Heterogeneous Computing for Data Stream Mining

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

Graphical Processing Units are the de-facto standard for acceleration of data parallel tasks in high-performance computing. They are widely used to accelerate batch machine learning algorithms. High-end discrete GPUs are characterized by a very high number of cores (thousands), high bandwidth memory optimized for the stream access and high power requirements. Integrated GPUs are characterized by a medium number of cores (hundreds), medium bandwidth memory shared with CPU optimized for the random access and low power requirements. Data stream processing applications are often required to provide response within a limited time frame, operate on data in relatively small increments and have strict power requirements if deployed on the embedded devices. This work evaluates performance of integrated and discrete GPUs belonging to the same chip family on several variants of k-nearest neighbours algorithm over sliding window and stochastic gradient descent using OpenCL and novel Heterogeneous System Architecture platforms. We conclude that integrated GPUs provide a niche solution catering to small work sizes that offers better power efficiency and simplicity of deployment.

Acknowledgements

I would like to express my gratitude to my supervisor Professor Bernhard Pfahringer for support and guidance, to Dr. Karl Rupp for making the excellent ViennaCL library available, and my family for allowing me to make this happen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real world applications such as industrial monitoring, sensor networks or financial data, generate large unbounded streams of data which have to be processed within the pre-defined response time. For instance, acquirers connected to international payment systems such as Mastercard are required to produce payment authorization message within the payment system's specified timeframe, or face financial penalties for each violation. The industrial or security monitoring system has to detect abnormal conditions and take the appropriate action. The data volume and computational complexity required for the data stream processing is often well above the capabilities of a single server and the problem is tackled by deploying stream processing systems based on either proprietary or open source real-time frameworks such as Apache Storm[62] or Apache Spark[79]. Graphical Processing Units (GPUs) are an industry standard accelerator for a high performance computing used for a wide variety of scientific and industrial tasks and existing machine learning frameworks such as Microsoft CNTK[22] incorporate GPU offload to speed up computations. Deployment of GPUs in a cluster environment presents certain challenges due to the high power consumption and fast interconnect requirements. The developer should also be aware of the GPU memory management and in clusters of heterogeneous GPUs of the different hardware computing capabilities and thus different scheduling requirements. An integrated GPU on the other hand

is present in most modern processors, tightly coupled with the system memory and might provide a cheap alternative to speed up computation without difficulties associated with discrete GPU deployment. Integrated GPUs on the embedded devices are used to satisfy growing computing requirements in robotics, avionics and medical diagnostics while maintaining or decreasing their weight, size and power requirements. A number of papers investigated the possibility of using integrated on-chip GPUs in high performance computation. Jasson[55] performs simple benchmarks such as vector addition and chained kernel execution and concludes that iGPU performs better for the small problem sizes, Ching et al[38] highlight the CPU-GPU data transfer bottleneck of the discrete GPU computation and perform benchmarks of the database management system performance. The other avenue is the embedded computing where GPU offload is seen as a way to conserve power — Grasso et al.[52] report 8.7x speedups with 32% of the power consumed by Mali GPU compared to Cortex-A15 core using their benchmark application. The recent Heterogeneous System Architecture(HSA) [21] is standard for a heterogeneous computation on CPUs, GPUs and other programmable and fixed-function devices with a high-bandwidth shared memory access. The HSA-based application interface can be deployed in both desktop and embedded settings. The current reference standard implementation is supported by AMD on A-series APU. This work attempts to investigate integrated GPU performance on machine learning algorithms such as k-nearest neighbours used in latency constrained applications such as network intrusion and fraud detection and an algorithm requiring iterative processing of the small workloads such as stochastic gradient descent. The performance is evaluated on discrete GPU and integrated GPU. The integrated GPU tests were performed using standard OpenCL drivers and a reference HSA Runtime for Linux platform. This work is organized as follows: Chapter 2 presents GPU programming concepts and notable frameworks, Chapter 3 describes system architecture, Chapter 4 describes several approaches to solving and parallelising k-nearest neighbours problem, Chapter

5 describes stochastic gradient descent and its parallelisation, Chapter 6 contains performance benchmarks and Chapter 7 presents conclusions and future work.

Chapter 2

General Purpose GPU Computing

2.1 Introduction

The modern Graphical Processing Units (GPU) greatly outpace CPUs in arithmetic throughput and memory bandwidth for data-parallel tasks. Since 2001 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 targeting specifically GPU architecture such as CUDA[66], 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 languages which contain hardware architecture specific language constructs. This chapter provides an overview of GPU architecture and most known programming frameworks and lists limitations of the traditional General Purpose GPU (GP GPU) programming. It also discusses the OpenCL 2.0 standard, which addresses some of the limitations and describes the Heterogeneous System Architecture (HSA), an optimized

platform architecture for OpenCL 2.0.

2.2 GPU Architecture

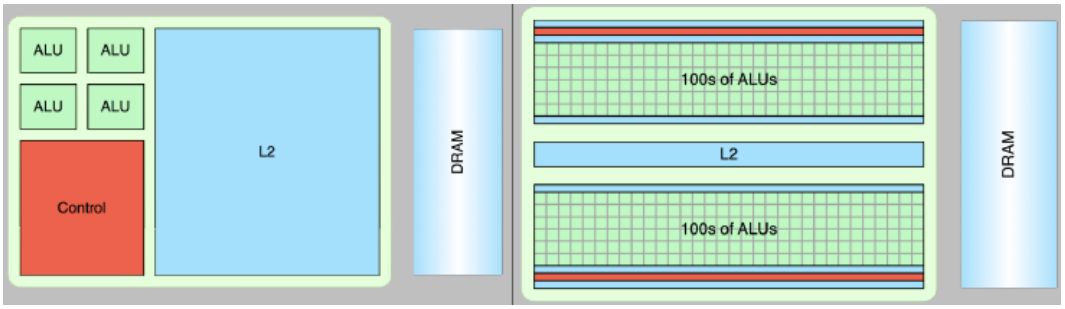


Figure 2.1: CPU versus GPU hardware architecture. Reproduced from NVIDIA GPU ARCHITECTURE and CUDA PROGRAMMING ENVIRONMENT by Alan Tatourian[87]

Figure 2.1 shows a side by side comparison between CPU and GPU architectures. The main differences between modern CPU and GPU architectures are the level of parallelism 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 scheduling unit — *wavefront* — for the NVIDIA GPU is 32 threads. Modern GPUs implement an SIMT (Single Instruction-Multiple 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][66]. The CPU thread is a heavy-weight entity which is centered around the 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 the 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

as shown in Figure 2.2.

