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

1.1 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 the C programming language - CUDA[14]. Presently there is a number of programming frameworks targetting specifically GPU architecture such as CUDA[60], OpenCL[13], RenderScript[18], DirectCompute[6] and more generic parallel-processing frameworks such as OpenMP[16] and AMP[4] 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 and most known programming frameworks, lists limitations of the traditional GP GPU programming. It also discusses the OpenCL 2.0 standard, which addresses some of the limitations and describes the Heterogenous System Architecture (HSA), an optimized platform architecture for

OpenCL 2.0.

1.2 GPU Architecture

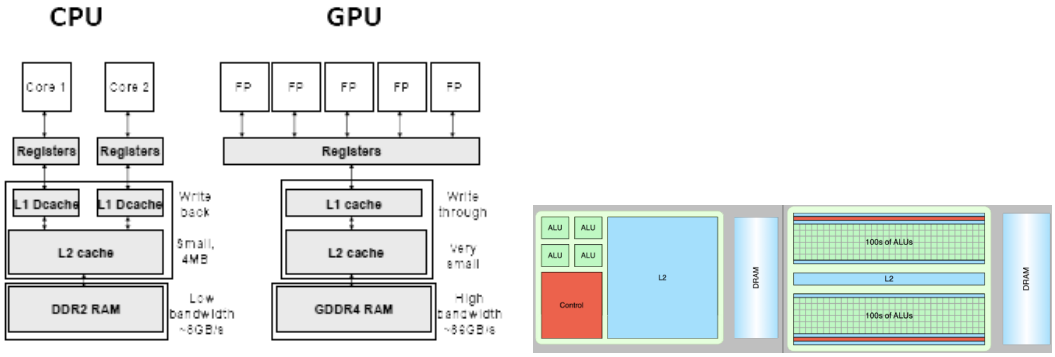


Figure 1.1: CPU versus GPU hardware architecture. Reproduced from NVIDIA GPU ARCHITECTURE and CUDA PROGRAMMING ENVIRONMENT by Alan Tatourian[79]

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 a maximum of 12 threads (24 with hyper-threading), where the minimal unit of execution for the NVIDIA GPU (called *wavefront*) is 32 threads. Modern GPUs implement a SIMT (Single Instruction - Multple Thread) execution model (AMD/NVIDIA desktop GPUs) first introduced by NVIDIA in the G80 model[14]. 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[19][60]. The CPU thread is a heavy-weight entity which is centered around execution of a specific task for an extended period of time. Whenever the CPU needs to preempt the running thread, its 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

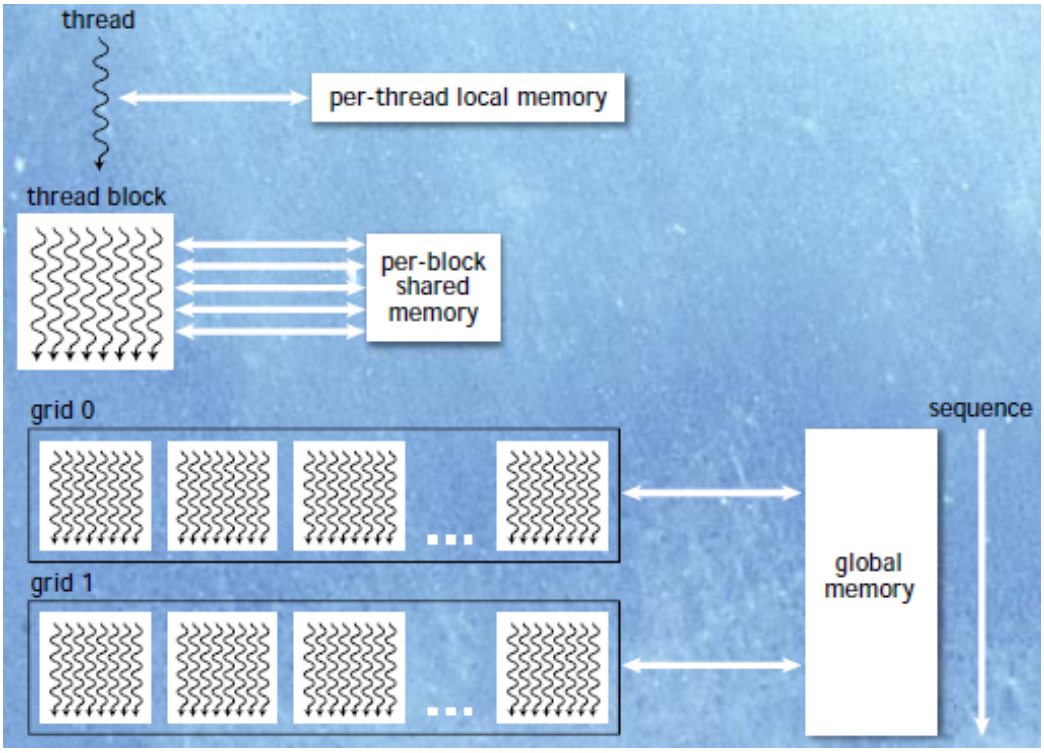


Figure 1.2: GPU Memory Tiers. Reproduced from NVIDIA GPU ARCHITECTURE and CUDA PROGRAMMING ENVIRONMENT by Alan Tatourian[79]

GPU programming uses the following abstractions:

- *Kernel* - a unit of execution
- *Thread* - a single unit of processed data
- *Threadblock* - a group of *threads* sharing the 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 the 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[75][60]. It should be noted that the wavefront size is a hardware specific feature and its optimization should be performed at the run-time.

1.3 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[15]. Traditional GPU languages such as CUDA[60] expose task scheduling and memory management giving the expert user fine-tuning capabilities. Low level languages provide an intermediate binary format compatible with multiple hardware targets. The tree of the GP-GPU technologies is presented in the Figure 1.

1.3.1 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[1][16]. Microsoft C++ AMP[4] is a C++ library which enables parallel computations for CPU and GPUs (using Microsoft DirectX Shading Language). The Rootbeer GPU compiler provides for transparent compilation of Java code into CUDA[67]. Aparapi provides a way to generate OpenCL kernel code from Java, theoret-

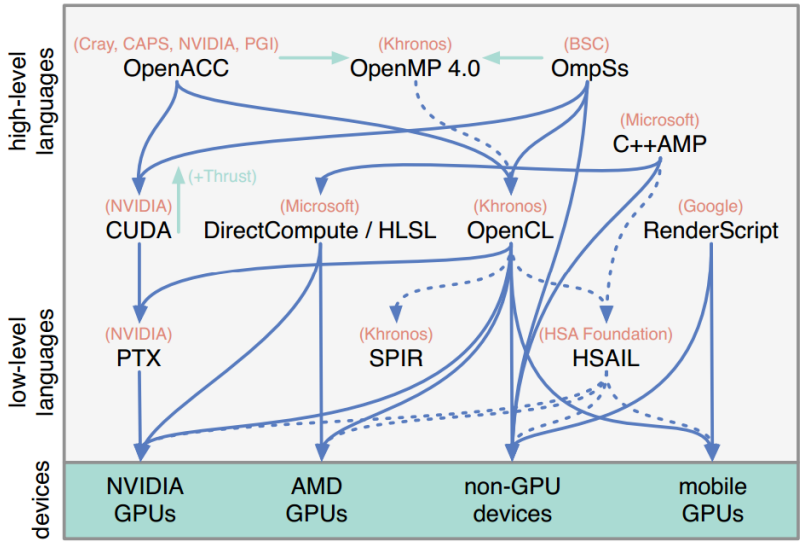


Figure 1.3: GP GPU technologies tree. Reproduced from C. Nugteren, Improving the Programmability of GPU Architecture, p. 21 [62]

ically allowing code which can be executed on CPU and offloaded to GPU if needed[2]. Project Sumatra is a OpenJDK project which focuses on the development of Hotspot virtual machine extensions capable of offloading JDK 8 Stream API[12] computations to the GPU[15].

1.3.2 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 the C language. Kernels are expressed as C-functions for one thread with parallelism defined at run-time by specifying dimensions of the execution grid and the thread blocks[60]

- OpenCL 1.X builds upon ideas implemented in CUDA by adding device management APIs and providing hardware-agnostic programming specification. OpenCL gives a *write once-run anywhere* guarantee but does not give any performance consistency guarantees across different hardware[77].

- RenderScript - an 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 *workgroup* size control in a bid to provide performance portability between different devices[18].

- DirectCompute/HLSL - Microsoft extension to Direct3D API for general purpose computing. It uses propietrary scripting language first introduced in DirectX 9 that has a limited support for double precision computing. DirectCompute only allows to specify the *workgroup* size at the compile time[6].

1.3.3 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 a *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)[17], Khronos Group specifies Standard Portable Intermediate Representation(SPIR)[20], and HSA Foundation specifies Heterogenous System Architecture Intermediate Language (HSAIL)[21].

1.3.4 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. Figure 1.4 shows execution time of a kernel which assigns index to each array element $X_i = i$ on AMD A8-7600 integrated GPU. The execution time starts to increase when input size is above 1024 and remains constant for lower values. To maximum performance on the integrated GPU of AMD A8-7600 will be achieved when input size will exceed 1024 elements.

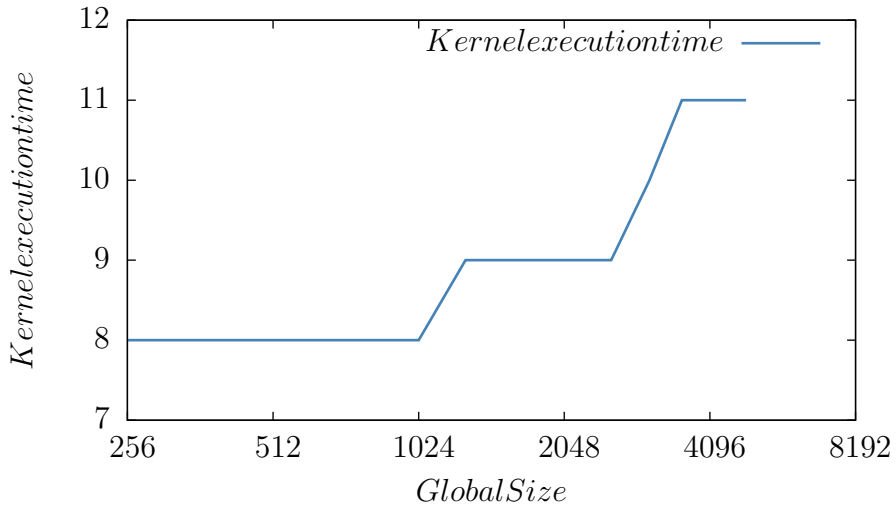


Figure 1.4: Kernel launch time on integrated Radeon R7 GPU(μsec).

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 the GPU[74]. Memory transfer is a bottleneck for Aparapi and its developers allow explicit memory management[2]. This effectively reduces a framework which promises general CPU-GPU interoperability to a mere Java wrapper of the OpenCL API.

Kernel Launch Constant time is needed to setup kernel launch which might offset any gain from parallelization if the data can be processed faster sequentially. Some algorithms have stop conditions that have to be checked to find out if the algorithm requires additional iterations. OpenCL 1.X specification does not allow scheduling of the kernel execution from within the kernel itself. In this case we need to synchronize with the host portion of the program to set up additional kernel launches introducing a bottleneck.

1.4 OpenCL 2.0

OpenCL 2.0 standard[13] introduces several features which attempt to address limitations of GPU programming:

- Shared Virtual Memory - both host and kernel code share the same address space thus either hiding memory transfers (discrete GPU driver stack) or if backed by the hardware architecture such as HSA eliminate the need for it[21].

- Dynamic Parallelism - OpenCL 2.0 allows scheduling of kernels from within a kernel without host interaction reducing the host CPU-GPU synchronization bottleneck.

- Pipes - the pipes feature allows for passing data from kernel to kernel without processing the whole input.

1.5 HSA Platform

AMD introduced the Heterogeneous System Architecture platform as an optimized platform architecture for OpenCL 2.0. Its specification introduces a set of requirements that allow both GPUs and CPU share the same memory space, synchronize execution using signals and atomics, and to schedule execution both from the GPU and the CPU[21]. 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. At the moment (Feb 2016) there is a OpenCL 2.0 - HSAIL compiler available[10] and a Linux-based runtime environment[9].

1.5.1 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 a packet is written to the queue, the ownership is taken by the HSA packet processor and it may change packet contents at any time[21]. 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 a 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.

1.5.2 HSA Signals

HSA uses *signals* to perform synchronization between the 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. The runtime environment provides a way to check the value of the signal or wait for the specific value.

1.5.3 HSA Memory Model

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 a relaxed memory model to allow a range of hardware optimizations to provide better performance by reodering load and store operations[56]. The HSA platform specification states[21]

> 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.

Similiar to Java Memory Model[55] it guarantees sequential consistency for correctly synchronized programs, that is 'synchronizing operations meet the requirements for sequential consistency within each scope/segment instance'[21]. This 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[21]. In addition to memory segments the 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, group segments are limited to wavefront and workgroup scopes [21]. Different workgroups accessing a global variable within the same workgroup scope will work with different instances of the variable. Write serialization only applies to the operations within the segment/scope that they specify.

1.5.4 Implementation Notes

At the time of writing (February 2016) the HSA Runtime implementation ignores sequential barrier flag thus iterative algorithms have to explicitly synchronize kernel execution using signals to avoid starting a new kernel before the previous finishes. There is a constant time needed to setup kernel launch, e.g. for AMD A8-7600, it is 6 μsec using HSA.

1.6 Conclusion

Modern specifications, such as OpenCL 2.0 and HSA, attempt to address some of the latency issues of GPU programming by the introduction of shared memory and lightweight dispatch/data passing mechanisms. This work will focus

on the evaluation of suitability of those technologies for the latency-sensitive processing of the data streams.

Chapter 2

System Architecture

We have implemented a data stream processing library which provides a set of classification algorithms for Massively Online Analysis(MOA)[31]. This library uses existing linear algebra package ViennaCL[80] which allows multiple backends such as CUDA, OpenCL, CPU. We have built some machine learning algorithms such as nearest neighbours search and stochastic gradient descent using its interfaces.

