation and radix select based on Alabi, et. al. implementation[20]. The CPU sort and choose is a clear winner for small (¡65535) window sizes. The merge sort should be applied to sub $2^25$ windows and radix select (with data copy) should be used for larger window sizes. *TODO:* The work should investigate in-place radix select and device enqueue for radix select optimization. *TODO:* Radix select shows a semi-flat line up to $2^23$ window size. The implementation should be checked for excessive setup.

**KD-Tree based k-Nearest Neighbours Search** The KD-Tree nearest neighbours search is composed of parallel tree construction over fixed set of

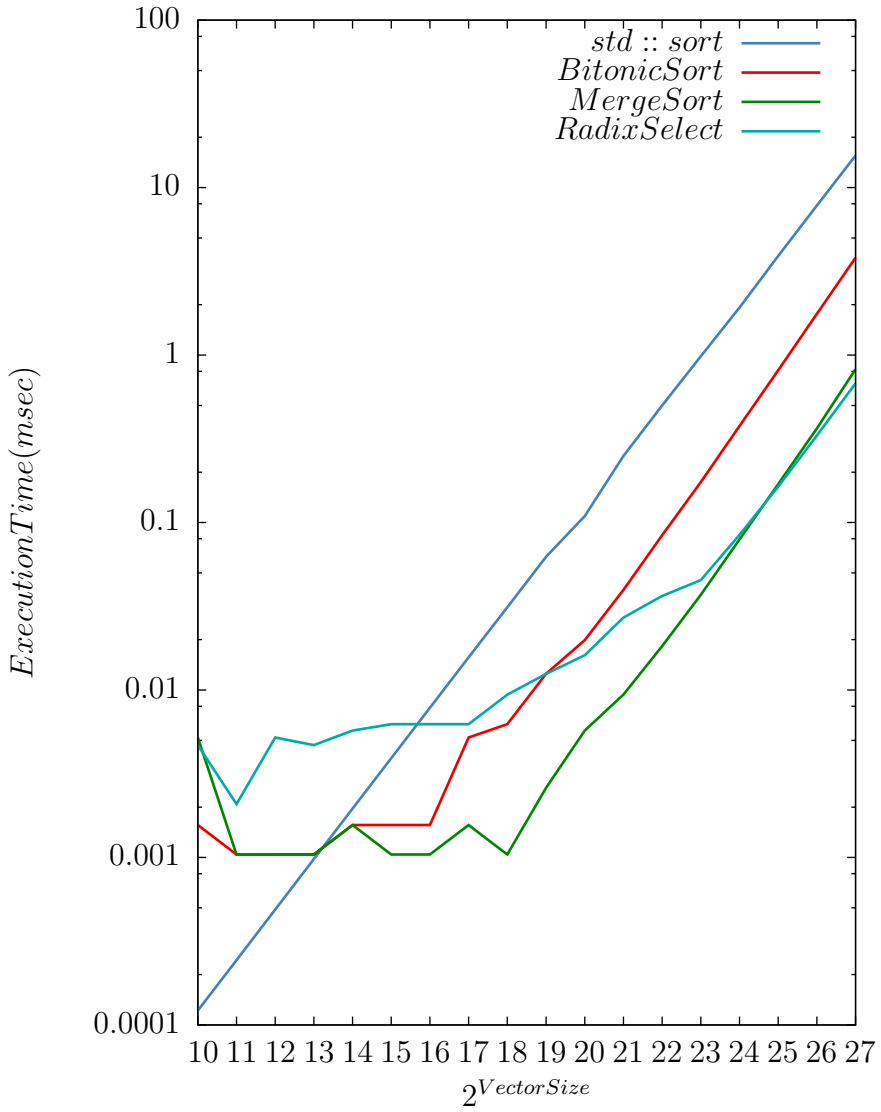


Figure 3.3: Selection Algorithm Performance for K=128. Test configuration GPU R9 390, CPU AMD A8-7600, AMD Catalyst version 15.20.

instances and evaluation.