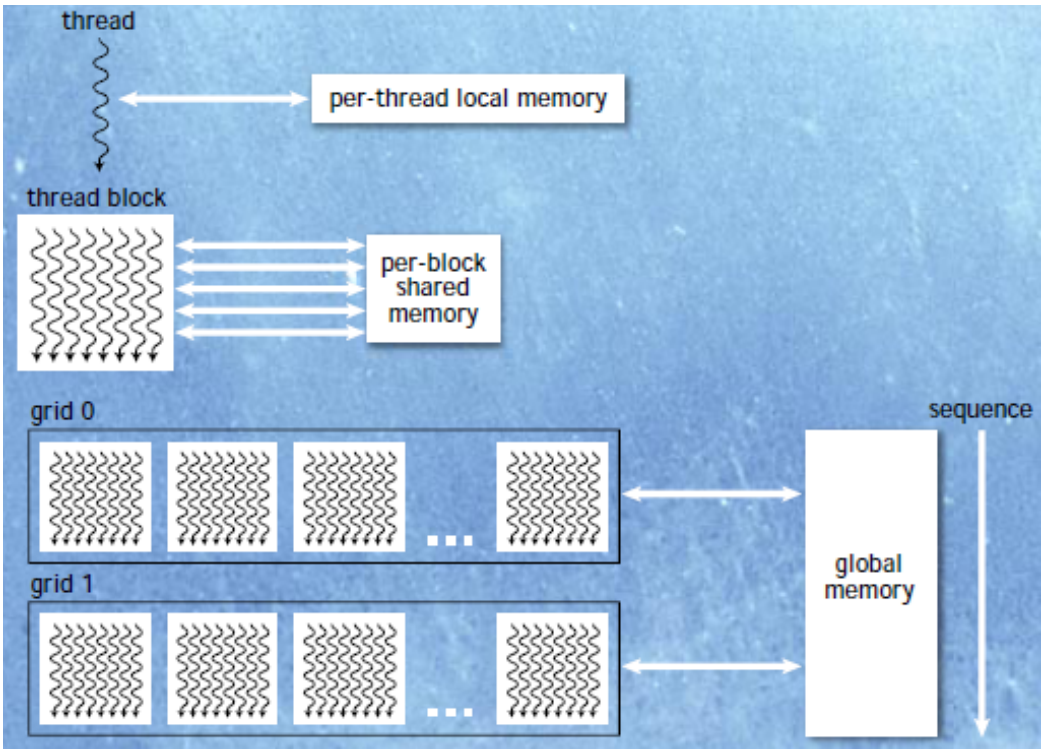


Figure 2.2: GPU Memory Tiers. Reproduced from NVIDIA GPU ARCHITECTURE and CUDA PROGRAMMING ENVIRONMENT by Alan Tatourian[87]

GPU programming uses the following abstractions:

- *Kernel* — a procedure executed by the group of concurrent *threads*
- *Thread* — a unit of execution

- *Threadblock/Workgroup* — 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 thread within the same *warp*. Whenever the threads within the same wavefront need to execute different instructions the divergence of control occurs and results in underload of the processing units and reduces performance. The branching should be reduced to wavefront granularity to avoid wasting execution cycles[83][66]. It should be noted that the wavefront size is a hardware specific feature and its optimization should be performed at the run-time.

2.3 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[66] 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 2.3.

2.3.1 High Level Languages

OpenACC and OpenMP are high level parallel programming frameworks that specify a set of annotations, environment variables and library routines for

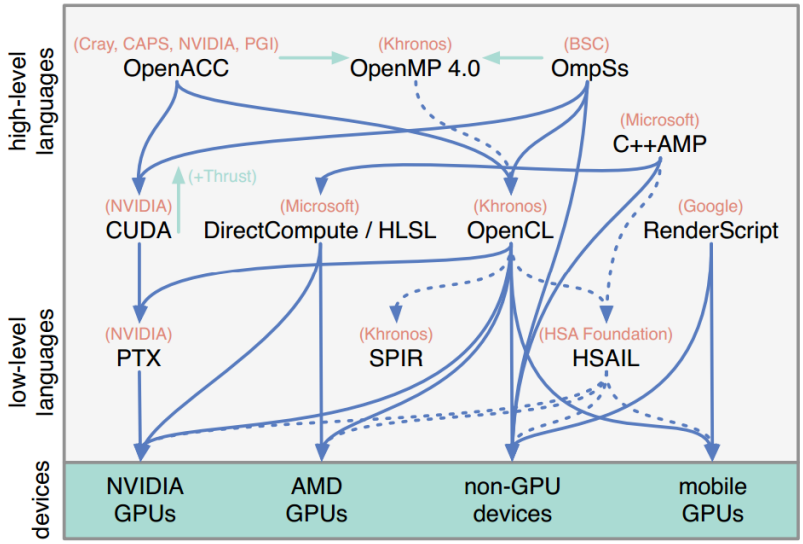


Figure 2.3: GP GPU technologies tree. Reproduced from C. Nugteren, Improving the Programmability of GPU Architecture, p. 21 [68]

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[73]. Aparapi provides a way to generate OpenCL kernel code from Java, theoretically allowing code which can be executed on CPU and offloaded to the GPU if needed[2]. Project Sumatra is an OpenJDK project which focuses on the development of Just-In-Time compiler extensions for Java Virtual Machine capable of offloading JDK 8 Stream API[12] computations to the GPU[15].

2.3.2 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 the number of scheduled threads and the size of the thread block[66]

- 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[85].

- 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 a proprietary scripting language first introduced in DirectX 9 that has limited support for double precision computing. DirectCompute only allows to specify the *workgroup* size at the compile time[6].

2.3.3 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 Heterogeneous System Architecture Intermediate Language (HSAIL)[21].

2.3.4 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 2.4 shows execution time of a kernel which assigns an 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. Maximum performance on the integrated GPU of AMD A8-7600 will be achieved when input size will exceed 1024 elements.

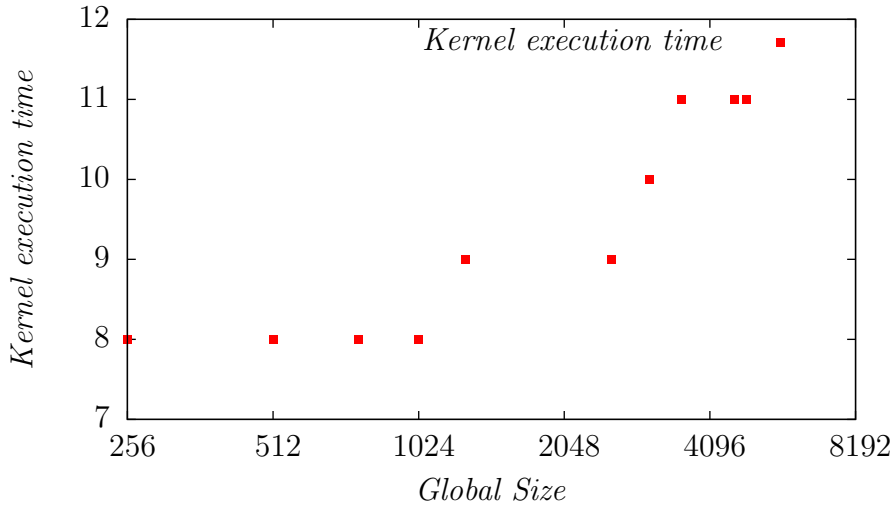


Figure 2.4: Empty OpenCL kernel execution time on integrated Radeon R7 GPU(μsec).