2.1 MOA interface

The ViennaCL library is implemented in C++ and as such requires Java Native Interface[11] to be used to interface from the Java Virtual Machine. Java Virtual Machine manages its own memory space and garbage collection may move data at any time. JNI provides two mechanisms to access array data from native code. First is copying - a java pointer is locked by a critical section and array content is copied to the native array. Second one 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 the *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.

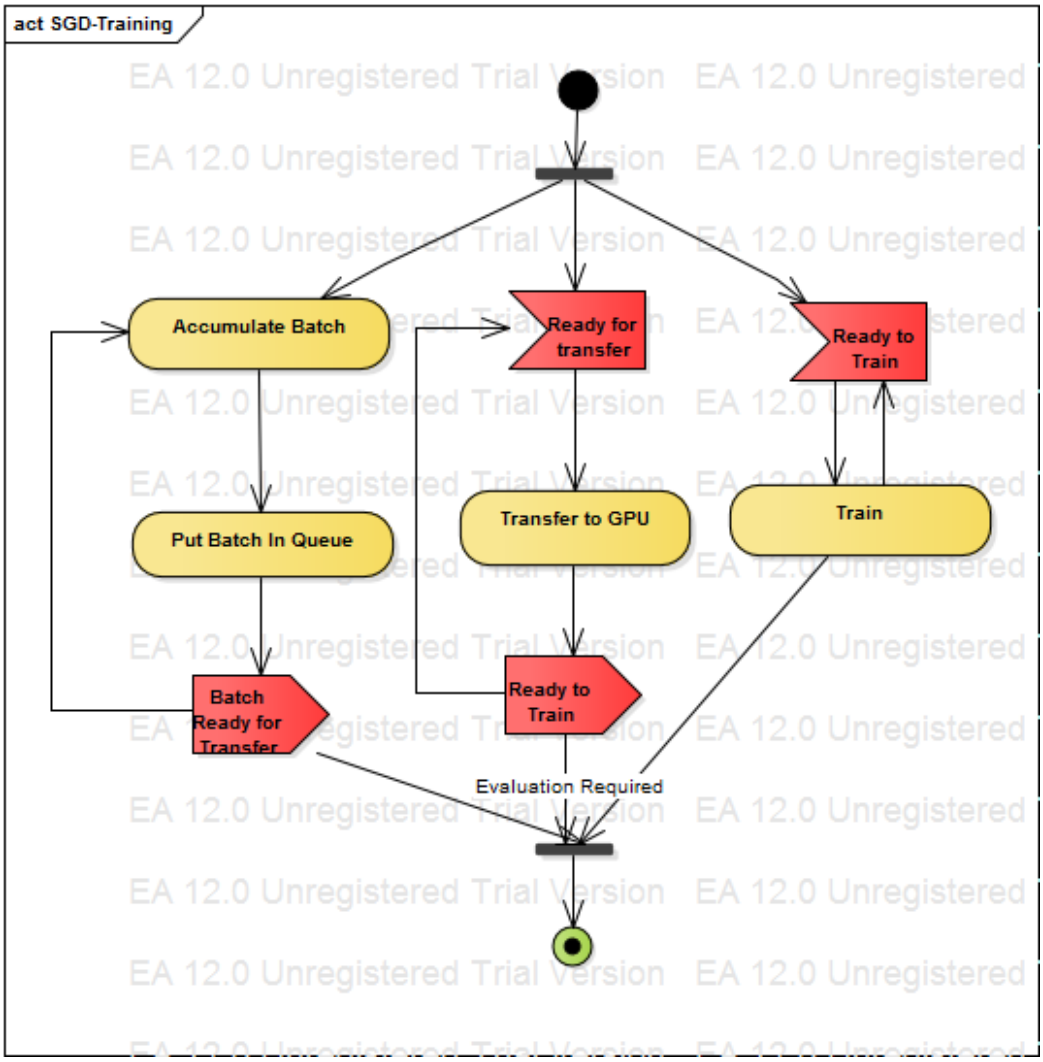


Figure 2.1: SGD classifier training activity.

2.2 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 larger problems. In such case partitioning will be used - the problem data is kept in the Java memory as a 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 better than double one (R390, R290). NVIDIA GPUs have 32x scale(Maxwell)[8]. There are several works that explore using fixed precision numbers to reduce memory requirements of machine learning tasks[69][48]. An alternative precision implementation will impose the overhead costs of double to fixed/floating point conversion, if a GPU classifier will be used, for instance, as a part of the meta-classifier ensemble.

2.3 HSA Backend

This work adds a new HSA backend to the ViennaCL library based on the HSA Runtime[21]. This implementation is tuned for Kaveri AMD APU and uses the same set of OpenCL kernels as the OpenCL backend. In the HSA backend the main system memory is transparently mapped to GPU memory and vice-versa, allowing to use vector or matrix element-addressing operations without first copying data to the CPU memory space.

2.4 OpenCL 2.0 features

At the time of writing (February 2016) the OpenCL 2.0 features such as work-group functions and device enqueue impose significant performance impact.

The library uses Shared Virtual Memory buffers to facilitate easy switching between OpenCL and HSA backends. The ViennaCL types are constructed from *cl_mem* representation of the shared virtual memory buffers.

2.5 Conclusion

The library provided as a part of this work is heterogenous. It uses a Java platform at the top level to interface with MOA and perform high-level computation. The linear algebra primitives and GPU interface are implemented in C++ and interface with the Java part via Java Native Interface. To reduce the overhead of Java to native code data transfers, this implementation uses $sun.misc.Unsafe$ that might be incompatible with the future Java releases. The library does not use OpenCL 2.0 features as tests show that they provide a negative performance impact at the time of writing. The training interface uses a latency-hiding trick to maximize GPU load for the model training. The test interface $getVotesForInstance$ performs synchronously and strives to provide the result with the lowest possible latency.

Chapter 3

k-Nearest Neighbours

3.1 Introduction

The k-Nearest Neighbours method[36] is a non-parametric method used for classification and regression. It computes a given instance distance to the examples with known labels 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 is bounded by twice the Bayes error if the number of examples approaches infinity[36]. Online implementation of the algorithm uses a sliding window of example instances updated by the data stream. Window size, instance dimensionality and allowed error bounds define the optimal approach to solving the k-Nearest Neighbours problem. *Exhaustive Search* has high computational complexity for the queries, but provides a constant update time. *Exact Clustering Methods* partition the search space to achieve logarithmic query complexity at the expense of the window update time. *Approximate Clustering Methods* reduce search space dimensionality to provide an approximate result with a given error bound. GP GPU computing may be used to accelerate their runtime though success for discrete GPUs depends on developer ability to eliminate branching, optimize memory access, and avoid excessive Host-GPU transfers. The latter poses a most significant problem for online

k-Nearest neighbours implementations. This chapter reviews the *Exhaustive Search*, some of the *Exact Clustering Methods* and *Approximate Clustering Methods* and provides notes on GP GPU implementation.

3.2 Exhaustive Search

The exhaustive approach to Nearest Neighbour search is to compute distance to each instance present in the sliding window. The computational complexity of the query $O(Nd)$ where N - number of instances and d - number of attributes. The GP GPU implementation of the exhaustive search consists of distance calculation and selection phase. The distances to the query can be computed using standard vector and matrix routines provided by GP GPU libraries implementing Basic Linar Algebra Subroutines(BLAS)[7][5][80]. The selection phase finds the nearest examples to the query out of all the computed distances. Sismanis et. al[74] provide time complexity of reduced sort algorithms, evaluate their performance on GPU and propose to interleave distance calculation and selection phases to hide latency as shown in Figure 3.1. This approach allows to obtain better performance on low window sizes[59].

The data for the distance calculation should be offloaded to GPU, while it performs the selection phase. The sliding window is partitioned, if needed, across multiple GPUs according to the GPU memory capabilities and does not use instances spatial information.

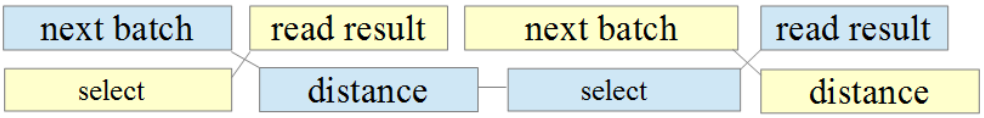


Figure 3.1: Interleaved distance calculation and sorting phases

3.2.1 Distance Calculation

The optimal implementation depends on the size of the window and number of attributes present[76]. For the small instance size (≤ 100) and windows of less than 10^4 elements one thread per instance implementation will provide the best solution. Best all around distance calculation should apply different strategies depending on the window size and number of attributes[76]. The alternatives are presented in the Figure 3.2.

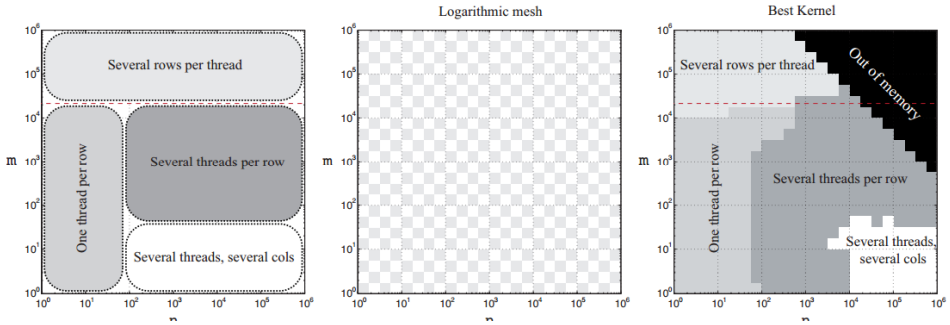


Figure 3.2: Left; Four matrix-vector multiplication kernels designed to perform well at different shapes m n of the matrix. Middle; Tuning mesh. Right; Best kernel in practice. The dashed line indicates the minimum 21504 rows needed in the matrix for full occupancy of the Nvidia Tesla C2050 card in a one-thread-per-row kernel. Note the logarithmic axes. Reproduced from High-Performance Matrix-Vector Multiplication on the GPU by Hans Henrik Brandenborg Sørensen[76]

The distance calculation primitive provided as part of this master thesis uses one thread per row approach.

3.2.2 Selection

Alabi, et.al[25] evaluated different selection strategies based on bucket sort algorithm and Merril-Grimshaw[57] implementation of radix sort.

Figure 3.3 shows performance of different selection strategies - merge sort from AMD Bolt library[3], bitonic sort similiar to reference AMD implementation and radix select based on Alabi, et. al. implementation[25]. The CPU sort and choose is a clear winner for small window sizes, e.g. 4096 for the test hardware. The larger windows should be processed by radix select. The

threshold will be different for each CPU/GPU combination and should be tuned by the runtime.

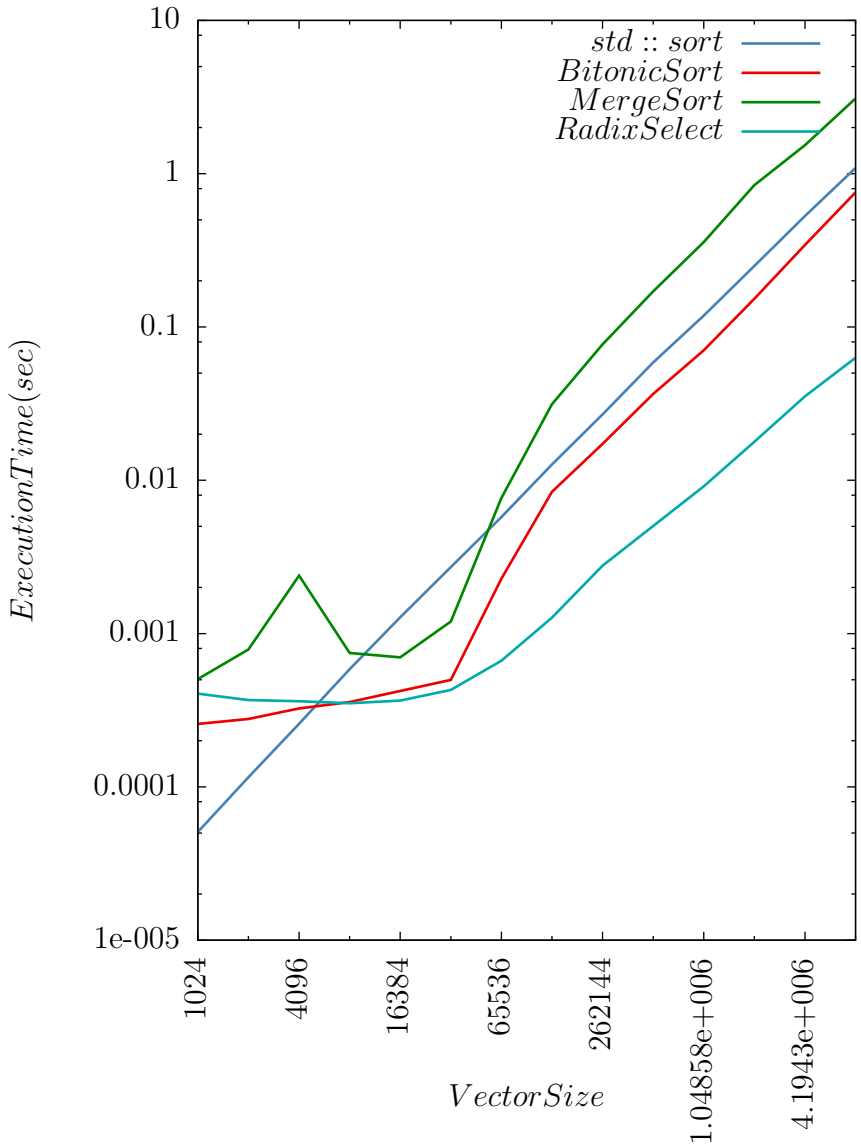


Figure 3.3: Selection Algorithm Performance for K=128. Test configuration GPU R9 390, CPU AMD A8-7600, AMD Catalyst version 15.20.

3.2.3 Conclusion

Exhaustive Search is a basic building block of the k-nearest neighbours algorithms. It is mandatory if we need to obtain an exact solution and is used to refine approximate methods results. The distance calculation parallelisation strategy should be adapted to the instance and windows size. Radix sort-based algorithm should be used for selection.