**Parallel Tree Construction** The input to the algorithm is the matrix containing example instances and the desired tree depth. The tree nodes contain row numbers of the associated instances, split attribute, split value and ranges for the associated instances. The tree split is performed iteratively until the desired tree depth is reached. First the ranges are updated for each tree node using kernel based on Bolt max element kernel[2]. The kernel finds ranges for a given attribute for each of the nodes in a tree. A split dimension and point is chosen using the CPU routine. A mark kernel composes two flag

vectors - one for the left child, other for the right one, a scan is performed to compute offsets and then the child nodes are populated with the indices of the instances belonging to them. *TODO Picture*

**Evaluation** The evaluation uses same distance calculation kernel as the Naive Implementation. The evaluation can be either performed sequentially - a recursive algorithm similiar to the[24] implementation or several query instances can be scheduled at once. In this case the progress of each individual query instance is tracked by the bit vector containing the tree path processed so far, current node number and current found nearest neighbours. The algorithm alternates between invoking kernels for leaf node distance calculation and split plane distance calculation until all query instances are fully processed. *TODO Diagram*

**Fast Johnson-Lindenstrauss transform implementation** Figure 3.4 shows comparative performance of dense matrix multiplication for random projection and Fast Johnson-Lindenstrauss transform. For the selected hardware configuration the latter starts to outperform matrix multiplication starting from $N \geq 16384$. It should be noted that FLJT has lower memory requirements than *O(kd))* as it does not require to store dense transformation matrix and thus capable of projecting higher dimensional data on the same hardware.

**Approximate k-Nearest Neighbours Search** *TODO*

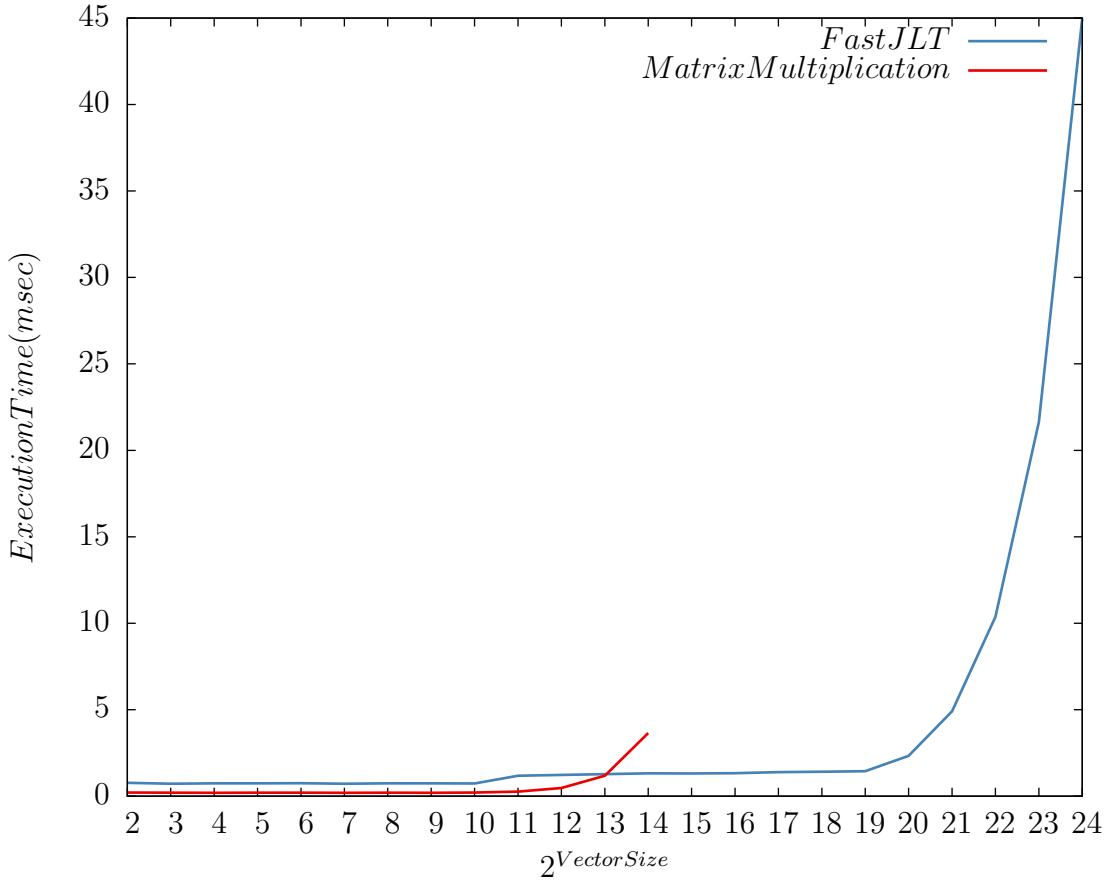


Figure 3.4: Fast Johnson-Lindenstrauss transform vs. dense matrix implementation using ViennaCL. Test configuration GPU R9 390, CPU AMD A8-7600, AMD Catalyst version 15.20.