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[82]. 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.

2.4 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.

2.5 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 to 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].

2.5.1 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. A developer may opt to provide his own queue implementation. This feature allows to schedule CPU code execution from within the GPU kernel.

2.5.2 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.

2.5.3 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 reordering load and store operations[61]. 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.

Similar to the Java Memory Model[60] 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.

2.5.4 Implementation Notes

At the time of writing (February 2016) the HSA Runtime implementation ignores the sequential barrier flag requiring iterative algorithms to explicitly

synchronize kernel execution using signals to avoid starting a new kernel before the previous one finishes. Constant time is needed to setup kernel launch, e.g. for AMD A8-7600, it is 6 μsec using HSA.

2.6 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 latency-sensitive processing of data streams.

Chapter 3

System Architecture

We have implemented a data stream processing library that provides a GPU-accelerated implementation of machine learning algorithms such as nearest neighbours search and stochastic gradient descent. The library uses interfaces provided by Massively Online Analysis(MOA)[32] for implementation of the classification algorithms. The library uses existing linear algebra package ViennaCL[74] which allows multiple backends such as CUDA, OpenCL or CPU.

3.1 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. The 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.

MOA $Classifier$ interface defines methods $getVotesForInstance$ that predicts the class memberships for a given instance and $trainOnInstance$ that trains classifier incrementally using the given instance. The library provides the synchronous implementation of $getVotesForInstance$ and asynchronous implementation of $trainOnInstance$ to hide the latency of data transfer to the discrete card. Each call of $trainOnInstance$ schedules a data transfer and kernel execution on the same OpenCL command queue without waiting for the completion of the kernel. This allows to start the next transfer before training on the first instance is completed.

3.2 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 insufficient 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 better floating-point precision than double precision 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[76][53]. This work provides standard single or double precision floating-point imple-

mentation of the machine learning algorithms not covering the alternative floating point representations.

3.3 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.

3.4 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.

3.5 Conclusion

The library provided as a part of this work is heterogeneous. It uses a Java platform at the top level to interface with MOA and performs high-level computations. 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 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 4

k-Nearest Neighbours

4.1 Introduction

The k-Nearest Neighbours method[40] is a non-parametric method used for classification and regression. It computes for a given instance the distances 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[40]. 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.

4.2 Exhaustive Search

The exhaustive approach to Nearest Neighbour search is to compute the distance to each instance present in the sliding window. The computational complexity of the query $O(Nd)$ where N — the number of instances and d — the 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 Linear Algebra Subroutines(BLAS)[7][5][74]. The selection phase finds the nearest examples to the query out of all the computed distances. Sismanis et. al[82] 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 4.1. This approach allows obtaining better performance on low window sizes[65].

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.

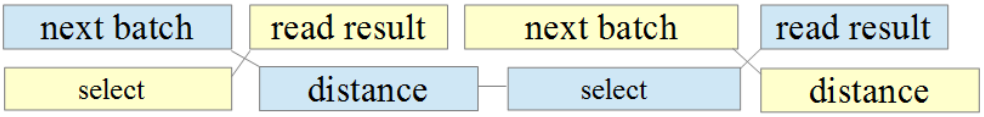


Figure 4.1: Interleaved distance calculation and sorting phases

4.2.1 Distance Calculation

A distance calculation operation is similiar to the one of matrix-vector multiplication — an element-wise operation is performed between matrix rows (sliding window) and a vector (test instance). Best all around distance calculation should apply different strategies depending on the window size and number of attributes[84]. Figure 4.2 shows the alternatives and the experimental results on NVIDIA C2050 card. A single thread processing several rows achieves a full GPU load for the large windows sizes (>10^4) and smaller window sizes require different strategies. For small instance sizes (<100) the kernel completes fast enough for the one thread per row to be the best solution. Larger instances will require several threads assigned to an instance each processing single attribute. In the extreme case of a very large instance size (>10^4) and small windows (<100), several columns should be processed by a single thread.

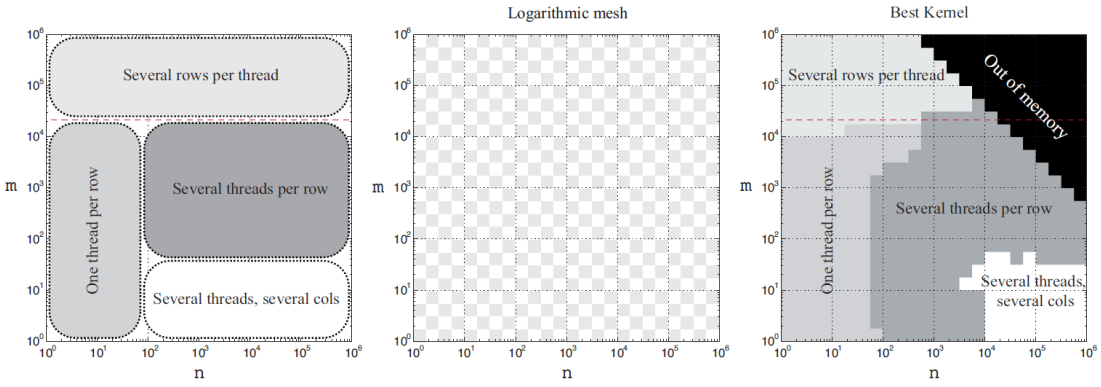


Figure 4.2: Left; Four matrix-vector multiplication kernels designed to perform well at different shapes m n of the matrix. Middle; Tuning mesh. Right; Best kernel in practice. The dashed line indicates the minimum 21504 rows needed in the matrix for full occupancy of the Nvidia Tesla C2050 card in a one-thread-per-row kernel. Note the logarithmic axes. Reproduced from High-Performance Matrix-Vector Multiplication on the GPU by Hans Henrik Brandenborg Sørensen[84]

The distance calculation primitive provided as part of this master thesis uses one thread per row approach for small dimensionalities and one workgroup per instance for larger ones with the different switch value due to the different

hardware configuration as described in Section 6.3.1.

4.2.2 Selection

Alabi, et.al[26] evaluated different selection strategies based on bucket sort algorithm and Merril-Grimshaw[63] implementation of radix sort.