3.3 Exact Clustering Methods

Clustering techniques are widely used to limit the number of distance calculations needed for a nearest neighbour search. Space partitioning by vector dimensions is used by the k-d tree method[45], the random projection tree[44] provides a data structure splitting the search space along random vectors. Metric trees such as ball tree[63], cover tree[30], random ball cover[34] provide solutions for finding nearest neighbours in general metric space by organizing data points in groups around some centroids.

3.3.1 k-d Tree

The $k-d$ tree [45] is a balanced binary tree where each node represents a set of points $P \in \{p_1 \dots p_n\}$ and its children are disjoint and almost equal 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*[81] have shown that $k-d$ tree is outperformed by the exact calculation at moderate dimensionality ($n > 10$) and results in full processing of the data points if the number of dimensions is large enough. It follows that the $k-d$ tree requires $N \gg 2^(dimensions)$ points to examine less points than exhaustive search.

The listing of the $k-d$ tree construction and nearest neighbours search pseudocode is shown in the Figure 3.4. Parallel $k-d$ tree construction on GPU utilizes breadth-first approach[82][72] - the $k-d$ tree is constructed top-down with the split criteria computed in parallel for all nodes at the specific level. The priority queue based nearest neighbours search using k-d trees as shown in Figure 3.4 does not benefit much from the GP GPU parallelism due to the branch divergence and irregular memory access patterns[47]. The $k-d$ tree search approach presented by Gieske et. al[47] focuses on parallel execution of nearest neighbour queries in a lazy fashion. The query points are accumulated

```
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      nearest_neighbours.update(root);
30    else
31    {
32      split = root.location;
33      dim = root.dim;
34      if (p[dim] < split ) // search "closest" node
35        search(nearest_neighbours, root.left, p)
36      else
37        search(nearest_neighbours, root.right, p)
38
39      distance_to_split_plane = abs(split−p[dim]);
40      distance_to_point = abs(nearest_neighbours.furtherst_point()[dim]−p[dim])
41      if (distance_to_point >=distance_to_split_plane) // outer radius of NN heap
42      intersects the split plane
43      {
44        if (p[dim] < split )
45          search(nearest_neighbours, root.right p)
46        else
47          search(nearest_neighbours, root.left, p)
48      }
49    }
50  }
```

Figure 3.4: k-d tree construction and NN-search pseudocode

in the leaf nodes of the kd-tree until enough of them are present and then they are processed as a batch. This solves the issue of GPU underutilization and low performance if leaf nodes are processed sequentially for each example. An other approach would be to compute distances to the leaf nodes split planes[82] to provide a short-list of the leaf nodes for k-nearest neighbours search.

3.3.2 Random Projection Trees

The $k-d$ tree provides an effective partitioning mechanism for low data dimensionality but suffers in higher dimensions[81]. Many machine learning problems that are expressed in high dimensional space have lower intrinsic dimension as shown in Figure 3.5. Random projection tree exploits this fact

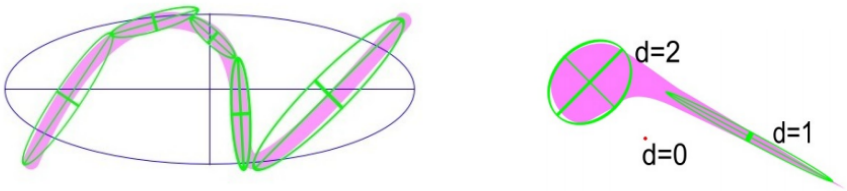


Figure 3.5: Distributions with low intrinsic dimension. The purple areas in these figures indicate regions in which the density of the data is significant, while the complementary white areas indicate areas where data density is very low. The left figure depicts data concentrated near a one-dimensional manifold. The ellipses represent mean+PCA approximations to subsets of the data. Our goal is to partition data into small diameter regions so that the data in each region is well-approximated by its mean+PCA. The right figure depicts a situation where the dimension of the data is variable. Some of the data lies close to a one-dimensional manifold, some of the data spans two dimensions, and some of the data (represented by the red dot) is concentrated around a single point (a zero-dimensional manifold). Reproduced from Learning the structure of manifolds using random projections by Freund Yoav et al.[44]

by splitting data along randomly chosen unit vectors as opposed to splitting along dimension axises in the k-d tree method as shown in Figure 3.6 [44]. The method performs a one dimensional random projection of the data points and splits them at the median of the projections.

The random projection tree split rules are presented in Figures 3.7 and 3.8. Figure 3.9 illustrates node selection for random projection tree median split

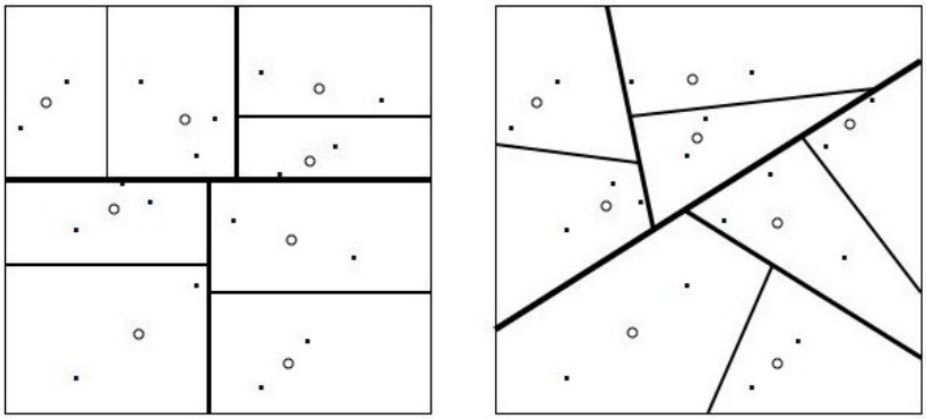


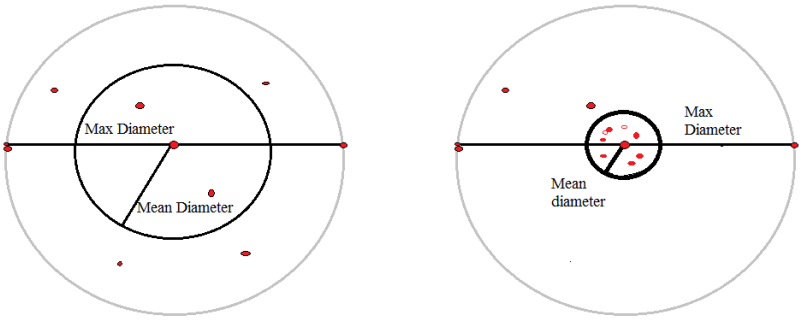
Figure 3.6: Left: Partitioning produced by k-d tree. Right: Partitioning produced by Random Projection Tree. Reproduced from Learning the structure of manifolds using random projections by Freund Yoav et al.[44]

```
1   tree_node random_tree_max(pointList, num_dimensions)
2   {
3     v = random_vector(num_dimensions);
4
5     x = pointList[ random() ];
6     y = max (distance( y in pointList, x) );
7     sigma = uniform_random(−1;1) ∗ 6 ∗ distance(x,y) / sqrt(num_dimensions);
8     split = median ( dot(v, x in pointList)+ sigma ) ;
9     left = {}
10    right = {}
11    for (x in pointList)
12    {
13      if (dot(v,x) <= split)
14        left += x;
15      else
16        right +=x;
17    }
18    node = {
19      .vector = v,
20      .split = split,
21      .left = create_tree(left, num_dimensions),
22      .right = create_tree(right, num_dimensions)
23    };
24    return node;
25  }
```

Figure 3.7: Random Projection Tree Pseudocode - Random Tree Max [37]

```
1
2     tree_node random_tree_mid(pointList, num_dimensions, c)
3     {
4       diameter = max( distance(x in pointList, y in pointList));
5       avg_diameter = mean(distance(x in pointList, y in pointList));
6       if ( diameter <= c* avg_diameter)
7       {
8         // small node, split using random projection threshold
9         v = random_vector(num_dimensions);
10        split = median( dot(x in pointList, v) );
11        left = {}
12        right = {}
13        for (x in pointList)
14        {
15          if (dot(v,x) <= split)
16            left += x;
17          else
18            right +=x;
19        }
20        node = {
21          .rule_type = dotproduct
22          .vector = v,
23          .split = split,
24          .left = create_tree(left, num_dimensions),
25          .right = create_tree(right, num_dimensions)
26        };
27        return node;
28      }
29      else
30      {
31        // large node. split by the distance from median
32        meanPoint = mean(x in pointList)
33        split = median( distance(x in pointList, meanPoint);
34        left = {}
35        right = {}
36        for (x in pointList)
37        {
38          if (distance(x, meanPoint) <= split)
39            left += x;
40          else
41            right +=x;
42        }
43        node = {
44          .rule_type = distance
45          .mean = meanPoint,
46          .split = split,
47          .left = create_tree(left, num_dimensions),
48          .right = create_tree(right, num_dimensions)
49        };
50        return node;
51
52      }
53    }
```

Figure 3.8: Random Projection Tree Pseudocode - Random Projection Tree Median Split[37]



(a) Small node, the maximum diameter is less than the mean diameter multiplied by the constant

(b) Large node, the maximum diameter exceeds mean diameter multiplied by the constant

Figure 3.9: Random projection tree median split node types

rule.

The projections along then chosen random vectors may be computed in a batch as a matrix by vector multiplication. The computation of projections of high dimensional data for the split criteria can be also viewed as a random projection operation described by Johnson and Lindenstrauss[50] and be implemented more efficiently than naive approach.

3.3.3 Random Projection

The seminal paper by Johnson and Lindenstrauss[50] established that for euclidian spaces any $x \in \mathbb{R}^n$ can be embedded into $\mathbb{R}^k$ with $k = O(logn/\epsilon^2)$ by projecting x in $\mathbb{R}^k$ using projection $k \times n$ matrix Φ without distorting inter-point distances by more than $(1 \pm \epsilon)$ and $k \geq O(logn)$. Johnson and Lindenstrauss[50] has shown that Johnson-Lindenstrauss condition holds for matrices with the following properties:

- Spherical symmetry - For any orthogonal matrix $A \in O(d)$, Φ multiplied by A and Φ have the same distribution.
- Orthogonality - rows are orthogonal to each other
- Normality - the rows are unit-length vectors

[23]

The lower bound of k was refined in several papers[43][38][49][22] with Dasgupta and Gupta[38] proving it to be $k \geq 4(\epsilon^2/2 - \epsilon^3/3)^{-1} ln(n)$ for $\epsilon \in (0, 1)$, where k - projection dimensionality, n - source dimensionality, ϵ - error. For high n this bound will still be too large to effectively employ low dimensionality search methods such as $k - d$ tree. An alternative would be to utilize very low dimensional space and then use disjunction to find the desired result. This approach is essentially an iterative random projection tree search where the dataset is split along leaf nodes.

The efficient implementation of the random projection-based algorithms requires a simple approach to construct Φ and a way to compute projection faster than naive multiplication of data point by $k \times n$ matrix. Achlioptas[22] achieved relatively sparse transformation matrix for random projection by proving that Johnson-Lindenstrauss condition holds if elements of the projection matrix are chosen independently according to the following distribution:

$$\begin{cases} +(n/3)^{-1/2}, P = 1/6 \\ 0, P = 2/3 \\ -(n/3)^{-1/2}, P = 1/6 \end{cases}$$

where n - source dimension and P - probability. This method provides a 3-fold speedup over the original one [50], since 2/3 of the transformation matrix elements are zero. Nir Ailon and Edo Liberty[24] have developed an almost optimal random projection transformation with runtime of $O(nlogn)$ as opposed to $O(kn)$ of the naive implementation. The main idea of the method is the application of the Heisenberg principle in its signal processing interpretation that both signal and its spectrum can not be both sharply localized. Thus applying Fourier transform to the sparse input vector will increase its support and will allow to make the transformation matrix even more sparse. To prevent the opposite, sparsification of the dense vector, the input data elements signs are randomly inverted with probability 1/2. The sparse transformation matrix used to complete the random projection[23] can be replaced by subsampled fourier

transform[24]. Nir Ailon and Edo Liberty define $k = O(\delta^{-4} log(N) log^4 n)$, where N - number of instances, n - source and k - projected dimensionality, that will preserve input vector norms by a given relative error δ[24]. The reference implementation used in this work is based on Gabriel Krummenacher implementation of subsampled randomized Fourier transform `https://github.com/gabobert/fast-jlt/tree/master/fjlt`. This algorithm is well suited for GPU implementation as it consists of FFT followed by element-wise operation. The last step (select random D elements) introduces irregular memory access that can not be worked around unless multiple transformations are performed in parallel. Figure 3.10 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(kn/3))* as it does not require to store transformation matrix and thus capable of projecting higher dimensional data on the same hardware.

3.3.4 Random Projection Tree Search

The random projection tree search can be performed in the same DFS manner as the k-d tree search. We may abort the search if the query point ends in the center leaf of the large node as shown in Figure 3.9 to obtain approximate solution.