# Chapter 4

# Stochastic Gradient Descent

## Problem Statement

The stochastic gradient descent is a iterative optimization method which approximates a gradient of the function by a gradient of a single example. The stochastic gradient descent is widely in many machine learning tasks and is a standard algorithm used in training neural networks.

The parallel implementations of the algorithm use either assumption that data in the training batch is loosely coupled and it is possible to loose some of the individual updates thus the batch can processed in parallel - Hogwild! algorithm [40] or extract independent data blocks from the traing batch - DSGD[31], FPSGD[29] or arbitrary divide the examples into parts processed in parallel and approximate the direction of the gradient descent [35] using weight update vectors produced by them.

## Algorithm Implementations

### Hogwild-based

This GP-GPU implementation for sparse instances uses Hogwild[40] approach to training. A sparse matrix containing the training batch is constructed and the training is performed on the assumtion that the data race between

individual updates to the same weight can be ignored.

The aglorithm is implemented as MOA[24] classifier with the native ViennaCL[**?**]-based part performing the GPU calculations.

The training batch is represented as a sparse matrix in Compressed Row Storage format[22], that is all non-zero elements of the matrix are stored sequentially in the elements vector in row major format, the row information vector contains indices of the first and last elements of each matrix row, and columns vector contains column indices of the matrix elements.

**OpenCL implementation** Figure 2.1 shows the training process - the classifier training and CPU-GPU data transfer are performed in parallel.

OpenCL implementation structure is shown in Figure 4.1. The device-side enqueue is used to launch kernels for individual instance update to simplify implementation. The device-side enqueue kernel call latency is *2x* higher than that of the host-side kernel enqueue (e.g. 1.68 $\mu$sec 0.91 $\mu$sec on the Spectre device) but its cost is hidden by the row processing time as the parent kernel would not return until all the child kernels has finished execution and $CLK\_ENQUEUE\_FLAGS\_KERNEL\_NO_WAIT$ flag is used to inform OpenCL runtime to start execution immediatly while parent kernel is still queueing row kernel. The atomic update operation is implemented using

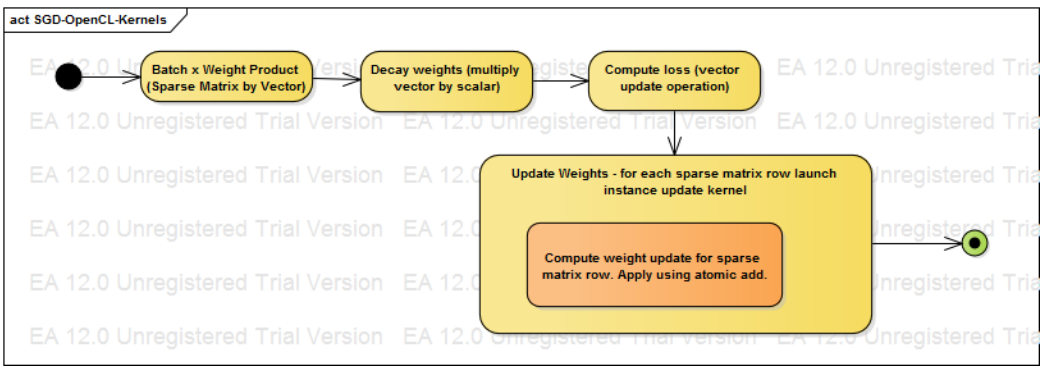


Figure 4.1: Kernel calls for SGD classifier training

$compare - and - swap$ primitive of OpenCL. It should be noted that the loop condition should be explicitly declared *volatile* to force runtime to re-evalutate the *compareandswap* expression.