Figure 4.3 shows performance of different selection strategies - merge sort from AMD Bolt library[3], bitonic sort similar to reference AMD implementation and radix select based on Alabi, et. al. implementation[26]. 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.

4.2.3 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.

4.3 Exact Clustering Methods

Clustering techniques are widely used to limit the number of distance calculations needed for the nearest neighbour search. Space partitioning by vector dimensions is used by the k-d tree method[49], the random projection tree[48] provides a data structure splitting the search space along random vectors. Metric trees such as ball tree[69], cover tree[31], random ball cover[36] provide solutions for finding nearest neighbours in general metric space by organizing data points in groups around some centroids.

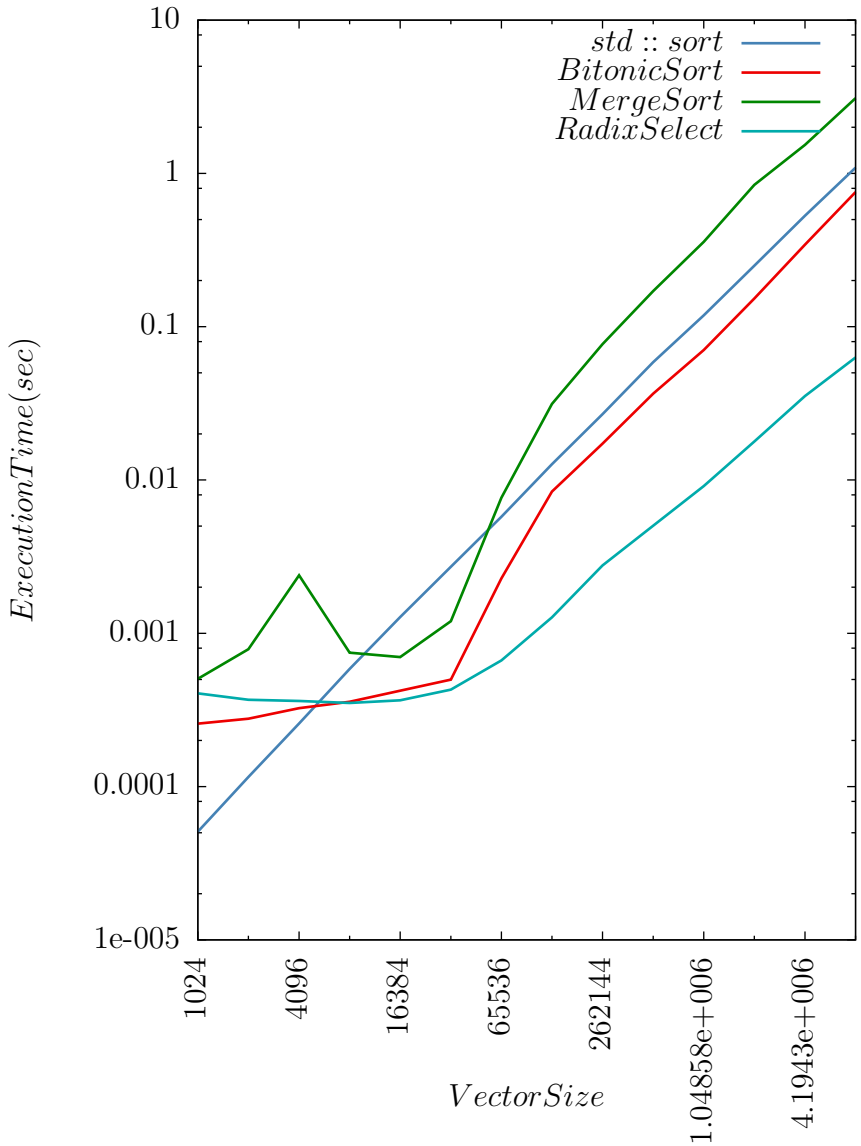


Figure 4.3: Selection Algorithm Performance for K=128. Test configuration GPU R9 390, CPU AMD A8-7600, AMD Catalyst version 15.20.

4.3.1 k-d Tree

The *k-d* tree [49] 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 are reached. The tree can be constructed in $O(nlogn)$ time and occupies linear space. Weber *etal*[88] 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 fewer points than exhaustive search.

The listing of the *k-d* tree construction and nearest neighbours search pseudocode is shown in the Figure 4.4. Parallel *k-d* tree construction on GPU utilizes breadth-first approach[89][80] — 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 4.4 does not benefit much from the GP GPU parallelism due to the branch divergence and irregular memory access patterns[51]. The *k-d* tree search approach presented by Gieske et. al[51] focuses on the parallel execution of nearest neighbour queries in a lazy fashion. The query points are accumulated in the leaf nodes of the kd-tree until enough of them 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. Another approach would be to compute distances to the leaf nodes split planes[89] to provide a short-list of the leaf nodes for k-nearest neighbours search.

4.3.2 Random Projection Trees

The *k-d* tree provides an effective partitioning mechanism for low data dimensionality but suffers in higher dimensions[88]. Many machine learning problems that are expressed in high dimensional space have lower intrinsic dimension as shown in Figure 4.5. Random projection tree exploits this fact 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 4.6 [48]. 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 4.7 and 4.8. Figure 4.9 illustrates node selection for random projection tree median split rule.

```
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 4.4: k-d tree construction and NN-search pseudocode

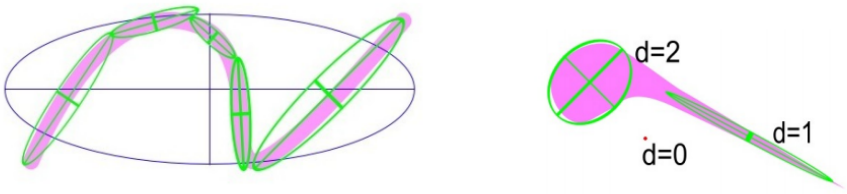


Figure 4.5: Distributions with low intrinsic dimension. The purple areas in these figures indicate regions in which the density of the data is significant, while the complementary white areas indicate areas where data density is very low. The left figure depicts data concentrated near a one-dimensional 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.[48]

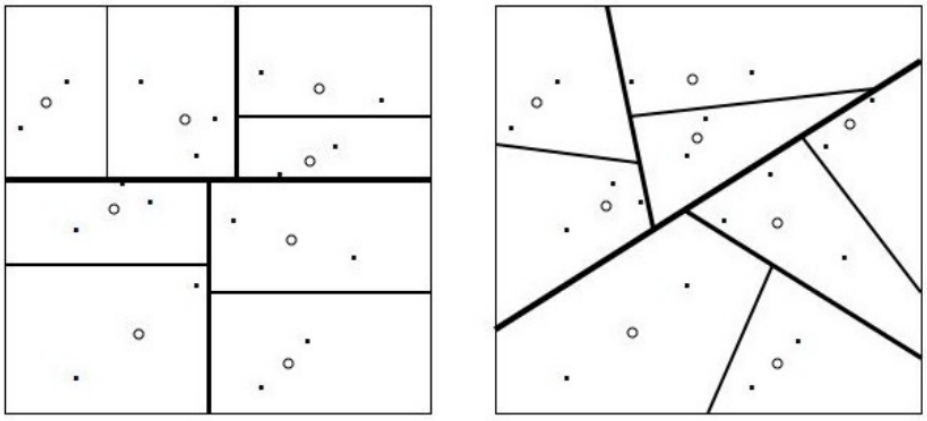


Figure 4.6: Left: Partitioning produced by k-d tree. Right: Partitioning produced by Random Projection Tree. Reproduced from Learning the structure of manifolds using random projections by Freund Yoav et al.[48]

```
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 4.7: Random Projection Tree Pseudocode - Random Tree Max [41]

The projections along the 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[56] and be implemented more efficiently than the naive approach.