3.4 Approximate Clustering Methods

The nearest neighbours search method in high dimensional space provides little benefit over exhaustive search where an exact distance is computed to each point in the database[81][28] unless the data has low intrinsic dimensionality. The approximate methods provide means to overcome this limitation by solving the problem of finding neighbours whose distance from the query point

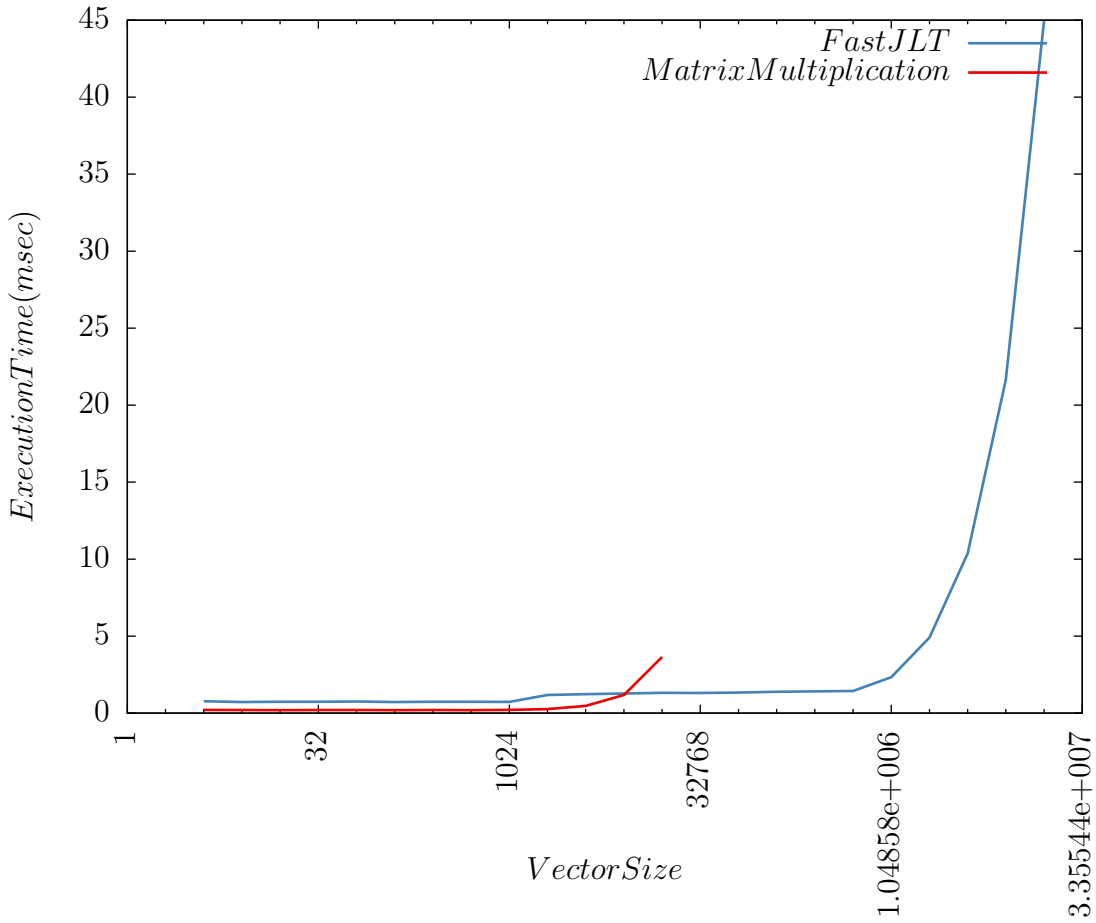


Figure 3.10: Fast Johnson-Lindenstrauss transform[24] vs. sparse matrix implementation[22] using ViennaCL. Test configuration GPU R9 390, CPU AMD A8-7600, AMD Catalyst version 15.20. The matrix multplication test was aborted due to the out of memory error.

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. Modern approximate methods use dimensionality reducing techinques such as random projection[66] and data set ordering using space filling curves[66][51][52] to improve query performance.

3.4.1 Locality Sensivity Hashing

Locality Sensivity Hashing[49] is a method that captalizes on the idea that there exists such hash functions $h(x), x \in \mathbb{R}^d$, that for points $p, q \in \mathbb{R}^d$, radius

R and an approximation constant c the following properties hold

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

where probabilty $P_1 > P_2$. The LSH algorithm uses a concatenation of $L \ll d$ such functions to increase the difference between P_1 and P_2[49]. Initially it was proposed to use Hamming distance as this function satisfies all required properties[49]. Later it was shown that other families of hash functions such as l_p distance[39], Jaccard coefficient [32][33] and angular distance(random projection)[35] are also locally sensitive. The algorithm selects L concatenations of the hash functions and uses them to transform input dataset points $v \in \mathbb{R}^d$ into lattice space $\mathbb{Z}^L$ storing them as L hash tables. The exact query is performed by concatenating contents of the L bins corresponding to the hash codes of the query, and then computing exact distances. The approximation is obtaining by stopping as soon as k points in cR distance from the query point are found. The method is GP GPU friendly as hash codes of the data points can be computed in massively parallel manner.

Alcantra A.F.[27] investigated several approaches to the hash table construction and retrieval on GPUs and has shown that the suitability of particular method is hardware dependent, iterative data structure content modification is difficult since insertion failure results in full hash table rebuild, which is offset by the fast rate of construction. For instance Alcantra et al[26] use tiered approach. A first-level hash function assigns an item to a bucket sized to fit into workgroup local memory and then performs rounds of cuckoo hashing[64] within the local memory until the hash table is built. Overall he notes that hash tables perform better than binary search despite uncoalesced memory access though radix sort/binary search remain a better option if queries are sorted and executed in bulk [27].

Space Filling Curves

Space filling curves[68] are curves that traverse all points of n-dimensional space in a given region providing a mapping from n-dimensional to 1-dimensional space. The GP GPU nearest neighbours algorithms[52][51][66] utilize z-order curve introduced by G. Morton in 1966 - an ordered list of numbers composed by interleaving bits of instance attributes[58]. This curve is often used due to the fact that the mapping can be constructed in $O(d)$ time and is easily implemented on GPU. A sample z-order curve is shown in Figure 3.11. A z-order

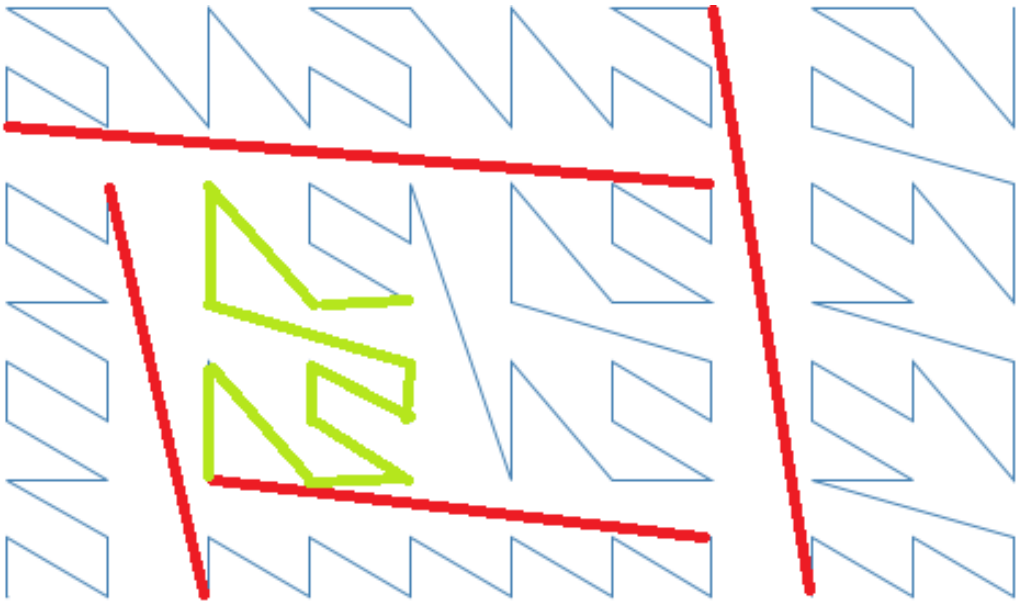


Figure 3.11: Z-Order Curve. Red lines highlight some region jumps. Green shows locality-preserving region.

curve mostly preserves data locality - points that are close in the n-dimensional space are also close together along the curve, but as shown in Figure 3.11 the z-order curve has jumps and to compensate for them existing GP GPU algorithms generate several curves from the input data shifted several times by some random vectors[52][51] to move the points into its locality-preserving regions. The shift search method by Li et al.[51] experimentally quantifies the shift value.

Sieranoja S.[73] generalized a z-order lookup table mapping algorithm for arbitrary number of dimensions. It can be translated into GPU implementa-

tion with minimal changes.

k-Nearest Neighbours using HSA architecture

The HSA architecture might allow to take OpenMP[16] approach for GP GPU acceleration of the nearest neighbours search by embedding parallel primitives such as FJLT and exhaustive search into existing algorithms without redesigning them to conform to the GPU architecture.

k-d tree queries The k-d tree queries require all or nothing approach - the algorithm design and data structures should be implemented in GP GPU specific manner to obtain higher throughput at the expense of latency[82][47]. In chapter 5 we will investigate whether it is possible to offload part of the computation to GPU without algorithm redesign to achieve performance improvement over serial version.

Random Projection Tree Construction and Queries The random projection tree tree can be built in the breadth first manner similar to existing GPU implementations of k-d tree algorithm. This approach requires pass over whole sliding window for each update. In chapter 5 we will investigate whether the online update of the random projection tree benefits from from offloading projection and thresholding to GPU.

Z-Order-based queries The significant problem in z-order query evaluation is a maintenance of the candidate lists for large k[51]. The cost of GPU-host transfer in this case is prohibitive and Li et al work around it by executing multiple queries at once so that it is more beneficial to perform sort as opposed to search[51]. In chapter 5 we will investigate whether it is efficient to perform single queries using z-order lists using approach by Li et al[51] for small k.

The approximate z-order based method can utilize random projections to shorten morton code length while preserving the relative distances to speed

up queries and sliding window updates.

Conclusion Current approach to use GP GPU for k-nearest neighbours problem is to design algorithms tailored for the target GPU platform. In chapter 5 we will investigate whether it is feasible to use GPU offload in a way similar to OpenMP[16] - by replacing parts of the serial algorithm with parallel primitives. HSA platform promises smaller overhead for heterogeneous computations and in chapter 5 we will investigate whether the improvement is sufficient to disregard latency hiding methods common to the current GP GPU implementations using z-order k-nearest neighbours as an example.

Chapter 4

Stochastic Gradient Descent

4.1 Introduction

Stochastic gradient descent is an iterative optimization method used in a wide variety of machine learning tasks. It minimizes the objective function $Q(w, x)$ dependent on the set of parameters w and set of examples x by updating parameters along the gradient of the randomly chosen example x_i : $w = w - \lambda \bigtriangledown Q(w, x_i)$, where λ is the iteration step size. also called the learning rate. In classical learning applications $Q(w, x)$ is a prediction loss function and the method aims to find the set of parameters w that provides a local minimum of an error on a training set. In online learning application $Q(w, x)$ is a regret function and the method aims to provide a best approximation of $y(x)$ where y is a classification associated with example x. Stochastic gradient descent may use an average gradient of several examples - a *mini-batch* to achieve lower variance. This chapter gives an overview of the approaches for stochastic gradient descent parallelisation, describes Hogwild! and 1bit SGD algorithms used in this work, and discusses a HSA algorithm implementation.

4.2 Approaches for SGD parallelisation

The stochastic gradient descent algorithm is inherently sequential. Ways to parallelise it across computing clusters are explored in a number of papers[46][61][54][83][70].

Langford et al[83] proposed a pipelined approach to SGD parallelisation. The input vector is divided into partitions that are processed in parallel by slave worker threads producing subgradients on each step. The subgradients are sent to the master worker that recomputes the resulting parameter vector and distributes it again to the slave threads. The authors observed that algorithm sensitivity to the delay in the weights update depends on the information gained from each example and that after a certain delay threshold the algorithm performance becomes much worse [83]. This highlights a problem for any parallel implementation of SGD - one has to minimize communication to achieve maximum computation speedup without introducing critical delay that will hurt convergence of the algorithm. The subsequent works [70][46][84][41][65] provided a way to balance the paralellism and delay either by *model paralellism* - partitioning data into independent sets and processing them in parallel [46][84], delaying update communication[70][41][65], or by removing a synchronization requirement for parameters update[61]. The state of the art cluster methods use a combination of all those techniques[41][65] to solve large scale optimization problems.

The single node stochastic gradient implementation often deals with *data parallelism* - for a given mini-batch its gradients are computed in parallel. This operation requires a parallel calculation of the objective function given model parameters and a set of example instances - essentially a vector (parameters) by matrix (examples) multiplication to obtain the current approximation. Davis and Chang[40] explore single-precision matrix-vector multiplication and conclude that modern CPUs gaining last-level cache sizes and external memory bandwidth, increasing data sizes of computational problems and constrained memory size of the consumer GPU cards create a barrier to adoption of GP-GPU, though they suggest that on-chip GPUs may be excellent building blocks for future heterogenous processors.

4.3 Hogwild!

The *Hogwild!* algorithm[61] uses an assumption that updates to the parameter vector w can be performed in a lock-free manner since each individual update only affects a small portion of it. The algorithm uses independent workers that have access to the shared parameter vector w. Each worker samples uniformly at random an example x and computes an update vector $\lambda \bigtriangledown Q(w, x_i)$. The worker then proceeds to apply updates to each element of the parameter vector w with non-zero gradient using atomic add function. The update may be implementented as either a *compare-and-swap* operation ensuring that no individual update is lost or as a replacement with the possibility to loose a certain portion of updates to the individual vector components. In both cases w is not locked and the implementation scales linearly with the number of available processors in replacement case and nearly linearly for *compare-and-swap* update as shown in tests using KDD Cup 2011[42] and Netflix prize[29] datasets [61].

4.3.1 Best Ball Optimization

The implementation of the Hogwild! algorithm (`http://i.stanford.edu/hazy/victor/Hogwild/`) contains a *best ball* autotuning method. The user picks a range of model parameters (e.g. learning rate) and the algorithm evaluates the corresponding models in parallel. After a pre-defined number of iterations the model with the lowest harmonic mean of the root mean square error is selected and its parameter vector is propagated to all other models.

4.3.2 Backoff scheme

The Hogwild! algorithm uses a diminishing learning rate. The algorithm uses a global synchronization point at the end of K iterations to reduce it by a constant β and continues running for the next $\beta^{-}1K$ iterations.

4.4 1bit SGD

The Hogwild! algorithm is inherently non-deterministic. The updates of the parameter vector are performed concurrently in a lock-free manner and may be lost should several updates contain a modification of the same parameter vector element w_i if update is performed with replacement. In both *compare-and-swap* and replacement cases the workers use parameter vector with non-deterministically partially applied updates to perform next the iteration. 1Bit SGD[70] approaches the communication bottleneck from the different angle. It works on the same assumption as Hogwild! - the updates from each mini-batch only affect a small portion of the parameter vector and adds a further constraint - only updates that are greater than a certain threshold should be communicated. In 1Bit SGD workers exchange gradient updates quantizied to one bit - that is all workers share a quantization constant τ and communicate gradient vector element that should be updated by this constant. The difference between computed value and quantization constant is stored locally by the worker and added to the next iteration gradient update [70].