**HSA implementation** The HSA implementation shares Java interface most of the native implementation with OpenCL SGD. The notable differences are:

- Memory management. The HSA implementation constructs ViennaCL container classes directly from the host pointers. The memory updates are fine-grained and global memory update done by kernel is immediatly visible by the host as opposed to the coarse-grained updates of OpenCL implementation.

- Device enqueue. The device enqueue is a largely untested functionality in the current HSA runtime and since the sparse matrix row information pointer containing indexes of row bounds is directly accessible by the CPU the individual row update kernels are scheduled from the host.

**Datatype Selection** The implementation can use either *double* or $float$ precision numbers for the calculations.

# Chapter 5

# Experimental Results

## k-Nearest Neihbours

### kNN Naive implementation

The first version of k-NN algorithm used Java implementation with JavaCL library. The results in Figure 5.1 provide overview of the achieved speedups.

### Stochastic Gradient Descent

The SGD implementation training speedup for sparse instances with the double data type is presented in the Figures 5.2, 5.3. The CPU sampling profiling showed that most of the time is spent inside buffer commit function responsible for copying data from WEKA instance class into the sparse matrix. As the GPU implementation uses batching to achieve training speedup the latency is *5-10x* worse than CPU. The profiling of the sample run on a Spectre device of OpenCL algorithm implementation with 1024 non-null attributes in an instance and 8192 instances in a batch shows that only *32%* is spent training the classifier, rest is spent in the CPU-GPU data transfer.

The sparse matrix OpenCL SGD implementation is bounded by the memory copy operation as shown by the profiling data and provides a limited speedup.