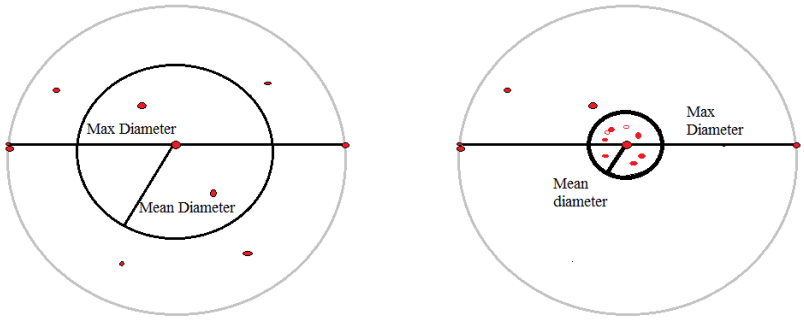
4.3.3 Random Projection

The seminal paper by Johnson and Lindenstrauss[56] 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[56] has shown that Johnson-Lindenstrauss condition holds for matrices with the following properties:

- Spherical symmetry — For any orthogonal matrix $A \in O(d)$, Φ multi-

```
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 4.8: Random Projection Tree Pseudocode - Random Projection Tree Median Split[41]



(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 4.9: Random projection tree median split node types

plied by A and Φ have the same distribution.

- Orthogonality — rows are orthogonal to each other
- Normality — the rows are unit-length vectors

[24]

The lower bound of k was refined in several papers[47][42][54][23] with Dasgupta and Gupta[42] 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 a 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 require a simple approach to construct Φ and a way to compute projection faster than a naive multiplication of data point by $k \times n$ matrix. Achlioptas[23] achieved relatively sparse transformation matrix for random projection by proving that the Johnson-Lindenstrauss condition holds if elements of the projection matrix are chosen independently according to the following dis-

tribution:

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

where n — source dimension and P — probability. This method provides a 3-fold speedup over the original one [56], since 2/3 of the transformation matrix elements are zero. Nir Ailon and Edo Liberty[25] 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 making 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[24] can be replaced by subsampled Fourier transform[25]. Nir Ailon and Edo Liberty define $k = O(\delta^{-4}log(N)log^4n)$, where N — number of instances, n — source and k — projected dimensionality, that will preserve input vector norms by a given relative error δ[25]. 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 4.10 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 re-

quire to store transformation matrix and is thus capable of projecting higher dimensional data on the same hardware.

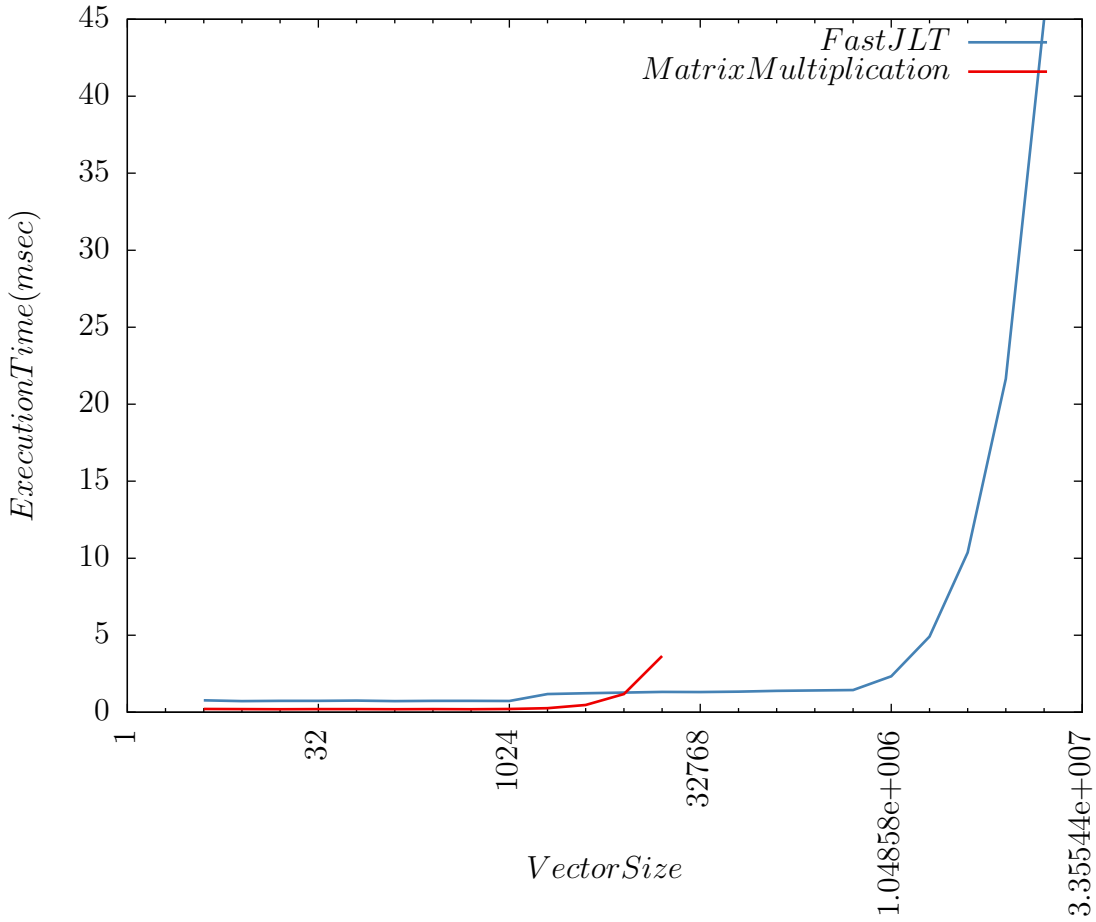


Figure 4.10: Fast Johnson-Lindenstrauss transform[25] vs. sparse matrix implementation[23] using ViennaCL. Test configuration GPU R9 390, CPU AMD A8-7600, AMD Catalyst version 15.20. The matrix multiplication test was aborted due to the out of memory error.

4.3.4 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 a large node as shown in Figure 4.9 to obtain approximate solution.

4.4 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[88][29] 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 are at most $c > 1$ times greater than the distance to the closest neighbour. The approximate solution can be used to find the 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[72] and data set ordering using space filling curves[72][57][58] to improve query performance.

4.4.1 Locality Sensitive Hashing

Locality Sensitive Hashing[54] 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 probability $P_1 > P_2$. The LSH algorithm uses a concatenation of $L \ll d$ such functions to increase the difference between P_1 and P_2[54]. Initially it was proposed to use Hamming distance as this function satisfies all required properties[54]. Later it was shown that other families of hash functions such as l_p distance[43], Jaccard coefficient [34][35] and angular distance(random projection)[37] 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.[28] investigated several approaches to the hash table construction and retrieval on GPUs and has shown that the suitability of a particular method is hardware dependent, the iterative data structure content modification is difficult since insertion failure results in a full hash table rebuild, which is offset by the fast rate of construction. For instance, Alcantra et al[27] use a 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[70] 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 remains a better option if queries are sorted and executed in bulk [28].

4.5 Space Filling Curves

Space filling curves[75] 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[58][57][72] utilize z-order curve introduced by G. Morton in 1966 — an ordered list of numbers composed by interleaving bits of instance attributes[64]. 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 4.11. A z-order 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 4.11 the z-order curve has jumps. To compensate for them existing GP GPU algorithms generate several curves from the input data shifted several times by some random vectors[58][57] to move the points into its locality-

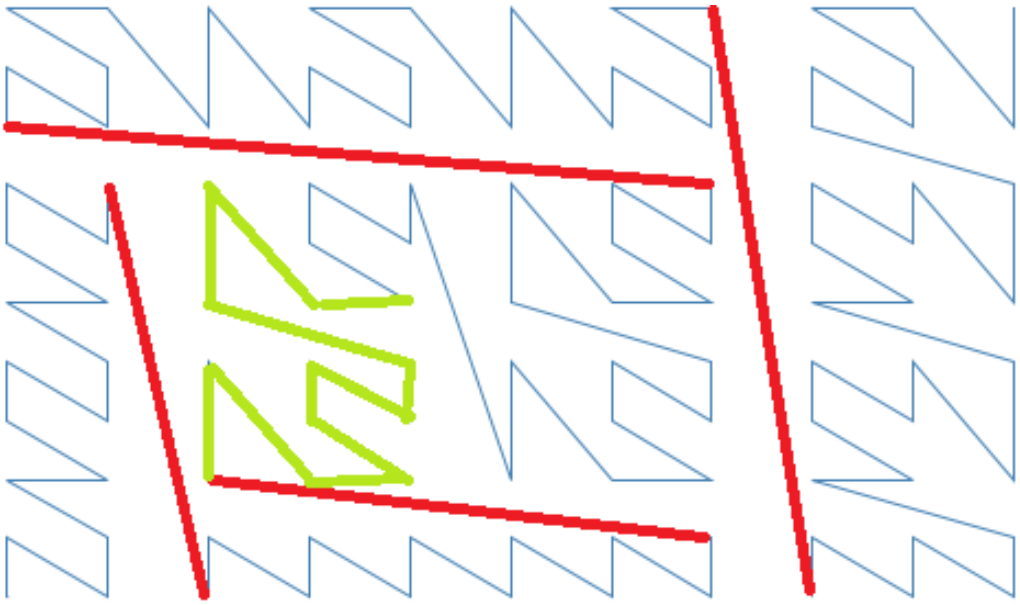


Figure 4.11: Z-Order Curve. Red lines highlight some region jumps. Green shows locality-preserving region.