This approach allows nearly linear speed-ups in a cluster setting[78] as opposed to previous results such as[71].

4.5 Stochastic Gradient Descent using OpenCL/HSA architecture

This work implements the Hogwild! algorithm for linear models on Heterogenous System Architecture and OpenCL platforms and compares its performance with the baseline MOA single-threaded implementation.

The implementation stores parameter vector in the memory-mapped file making it trivial to implement concurrency either as multiple threads or as multiple processes. The best ball optimization is not implemented though it would be trivial to launch several instances of the model and synchronize

parameters with the best model at the pre-defined intervals.

The 1bit SGD thresholding is used to minimize the number of updates to the shared parameter vector performed by the individual workers - only updates exceeding the quantization parameter τ are written to the memory-mapped file. The quantization also allows to work around the absence of the floating-point precision atomic *compare-and-swap* operation in OpenCL specification. The workers use atomic add and subtract operations instead to submit results to the shared memory. The designated worker locks the parameter vector at specified intervals to recalculate the quantization parameter τ so that the the quantization error is minimized and apply cumulative update.

The algorithm is parametrized by a number of minibatches it processes simultaneously - B. The residual quantization error is stored in matrix E with each row representing the residual error for the respective mini-batch. The parallel implementation processes as follows. Sparse matrix-vector multiplication is used to obtain a vector of dot-products. For each dot product a loss function value and corresponding update value are computed. The reduce operation is used to obtain a cumulative weight update value and to average it across all minibatches. Finally a weight update kernel is launched with each thread processing separate W_{ib} - individual weight from a specific mini-batch $b \in B$. The parameter vector is updated by $\lambda\tau$, where λ - learning rate, using the atomic add function if the update value exceeds threshold, or added to corresponding element of residual error matrix E. Based on E and th eiteration number an average quantization error is calcuated for each column and τ used for the next iteration is updated. The pseudocode for the algorithm is presented in Figure 4.1. The 1Bit SGD technique is used to work around the lack of floating point atomic functions as the algorithm only counts number of positive and negative τ updates.

```
1  // X − input data
2  // Es,El − residual error for "too small" and "too large" cases
3  // W − weights
4  // lambda − weight update function
5  // g(x) − gradient function
6  // tau − quantization parameter vector (tau[i] > 0)
7  // temp_weights − weight updates used during training step (integer)
8  tau training_step(X, Es, El, lambda, g, tau)
9  {
10   parallel_for (mini_batch in X)
11   {
12     W = read_weights();
13   mini_batch = g(mini_batch, W); // compute gradients for each example
14   vector minibatch_gradients = average_gradients(individual_gradients) +
15   Es[mini_batch] + El[mini_batch];
16
17   parallel_for ( mg[i] in minibatch_gradients)
18   {
19     if (abs(mg[i]) < tau[i])
20     {
21           // update "too small" error
22      Es[mini_batch] = mg[i];
23     }
24     else
25     {
26     lambda(i, sign(mg));
27     // update "too large" error
28     El[mini_batch] += mg[i] < 0 ? mg[i] + tau[i] : mg[i] − tau[i];
29     }
30   }
31   }
32   commit_weights();
33   return update_tau(tau,Es, El);
34 }
35
36 // atomic update of temporary weights
37 lambda(i, sign)
38 {
39   atomic_add(temp_weights[i],sign);
40 }
41
42 read_weights()
43 {
44   return W + learning_rate ∗ temp_weights ∗ tau;
45 }
46
47 commit_weights
48 {
49   W = W + learning_rate ∗ temp_weights∗tau;
50 }
```

Figure 4.1: Hogwild!-based stochastic gradient descent

4.6 Conclusion

The proposed stochastic gradient descent algorithm is massively parallel with a single global synchronization point that occurs once in several mini-batches to recompute τ value. It can be hidden by host-GPU transfer as shown in chapter 2. Chapter 5 will compare runtimes of the algorithm on integrated and discrete GPU using HSA and OpenCL platforms respectively.

Chapter 5

Experimental Results

5.1 Experiment Setup

The experiments were performed using AMD A8-7600 Radeon R7 10 Compute Cores 4C+6G CPU and integrated GPU, 16 Gb DDR3 1600 RAM and AMD Radeon R9 390 as a discrete GPU. Catalyst 15.8 drivers were used in OpenCL tests unless otherwise specified and AMDKFD driver version 1.6 was used in HSA tests unless otherwise specified. The experiments were run using Java Virtual Machine 1.8u45 for Window and Linux platforms. The operation systems used were Window 8.1 and Ubuntu Linux 14.04.

5.2 Experiment Data

The experiments used Infinimnist MNIST dataset generator[53], synthetic data streams produced by MOA[31] and Twitter data stream from January 2016. Infinimnist was used to generate a dataset containing digits from 10000 to 99999 with 784 numeric attributes and an average 167 non null values per instance. MOA data streams were used to illustrate the edge cases and validate the correctness of the algorithms implementation The twitter data was represented as a two class classification problem - two popular hashtags were chosen as classes and tweets containing them were represented as a bag of words. The data generated has 39794 instances with 5591 numeric attributes.

Each instance has an average of 3 non-null attributes per instance.

5.3 Evaluation

The evaluation of k-Nearest Neighbours algorithms was performed using MOA *EvaluatePeriodicHeldOutTask* with $k = 5$ unless otherwise specified.

5.4 k-Nearest Neighbours

5.4.1 Exhaustive Search

The nearest neighbours queries rely on the exhaustive search primitive to find the exact solution. The tests below used the distance measure implemented as 1 instance per thread GPU kernels for small dimensionalities and 1 workgroup per instance for larger ones. The latencies of both approaches were measured the Spectre GPU and single float precision were measured as shown in Figure 5.1 and the kernel switch threshold was set to 256. The spikes on the graph correspond to the large power of two strides between work-items for the first kernel and work-groups for the second one that result in global memory read requests serialized over same memory channel. This can be solved by introduction of the extra column to the attribute vector to avoid large power of two strides.

Figures 5.4.2, 5.3 show latency of the algorithm and speedup compared to the reference MOA implementation.

The tests did not show any difference in the algorithm accuracy due to the single float calculation as the minimum non-negative distance between data points exceeded the rounding error.

The twitter data is extremely sparse and though the dense representation has unreasonable memory requirements it still shows reasonable performance as shown in Figure 5.4 and can be used to process small windows.

The HSA implementation shows better performance on small window sizes

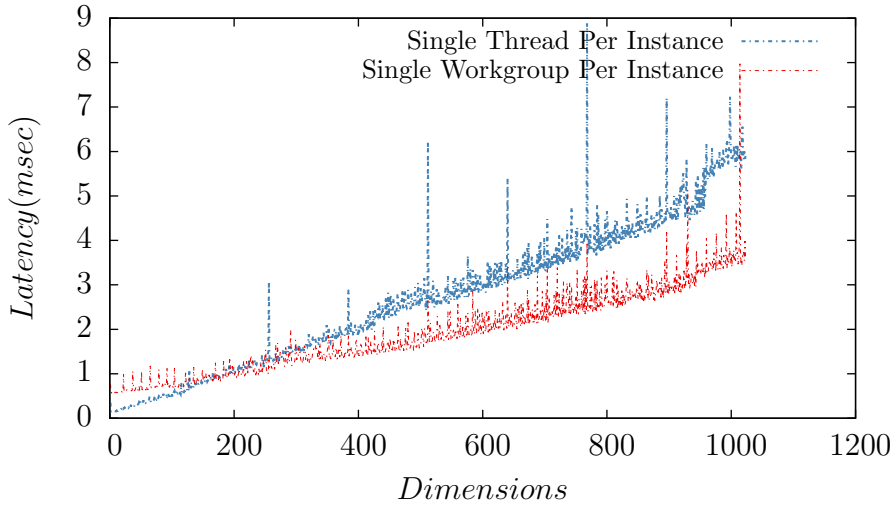


Figure 5.1: Latency of the distance calculation for an arbitrary number of dimensions

($\leq$ 1024 for MNIST) where discrete GPU is affected by the fixed data transfer cost. It should follow same pattern as the OpenCL implementation after breakeven point, but in fact the latency scales linearly with the number of scheduled work items.

The average Hawaii GPU utilization during single process test was 40% and additional speedup may be obtained by load-balancing queries between several processes.

5.4.2 k-d Tree

The refrence MOA imlementation tree leaf size was increased to 256 to limit the number of tree splits over sliding window and additionally *updateRanges* method of *NormalizableDistance* class was changed to iterate only over attributes present in the sparse instance. This has resulted in much faster (1500x) data evaluation without affecting training performance. Figure 5.5 shows training and evaluation latency on Inifinimnist dataset and Figures 5.6, 5.7 show respective speedups.

The training performance speedup was obtained due to the parallel calculation of ranges for the distance function. The bounds were calculated in a parallel fashion, one workgroup per attribute peaking out at 700x speedup for

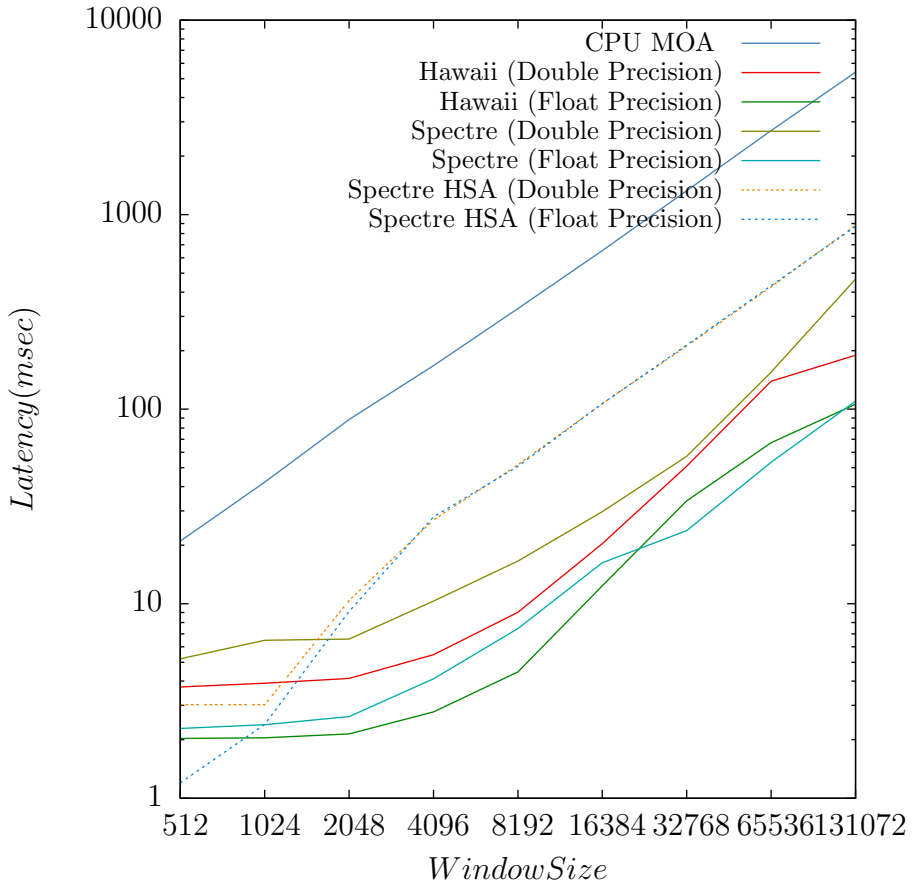


Figure 5.2: Latency of k-NN exhaustive search on Infinimnist stream

the optimal window size on Spectre integrated GPU.

The HSA backend severely underperforms compared to OpenCL. The latter uses buffers with CL_MEM_ALLOC_HOST_PTR and achieves zero-copy for the coarse memory buffers. The HSA performance is affected by the lack of optimization present in the OpenCL driver the cost of scheduling additional workgroups is linear according to Figure . *TODO - prove with empty kernel?.*

The Hawaii GPU utilization during single process evaluation test was averaging 27% and thus it is possible to improve evaluation performance by load-sharing queries between multiple processes without affecting query latency. *TODO - GPU utilization graph*

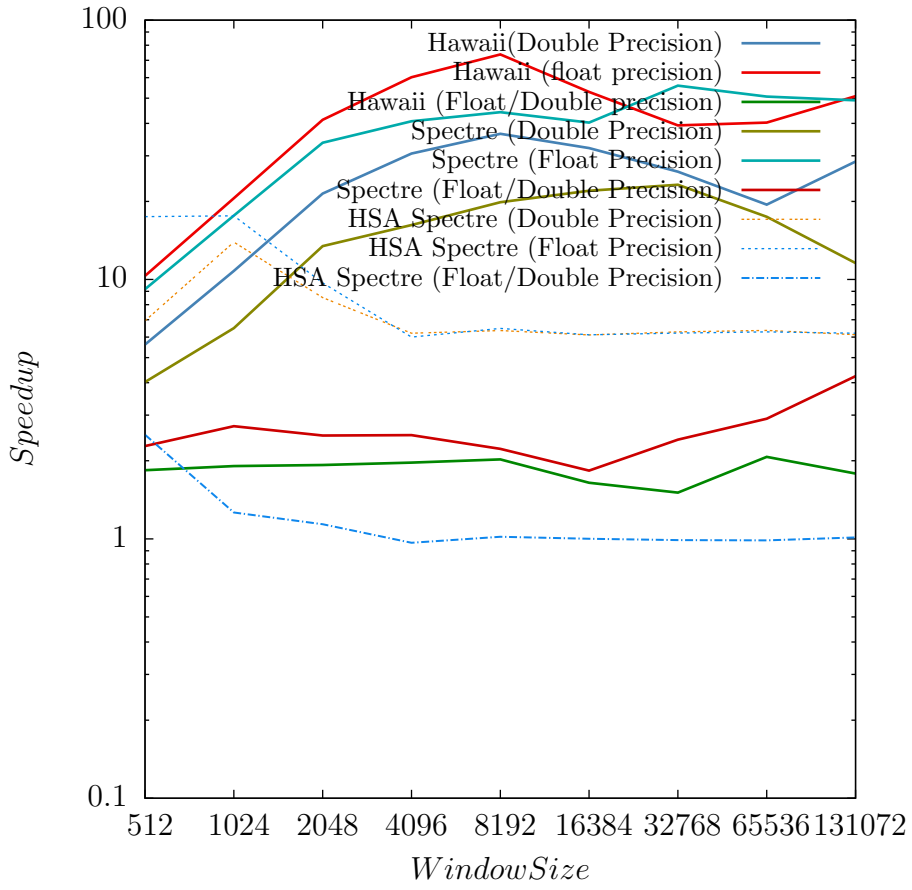


Figure 5.3: LinearNN Speedups on Infinimnist Stream

5.4.3 Random Projection Tree

The random projection tree performance on Infinimnist is shown in Figures 5.8,5.11, 5.10, 5.11. TODO: Accuracy compared to linear search/k-d tree?