*TODO* Provide zero-copy (HSA) speedups *TODO* Provide float speedups

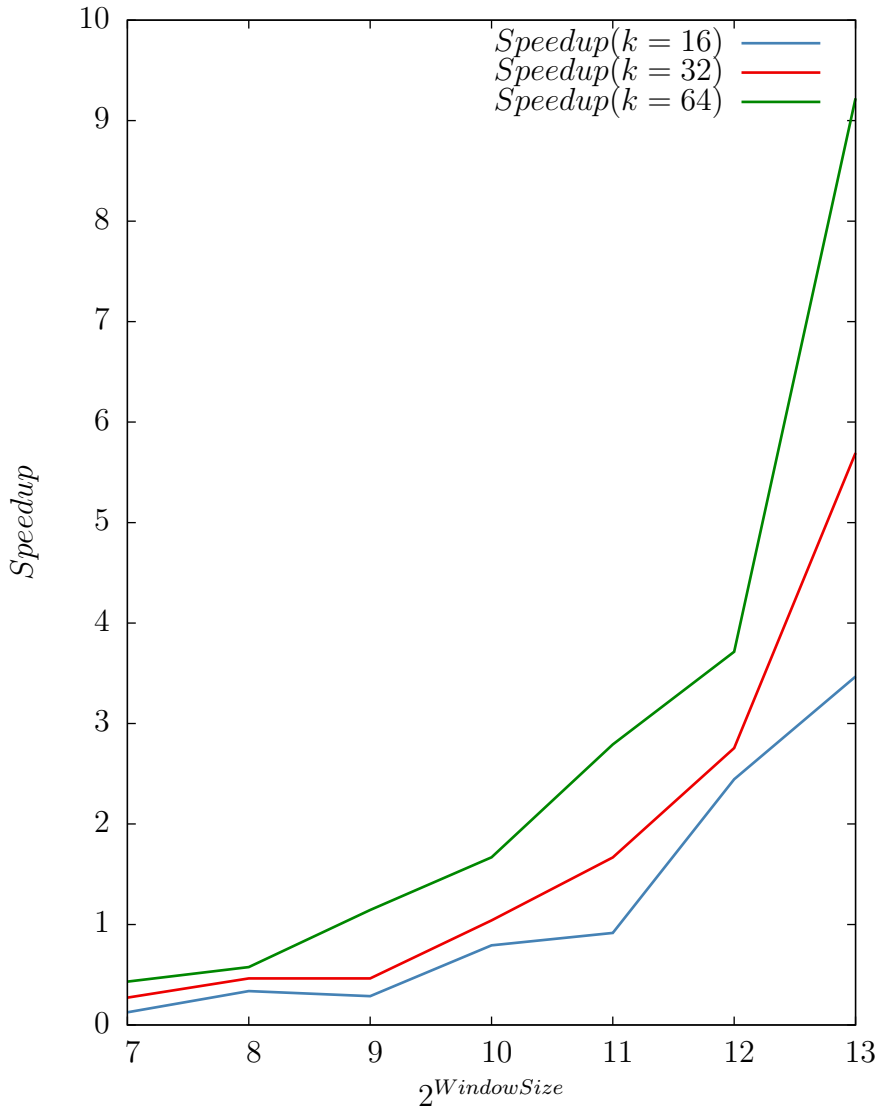


Figure 5.1: Naive KNN implementation (Radeon Mobility HD5730)

and impact on the accuracy of the algorithm for the known data sets.

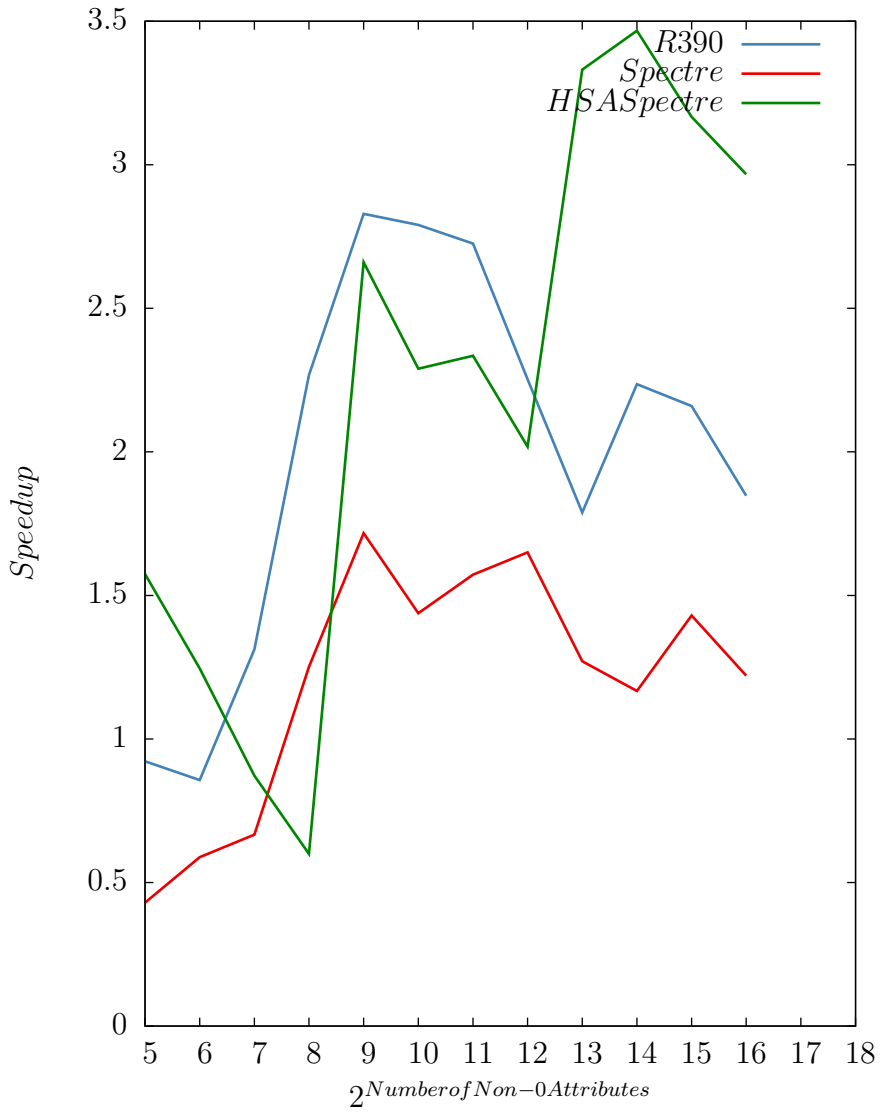


Figure 5.2: Stochastic Gradient Descent Training Speedup (AMD R9 390, Spectre vs MOA implementation on AMD A8-7600)