preserving regions. The shift search method by Li et al.[57] experimentally quantifies the shift value.

Sieranoja S.[81] generalized a z-order lookup table mapping algorithm for an arbitrary number of dimensions. It can be translated into GPU implementation with minimal changes.

A Morton code has the bit-length of the input data. A z-order curve performance degrades as dimensionality increases due to the increasing cost of the comparison operation needed to sort Morton codes and search in the resulting z-order curve. Nearest neighbours methods applied to a high-dimensional data may use dimensionality reduction techniques that preserve relative distances to improve the z-order curve performance.

4.6 k-Nearest Neighbours using HSA architecture

The HSA architecture might allow taking 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[89][51]. In Chapter 6 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 can be built in the breadth-first manner similar to existing GPU implementations of k-d tree algorithm. This approach requires pass over the whole sliding window for each update. In Chapter 6 we will investigate whether the online update of the random projection tree benefits 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[57]. The cost of GPU-host transfer, in this case, is prohibitive and Li et al[57] work around it by executing multiple queries at once so that it is more beneficial to perform sort as opposed to search. In Chapter 6 we will investigate whether it is efficient to perform single queries using z-order lists using approach by Li et al[57] 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.

4.7 Conclusion

The current approach to using GP GPU for k-nearest neighbours problem is to design algorithms tailored for the target GPU platform. In Chapter 6 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 6 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 5

Stochastic Gradient Descent

5.1 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 stochastic gradient descent 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 an HSA algorithm implementation.

5.2 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.

Langford et al[90] 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 [90]. This highlights a problem for any parallel implementation of SGD — one has to minimize communication to achieve maximum computation speedup without introducing a critical delay that will hurt convergence of the algorithm. The subsequent works [77][50][91][45][71] provided a way to balance the parallelism and delay either by *model parallelism* — partitioning data into independent sets and processing them in parallel [50][91], delaying update communication[77][45][71], or by removing a synchronization requirement for parameters update[67]. The state of the art cluster methods use a combination of all those techniques[45][71] 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[44] 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 ex-

cellent building blocks for future heterogeneous processors.

5.3 Hogwild!

The *Hogwild!* algorithm[67] 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 a non-zero gradient using atomic add function. The update may be implemented as either a *compare-and-swap* operation ensuring that individual updates are not lost or as a replacement with a possibility to lose a certain portion of updates to 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[46] and Netflix prize[30] datasets [67].

5.3.1 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.

5.3.2 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 $\frac{K}{\beta}$ iterations.

5.4 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 the 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 the next iteration. 1Bit SGD[77] approaches the communication bottleneck from a 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 quantized 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 [77].

This approach allows nearly linear speed-ups in a cluster setting[86] as opposed to previous results such as[78].

5.5 Stochastic Gradient Descent Using OpenCL/HSA Architecture

This work implements the Hogwild! algorithm for linear models on Heterogeneous 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 quantization error is minimized, and then applies cumulative update.

The algorithm is parametrized by a number of mini-batches 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 mini-batches. 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 is the learning rate, using the atomic add function if the update value exceeds the threshold, or added to the corresponding element of residual error matrix E. Based on E and the iteration number an average quantization error is calculated for each column and τ used for the next iteration is updated. The pseudocode for the algorithm is presented in Figure 5.1. 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.