The random projection tree training has lower speedup that k-d tree due to the lack of expensive batch operations such as calculation of min-max for the attribute values in the tree split. The projections are performed immediately as instances are added to the tree and instances are redistributed in the leaf nodes before evaluation using cached values. Thus the performance increase obtained from the parallel computations is hidden by the kernel launch overhead even in the zero-copy scenario. *Should I add a few words that method is overall better for speed-wise than k-d tree and training wise it is the same speed?* The matrix multiplication method of random projection outperformed FJLT in line with the benchmark in Figure 3.10 for the selected number of attributes (784).

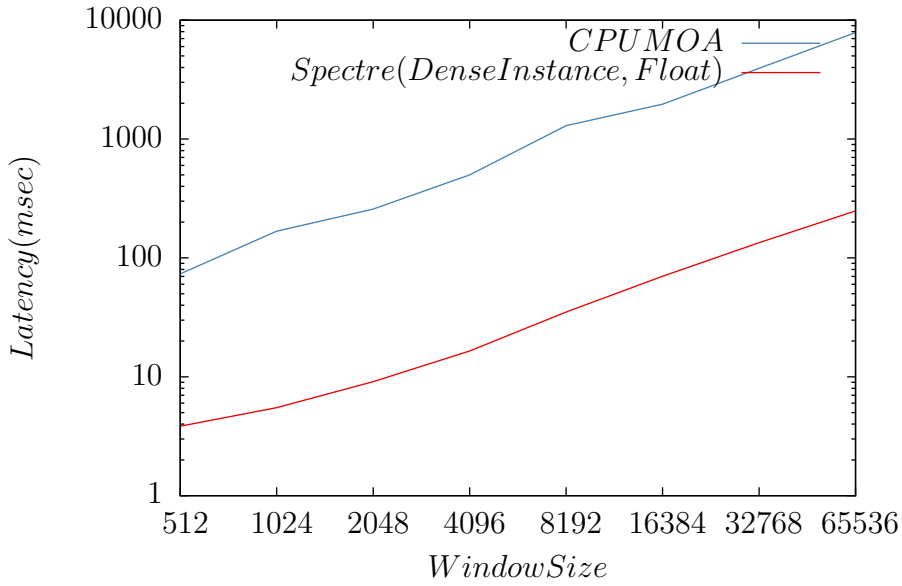


Figure 5.4: LinearNN Latency of CPU MOA and Spectre OpenCL implementations on Twitter Stream

The Hawaii GPU utilization during single process evaluation test was averaging 27% and thus it is possible to improve evaluation performance by load-sharing queries between multiple processes without affecting query latency. *TODO - GPU utilization graph*

5.4.4 Z-Order Search

TODO

5.5 Stochastic Gradient Descent

This work implements Hogwild![61] based variation of stochastic gradient descent using multinomial hinge gradient function and two versions of the updater:

- 1-Bit SGD-based - the updates to weight values are performed atomically by a pre-computed quantization constant. The individual weight updates are not lost.
- Direct - the updates are performed with replacement - a value is read

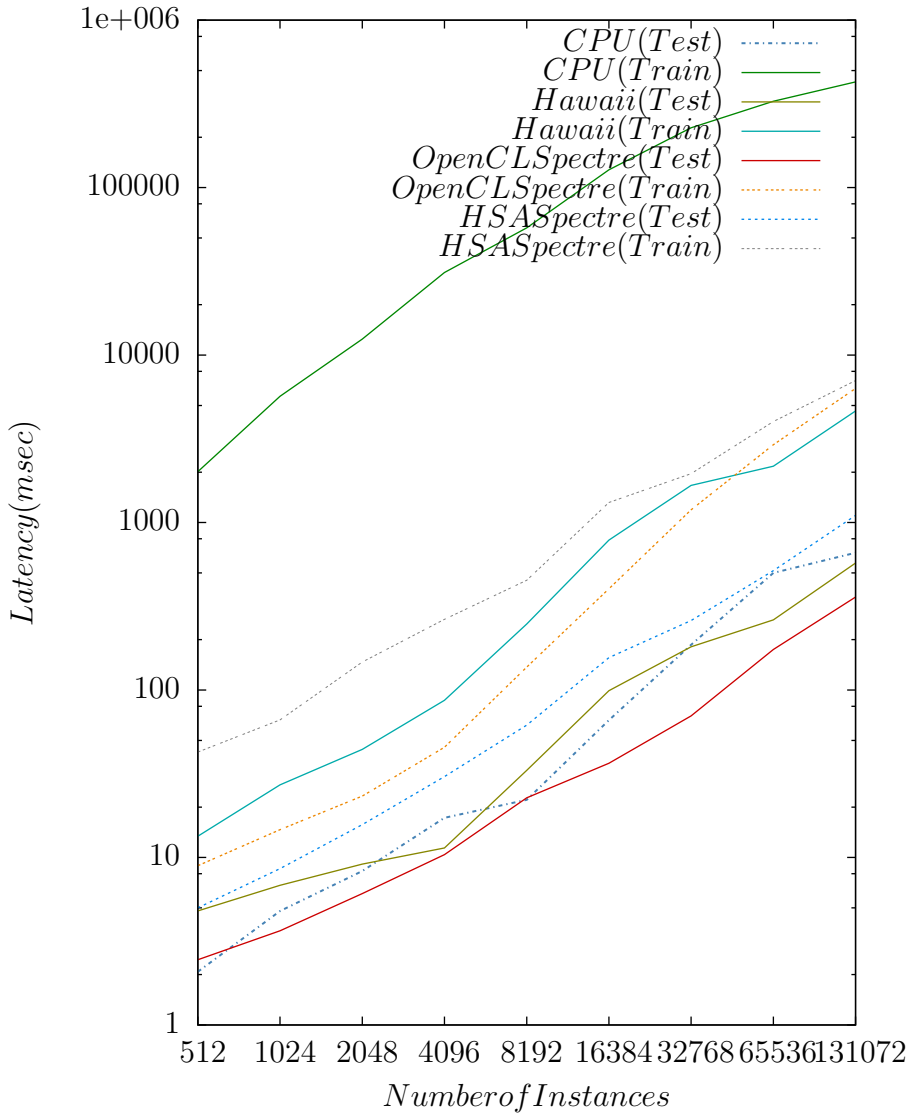


Figure 5.5: k-d Tree training and evaluation latency on Infinimnist

out, updated and atomically stored. It is possible to loose individual weight updates if two workers update the same weight.

The 1-Bit updater requires a global synchronization step to update quantization constant that can be performed with a certain delay. Figure 5.13 shows effect of the delay in a test with 1 update worker, minibatch size 1 and a data stream generated by MOA *RandomRBFGenerator* with default parameters - 10 numeric attributes and 2 classes. The Direct method runs without any synchronization steps but is prone to the update loss depending on the number of workers and sparsedness of the data stream. The Figure 5.12 shows that even for dense data the error is not significantly affected if the workers are not

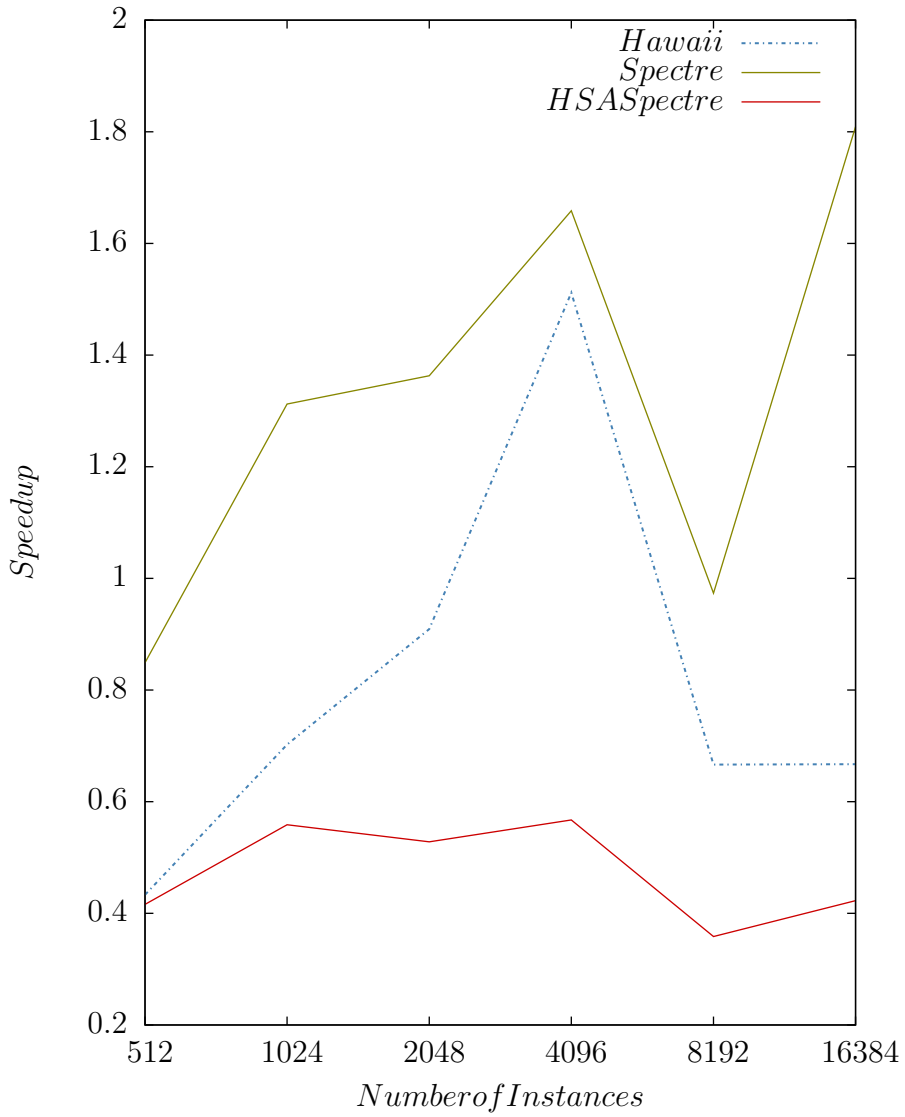


Figure 5.6: k-d Tree evaluation speedup on Infinimnist

synchronized.

The minibatch size used in further tests was set to 64 as per Figure 5.14. The performance of direct updater and 1bit updater with 1 synchronization per 1000 iterations was identical. Figure 5.15 shows performance depending on number of the worker processes. The single threaded implementation outperforms GPU due to the reads and writes to the shared memory. The HSA implementation is capable to accessing the shared memory segment directly from the kernel and thus achieves positive speedup.