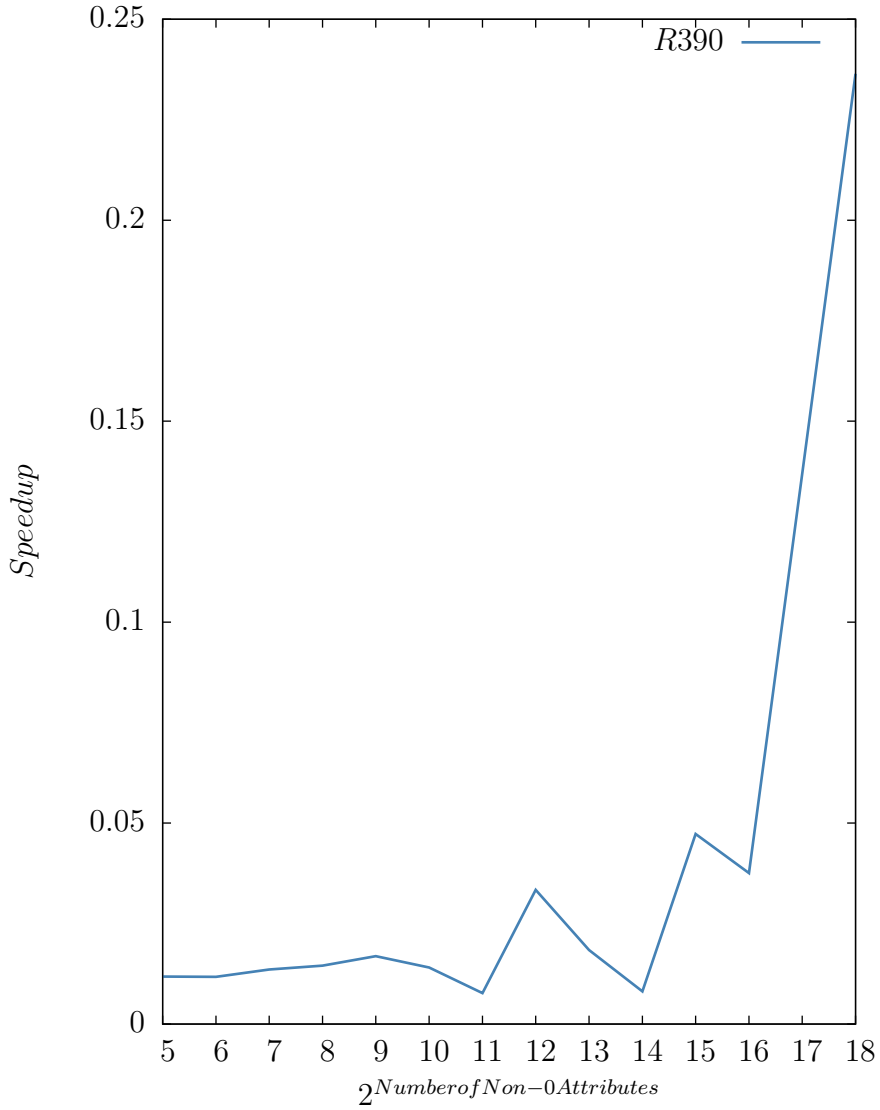


Figure 5.3: Stochastic Gradient Descent Training Latency (AMD R9 390 vs MOA implementation on AMD A8-7600)

# Chapter 6

# Conclusions and Future Work

*TODO*

# References


[1] aparapi - api for data parallel java. allows suitable code to be executed on gpu via opencl. https://code.google.com/p/aparapi/.

[2] Bolt c++ template library. http://developer.amd.com/tools-and-sdks/opencl-zone/bolt-c-template-library.

[3] C++ amp overview. https://msdn.microsoft.com/en-us/library/hh265136.aspx.

[4] clmath - amd. http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-math-libraries/.

[5] Compute shader overview. https://msdn.microsoft.com/en-us/library/windows/desktop/ff476331%28v=vs.85%29.aspx.

[6] cublas. https://developer.nvidia.com/cuBLAS.

[7] Hsafoundation. https://github.com/HSAFoundation.

[8] Hsafoundation/cloc. https://github.com/HSAFoundation/CLOC.

[9] Java se 7 java native interface-related apis and developer guides. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/.

[10] java.util.stream (java platform se 8). http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.

[11] Khronos opencl registry. https://www.khronos.org/registry/cl/.