```
// X − input data
// Es,El − residual error for "too small" and "too large" cases
// W − weights
// lambda − weight update function
// g(x) − gradient function
// tau − quantization parameter vector (tau[i] > 0)
// temp_weights − weight updates used during training step (integer)
tau training_step(X, Es, El, lambda, g, tau)
{
  parallel_for (mini_batch in X)
  {
    W = read_weights();
    mini_batch = g(mini_batch, W); // compute gradients for each example
    vector minibatch_gradients = average_gradients(individual_gradients) +
    Es[mini_batch] + El[mini_batch];

    parallel_for ( mg[i] in minibatch_gradients)
    {
      if (abs(mg[i]) < tau[i])
      {
        // update "too small" error
        Es[mini_batch] = mg[i];
      }
      else
      {
        lambda(i, sign(mg));
        // update "too large" error
        El[mini_batch] += mg[i] < 0 ? mg[i] + tau[i] : mg[i] − tau[i];
      }
    }
  }
  commit_weights();
  return update_tau(tau,Es, El);
}

// atomic update of temporary weights
lambda(i, sign)
{
  atomic_add(temp_weights[i],sign);
}

read_weights()
{
  return W + learning_rate ∗ temp_weights ∗ tau;
}

commit_weights()
{
  W = W + learning_rate ∗ temp_weights∗tau;
}
```

Figure 5.1: Hogwild!-based stochastic gradient descent

5.6 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. Chapter 6 will compare runtimes of the algorithm on integrated and discrete GPU using HSA and OpenCL platforms respectively.

Chapter 6

Experimental Results

6.1 Experiment Setup

The experiments were performed using AMD A8-7600 Radeon R7 10 Compute Cores 4C+6G CPU and integrated GPU, 16Gb DDR3 1600 RAM and AMD Radeon R9 390 as a discrete GPU. The hardware summary is given in Table 6.1. 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 operating systems used were Window 8.1 and Ubuntu Linux 14.04.

Device	*Hawaii*	*Spectre*
Revision	Graphics Core Next 1.1	
Computing Cores	2560	384
Clock Speed	1010Mhz	720Mhz
Bus Width	512Bit	128Bit
Memory Clock	1500Mhz	800Mhz
Thermal Design Power (TDP)	275W	65W

Table 6.1: Hardware Summary

6.2 Experiment Data

The experiments used Infinimnist MNIST dataset generator[59], synthetic data streams produced by MOA[32] and the 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 nonnull 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.

6.3 k-Nearest Neighbours

This section details experimental results obtained for the implemented k-nearest neighbours algorithms: exhaustive search, k-d tree, random projection tree and z-order search. k-Nearest Neighbours algorithms were evaluated by periodically testing on heldout set with $k = 5$ unless otherwise specified. In MOA this functionality is implemented by *EvaluatePeriodicHeldOutTest* task.