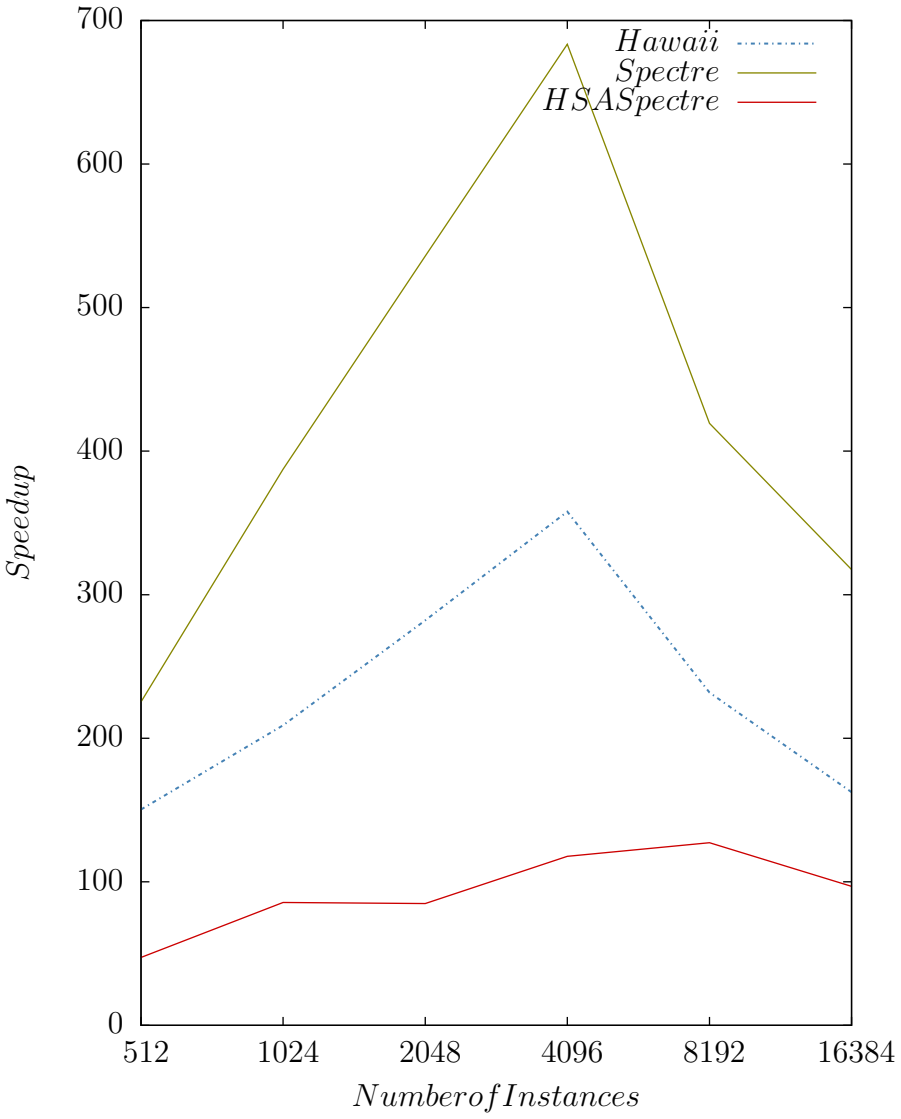


Figure 5.7: k-d Tree training speedup on Infinimnist

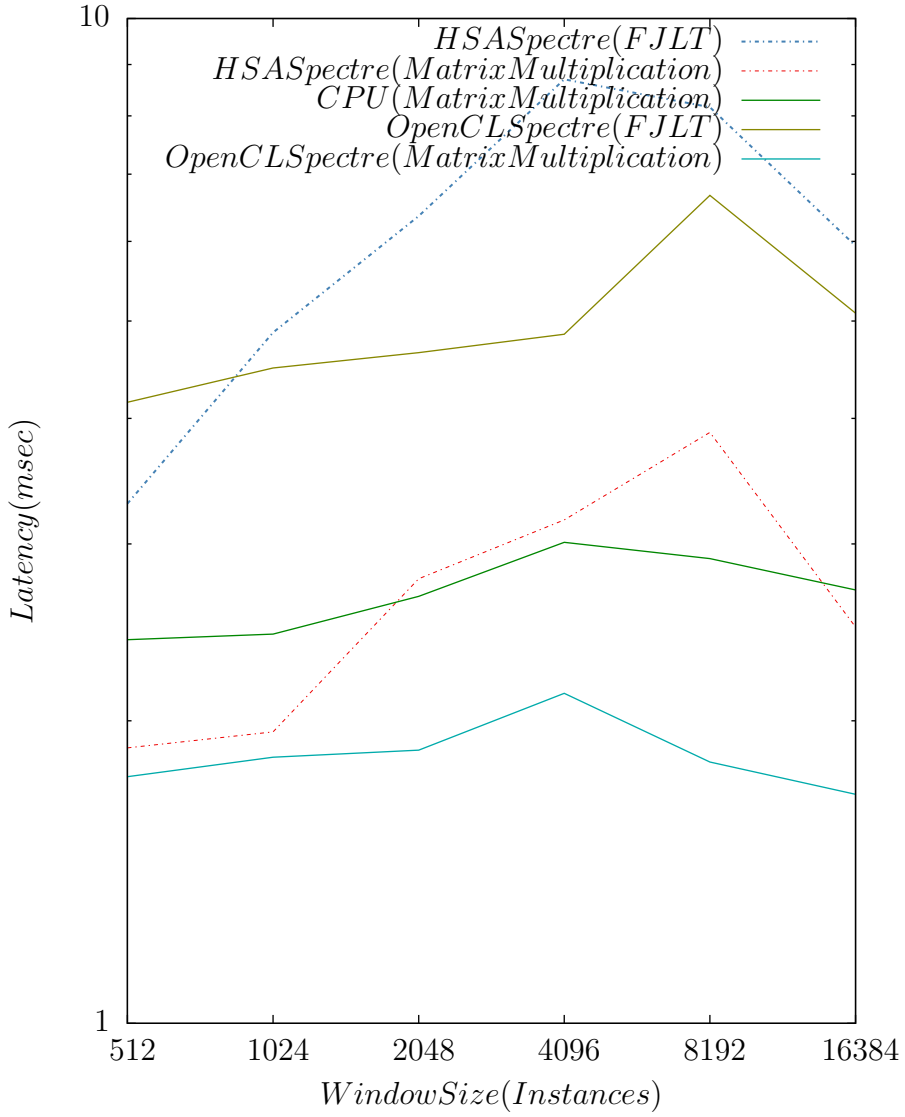


Figure 5.8: Random Projection Tree evaluation latency on Infinimnist dataset

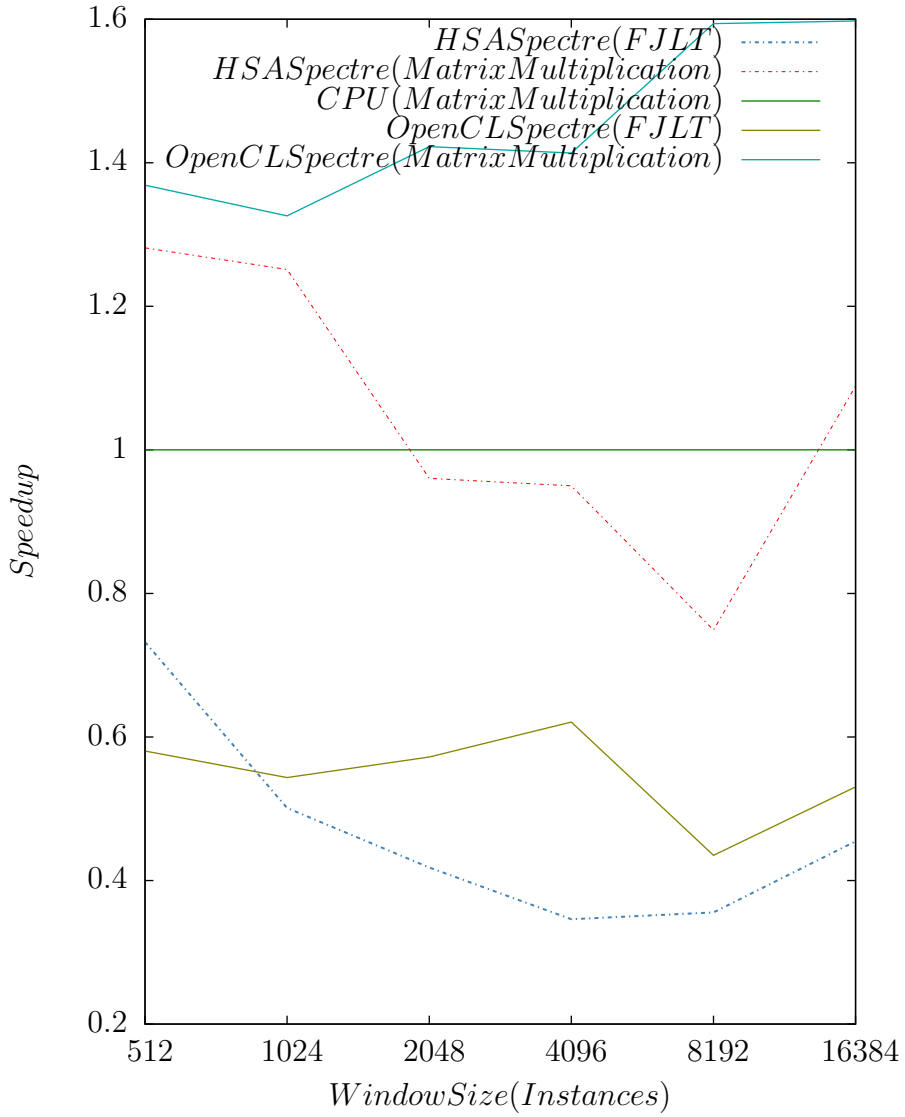


Figure 5.9: Random Projection Tree evaluation speedup on Infinimnist dataset

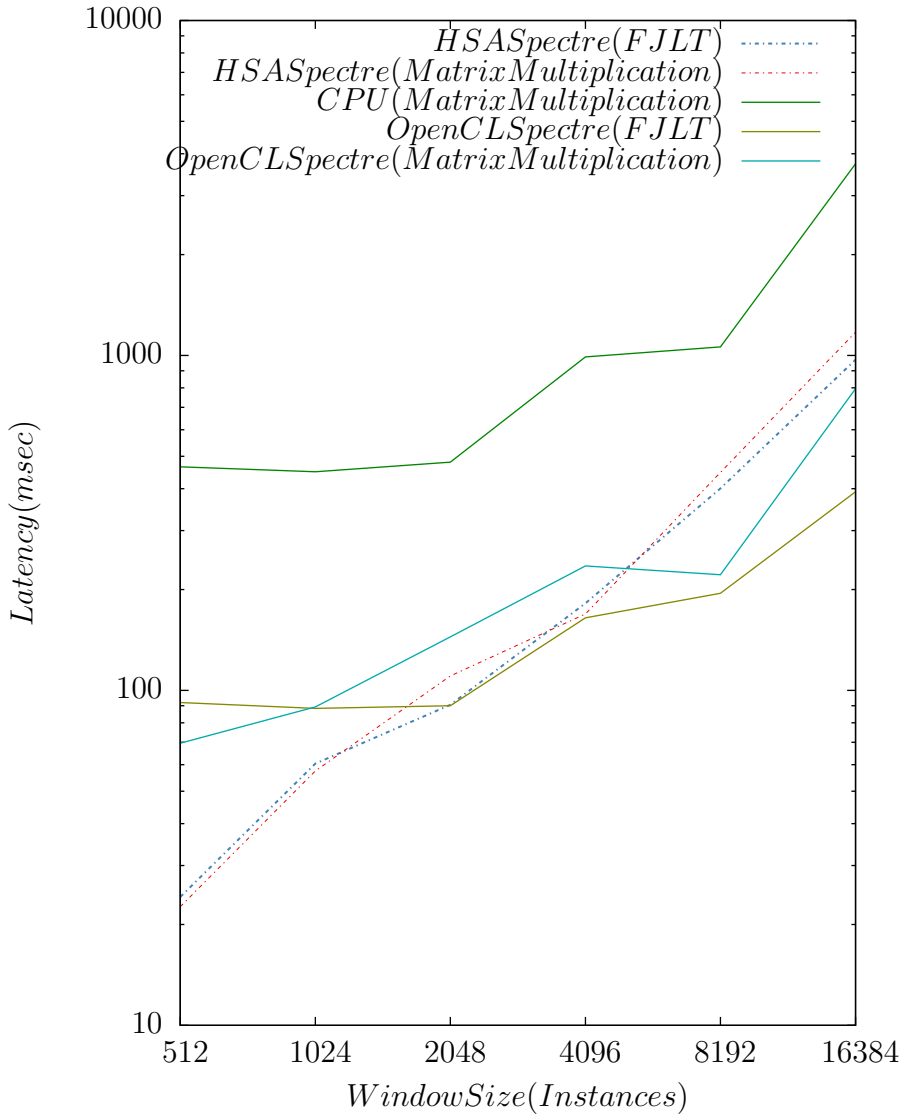


Figure 5.10: Random Projection Tree training latency on Infinimnist dataset

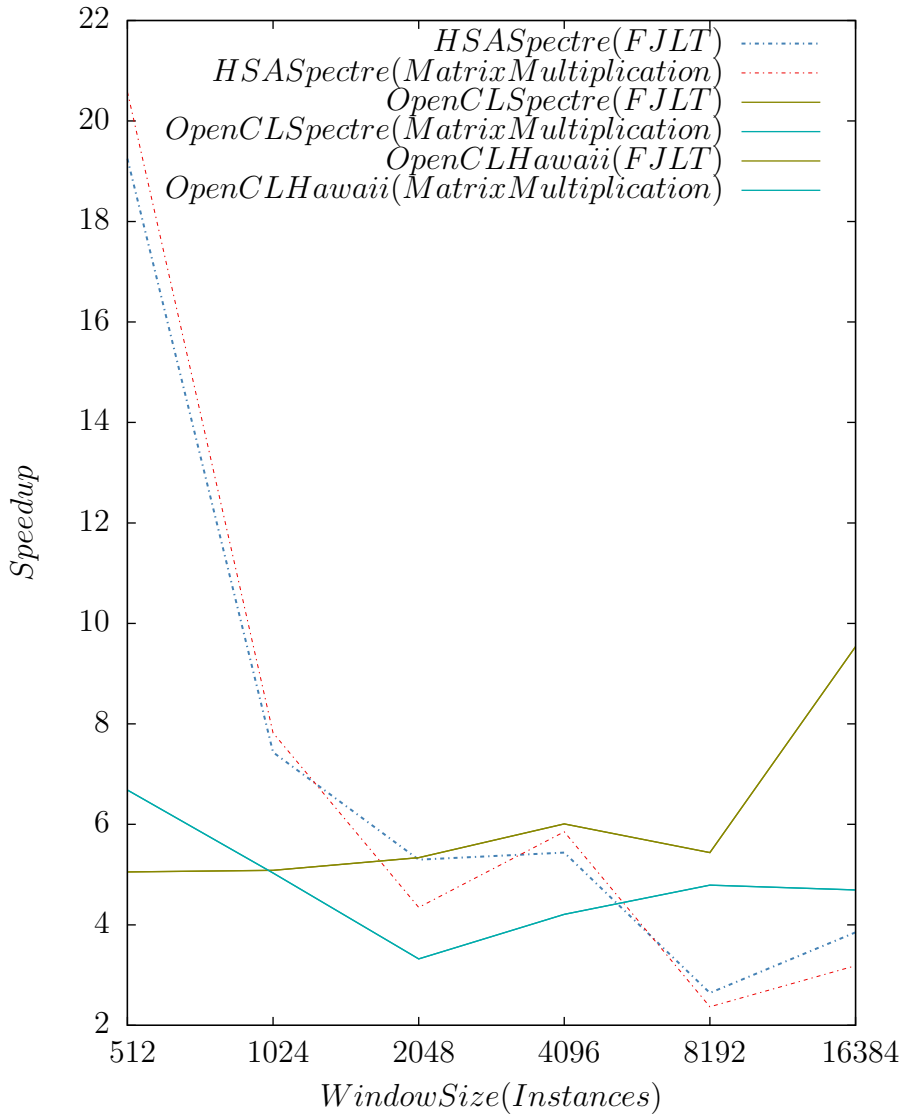


Figure 5.11: Random Projection Tree training speedup on Infinimnist dataset

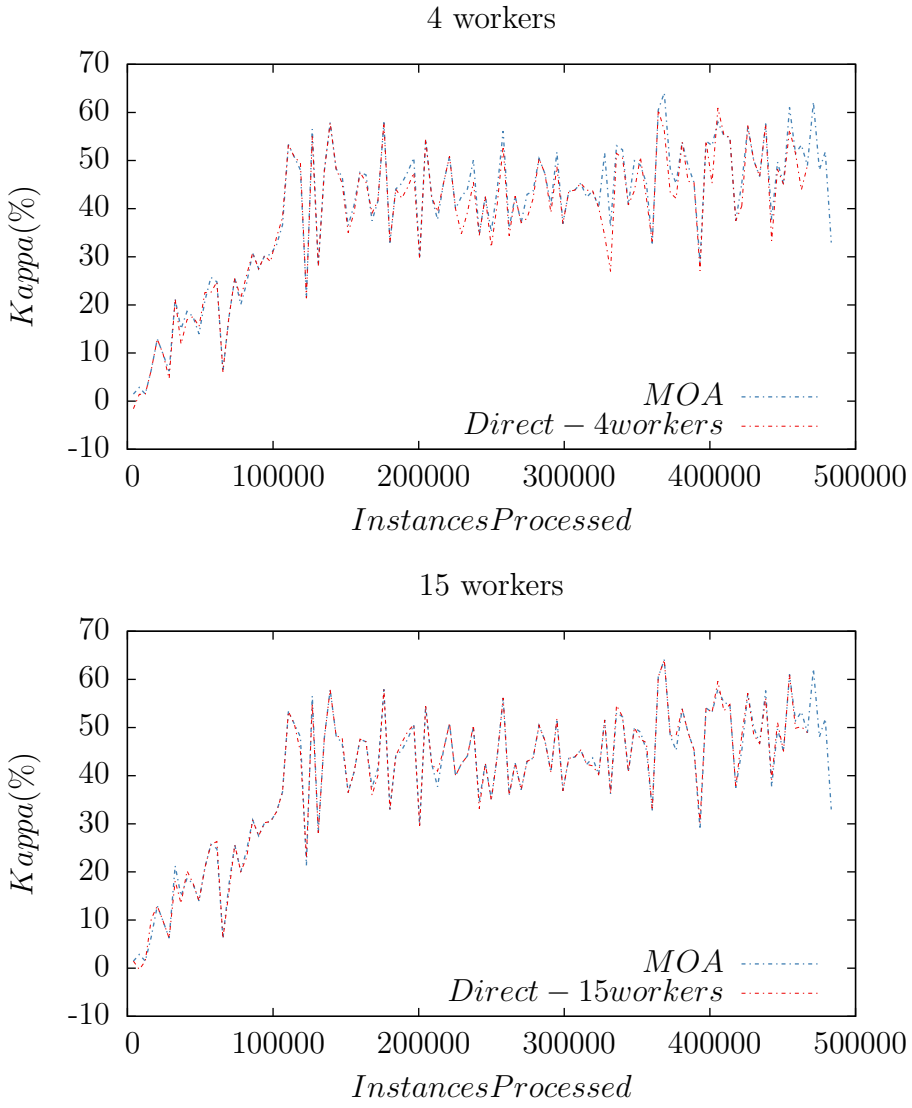


Figure 5.12: Direct update number of workers/Kappa statistic

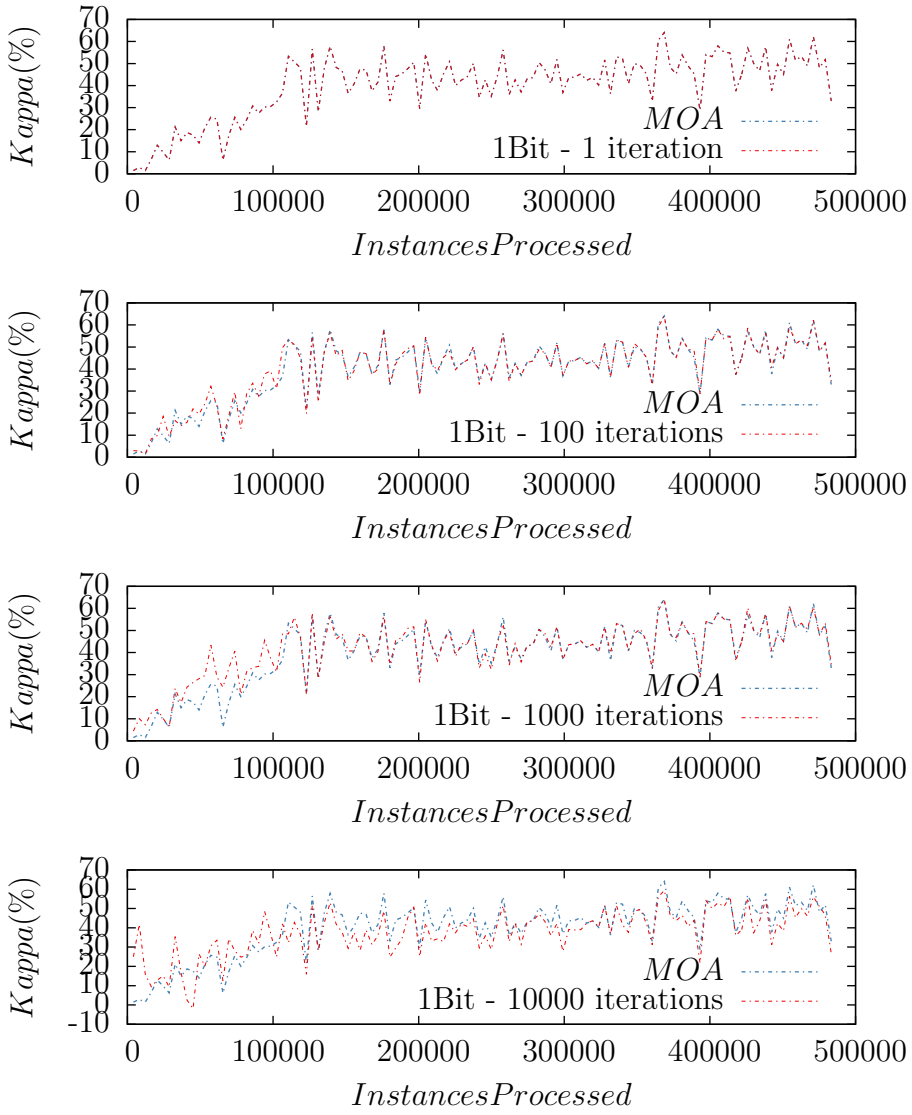


Figure 5.13: 1-Bit SGD quantization delay effect on Kappa statistic

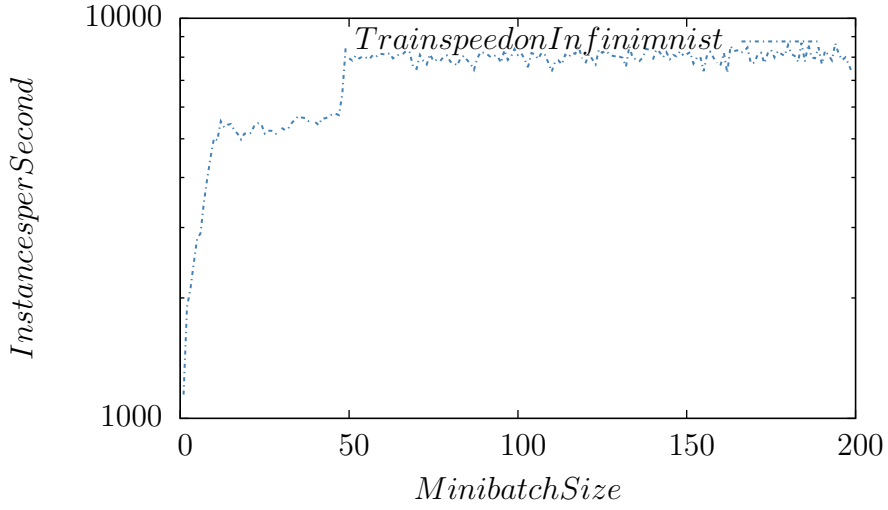


Figure 5.14: Training Speed depending on batch size

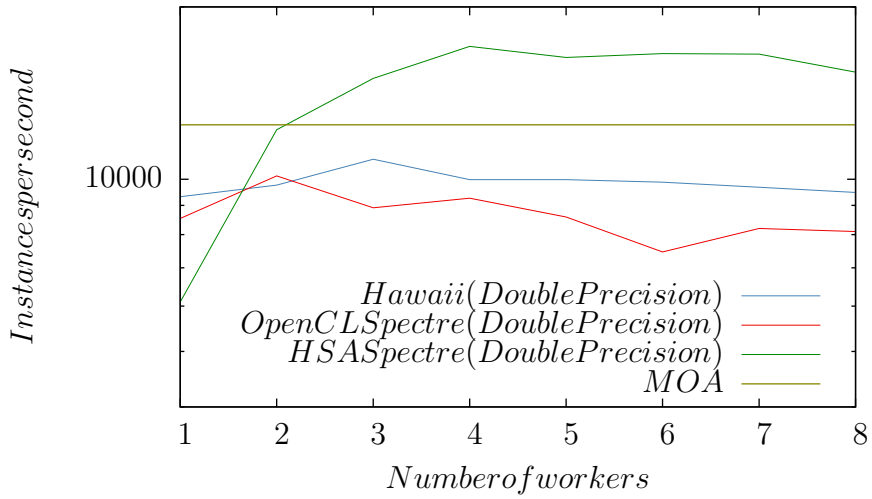


Figure 5.15: Training Speed depending on number of workers

Chapter 6

Conclusions and Future Work

6.1 HSA

The HSA platform provides means for the efficient offload of parallel computation to the GPU device where the tight coupling with the CPU is required. The traditional discrete GPUs outperform it in orders of magnitude when processing the batch tasks such as SGD classifier training but will perform on-par or slower when the algorithm requires frequent heterogeneous synchronization.

References

[1] About openacc—www.openacc.org. http://www.openacc.org/About_OpenACC.

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

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

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

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

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

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

[8] Explaining f64 performance of gpus—arrayfire. http://arrayfire.com/explaining-fp64-performance-on-gpus/.

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

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

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

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

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

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

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

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

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

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

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

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

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

[22] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687, June 2003.

[23] Nir Ailon and Bernard Chazelle. The fast johnson-lindenstrauss transform and approximate nearest neighbors. *SIAM Journal on Computing*, 39(1):302–322, 2009.

[24] Nir Ailon and Edo Liberty. An almost optimal unrestricted fast johnson-lindenstrauss transform. *ACM Transactions on Algorithms (TALG)*, 9(3):21, 2013.

[25] 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.

[26] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the gpu. In *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, pages 154:1–154:9, New York, NY, USA, 2009. ACM.

[27] Dan Anthony Feliciano Alcantara. *Efficient hash tables on the gpu*. University of California at Davis, 2011.

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

[29] James Bennett and Stan Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[30] 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.

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

[32] 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.

[33] 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.

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

[35] 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.

[36] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theor.*, 13(1):21–27, September 2006.

[37] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 537–546, New York, NY, USA, 2008. ACM.

[38] Sanjoy Dasgupta and Anupam Gupta. An elementary proof of a theorem of johnson and lindenstrauss. *Random Struct. Algorithms*, 22(1):60–65, January 2003.

[39] 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.