[12] Nvidias next generation cuda(tm) compute architecture:fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[13] Openjdk: Project sumatra. http://openjdk.java.net/projects/sumatra/.

[14] Openmp.org. http://openmp.org/wp/.

[15] Ptx isa :: Cuda toolkit documentation. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[16] Renderscript—android developers. http://developer.android.com/guide/topics/renderscript/compute.html.

[17] Single instruction, multiple threads. http://en.wikipedia.org/wiki/Single_instruction,_multiple_threads#cite_note-spp-1.

[18] Spir - the first open standard intermediate language for parallel compute and graphics. https://www.khronos.org/spir.

[19] Hsa platform system architecture specification. http://www.hsafoundation.com/?ddownload=4944, 2014.

[20] Tolu Alabi, Jeffrey D. Blanchard, Bradley Gordon, and Russel Steinbach. Fast k-selection algorithms for graphics processing units. *J. Exp. Algorithmics*, 17:4.2:4.1–4.2:4.29, October 2012.

[21] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, January 2008.

[22] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van Der Vorst. Templates for the solution of linear systems: Building blocks for iterative methods, 1994.

[23] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, pages 97–104. ACM, 2006.

[24] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *J. Mach. Learn. Res.*, 11:1601–1604, August 2010.

[25] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Selected Papers from the Sixth International Conference on World Wide Web*, pages 1157–1166, Essex, UK, 1997. Elsevier Science Publishers Ltd.

[26] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Comput. Netw. ISDN Syst.*, 29(8-13):1157–1166, September 1997.

[27] L. Cayton. Accelerating nearest neighbor search on manycore systems. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 402–413, May 2012.

[28] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 380–388, New York, NY, USA, 2002. ACM.

[29] Wei-Sheng Chin, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel stochastic gradient method for matrix factorization in shared memory systems. *ACM Trans. Intell. Syst. Technol.*, 6(1):2:1–2:24, March 2015.

[30] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 253–262, New York, NY, USA, 2004. ACM.

[31] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 69–77, New York, NY, USA, 2011. ACM.

[32] 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.

[33] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

[34] 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*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[35] Dhruv Mahajan, S. Sathiya Keerthi, S. Sundararajan, and Léon Bottou. A parallel SGD method with strong convergence. *CoRR*, abs/1311.0636, 2013.

[36] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the arm and power relaxed memory models, 2012.

[37] Duane Merrill and Andrew S. Grimshaw. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(2):245–272, 2011.

[38] Jatin Chhugani Anthony D. Nguyen Victor W. Lee Daehyun Kim Pradeep Dubey Nadathur Satish, Changkyu Kim. Fast sort on cpus,gpus and intel mic architectures. `http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-radix-sort-mic-report.pdf`.

[39] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[40] Feng Niu, Benjamin Recht, Christopher R, and Stephen J. Wright. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *In NIPS*, 2011.

[41] C. Nugteren. Improving the programmability of gpu architectures, 2014.

[42] Jia Pan and Dinesh Manocha. Bi-level locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 378–389, Washington, DC, USA, 2012. IEEE Computer Society.

[43] P.C. Pratt-Szeliga, J.W. Fawcett, and R.D. Welch. Rootbeer: Seamlessly using gpus from java. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 375–380, June 2012.

[44] D. Sculley, Daniel Golovin, and Michael Young. Big learning with little ram. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.298.2337&rep=rep1&type=pdf`.

[45] Maxim Shevtsov, Alexei Soupikov, and Er Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes.

[46] 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.

[47] T. Aila S.Laine, T. Karras. Megakernels considered harmful: Wavefront path tracing on gpus. `https://research.nvidia.com/sites/default/files/publications/laine2013hpg_paper.pdf`.

[48] 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.

[49] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[50] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proc. VLDB Endow.*, 6(14):1930–1941, September 2013.

[51] Alan Tatourian. Nvidia gpu architecture and cuda programming environment. `http://tatourian.com/2013/09/03/nvidia-gpu-architecture-cuda-programming-environment/`.

[52] Philippe Tillet, Karl Rupp, Siegfried Selberherr, and Chin-Teng Lin. Towards performance-portable, scalable, and convenient linear algebra. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2013. USENIX.

[53] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[54] 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.