[40] John D Davis and Eric S Chung. Spmv: A memory-bound application on the gpu stuck between a rock and a hard place. *Microsoft Research Silicon Valley, Technical Report14 September*, 2012, 2012.

[41] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[42] Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. The yahoo! music dataset and kdd-cup'11.

[43] Peter Frankl and Hiroshi Maehara. The johnson-lindenstrauss lemma and the sphericity of some graphs. *Journal of Combinatorial Theory, Series B*, 44(3):355–362, 1988.

[44] Yoav Freund, Sanjoy Dasgupta, Mayank Kabra, and Nakul Verma. Learning the structure of manifolds using random projections. In *Advances in Neural Information Processing Systems*, volume 20, 2007.

[45] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, September 1977.

[46] 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.

[47] 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.

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

[49] 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.

[50] William Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Conference in modern analysis and probability (New Haven, Conn., 1982)*, volume 26 of *Contemporary Mathematics*, pages 189–206. American Mathematical Society, 1984.

[51] Shengren Li, Lance Simons, Jagadeesh Bhaskar Pakaravoor, Fatemeh Abbasinejad, John D Owens, and Nina Amenta. kann on the gpu with shifted sorting. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 39–47. Eurographics Association, 2012.

[52] Michael D Lieberman, Jagan Sankaranarayanan, and Hanan Samet. A fast similarity join algorithm using graphics processing units. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 1111–1120. IEEE, 2008.

[53] Gaëlle Loosli, Stéphane Canu, and Léon Bottou. Training invariant support vector machines using selective sampling. In Léon Bottou, Olivier Chapelle, Dennis DeCoste, and Jason Weston, editors, *Large Scale Kernel Machines*, pages 301–320. MIT Press, Cambridge, MA., 2007.

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

[55] Jeremy Manson, William Pugh, and Sarita V Adve. *The Java memory model*, volume 40. ACM, 2005.

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

[57] 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.

[58] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

[59] 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.

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

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

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

[63] Stephen Malvern Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[64] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *JOURNAL OF ALGORITHMS*, page 2004, 2001.

[65] Thomas Paine, Hailin Jin, Jianchao Yang, Zhe Lin, and Thomas Huang. Gpu asynchronous stochastic gradient descent to speed up neural network training. *arXiv preprint arXiv:1312.6186*, 2013.

[66] 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.

[67] 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.

[68] Hans Sagan. Space-filling curves, 1994.

[69] 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.

[70] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*, September 2014.

[71] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. On parallelizability of stochastic gradient descent for speech dnns. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 235–239. IEEE, 2014.

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

[73] Sami Sieranoja. High dimensional knn-graph construction using space filling curves. `http://urn.fi/urn:nbn:fi:uef-20150325`, 2015.

[74] 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.

[75] 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`.

[76] 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.

[77] 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.

[78] Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

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

[80] 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.

[81] 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.

[82] 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.

[83] Martin Zinkevich, John Langford, and Alex J. Smola. Slow learners are fast. In Y. Bengio, D. Schuurmans, J.D. Lafferty, C.K.I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 2331–2339. Curran Associates, Inc., 2009.

[84] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.