6.3.1 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 for the Spectre GPU and single float precision were measured as shown in Figure 6.1 where 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 results in global memory read requests serialized over the same memory channel. This can be solved by

the introduction of an extra column to the attribute vector to avoid the large power of two strides.

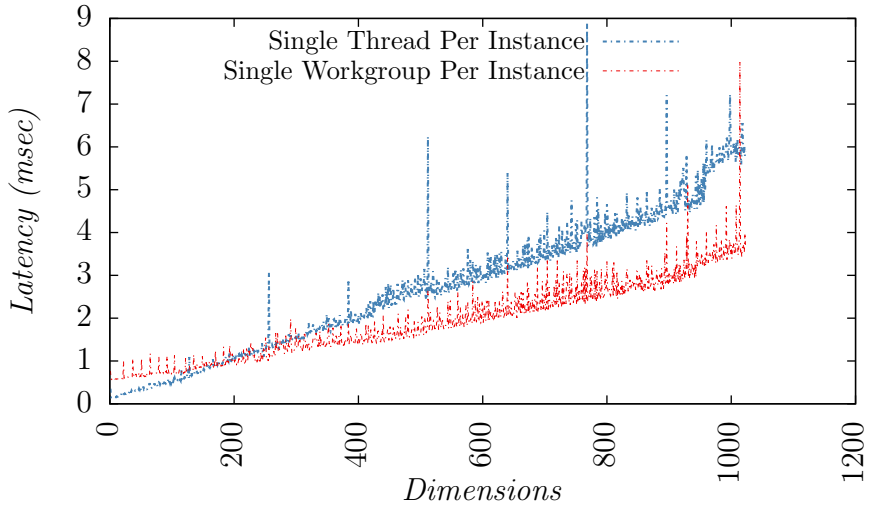


Figure 6.1: Latency of the distance calculation for an arbitrary number of dimensions

Figures 6.2,6.3 show the latency of the algorithm and speedup compared to the reference MOA implementation. The test was performed on the data with a fixed number of attributes and window size increasing by a factor of two. The single-threaded implementation shows a linear increase of the classification latency whereas GPU-based implementations show a step-wise increase. Low window sizes show constant latency due to the hardware underutilization. Large window performance degrades faster than linear and is suboptimal. To accommodate for large sliding window sizes the distance implementation have to include the "single thread per multiple rows" scheme as per [84]. The best window size for the discrete GPU is lower than the one for the integrated one due to the transfer overhead — a first attempt to classify an instance waits for all the pending CPU-GPU data transfers to complete.

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

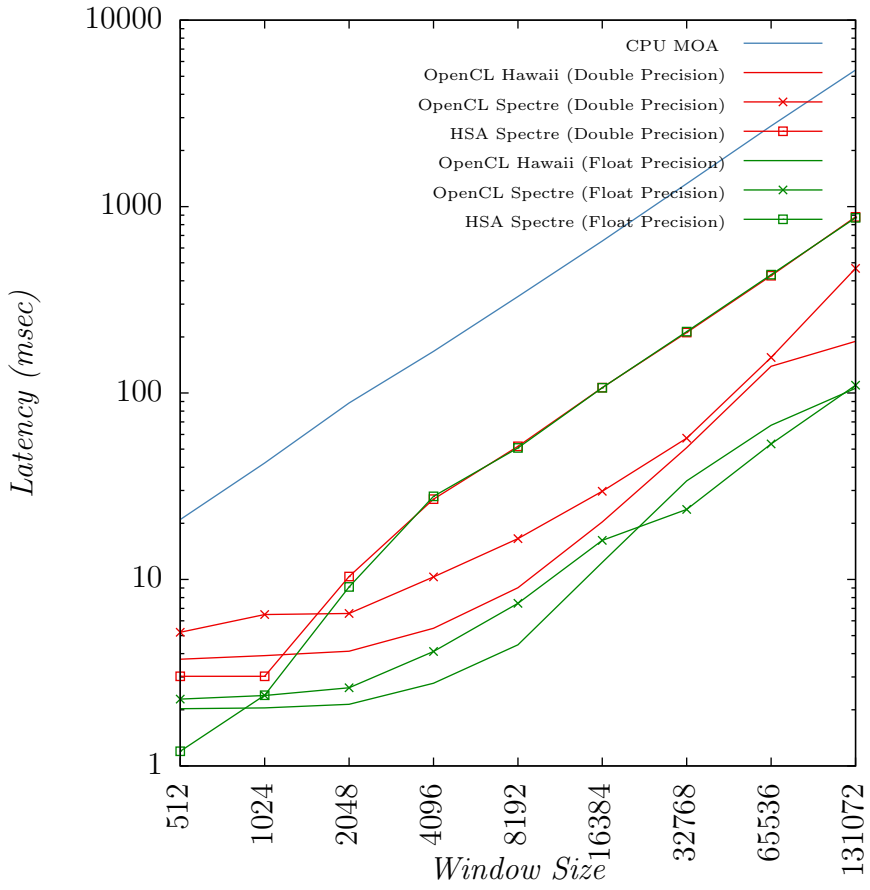


Figure 6.2: Latency of k-NN exhaustive search on Infinimnist stream

as shown in Figure 6.4 and can be used to process small windows.

The HSA implementation shows better performance on small window sizes ($\leq$ 1024 for MNIST) where discrete GPU is affected by the fixed data transfer cost. It should follow the same pattern as the OpenCL implementation after the breakeven point, but in fact, the latency scales linearly with the number of scheduled work items. 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 optimizations present in the OpenCL driver. The cost of scheduling additional workgroups is linear according to Figure . The additional test was performed to compare the performance of the OpenCL and HSA drivers in regards to the number of scheduled workgroups. A kernel without any operations was executed with a different number of scheduled workgroups. The results are shown in the Figure 6.8. It appears that un-

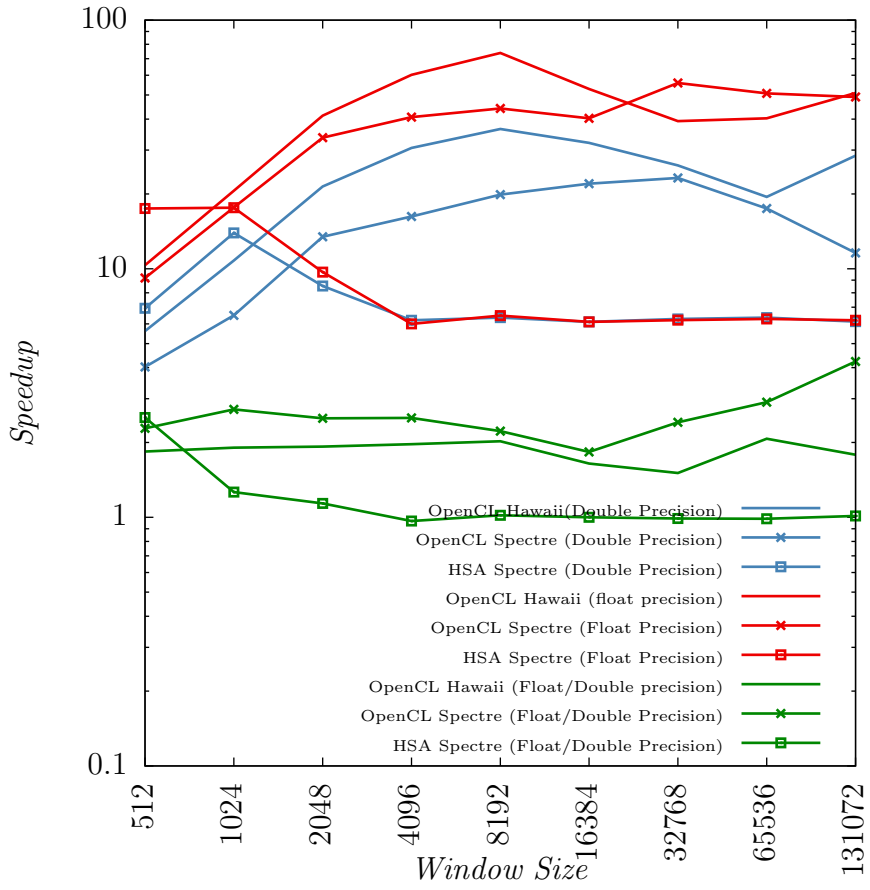


Figure 6.3: Speedups of kNN Exhaustive Search on Infinimnist Stream

like OpenCL HSA stack schedules no operation wavefronts for execution thus causing performance degradation.

6.3.2 k-d Tree

The reference MOA implementation 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 6.5 shows training and evaluation latency on Inifinimnist dataset and Figures 6.6, 6.7 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

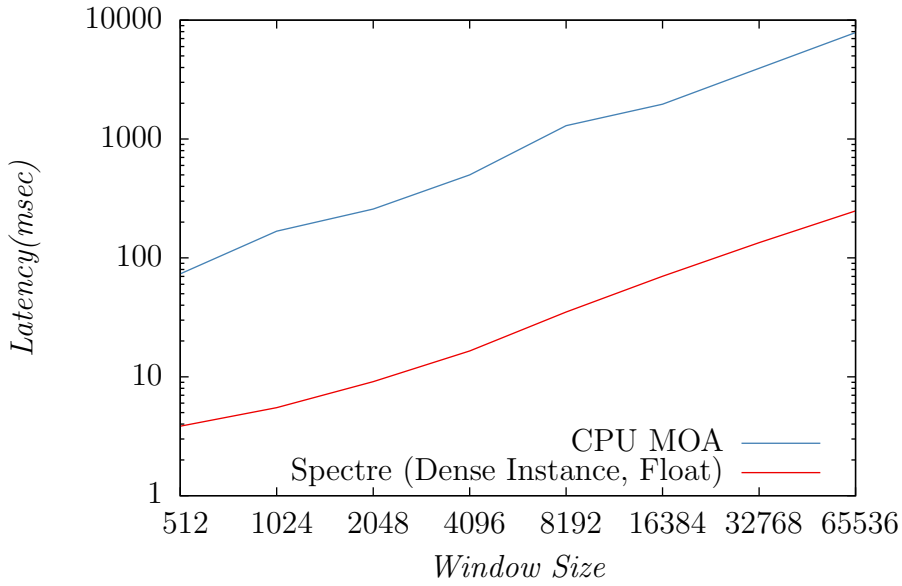


Figure 6.4: LinearNN Latency of CPU MOA and Spectre OpenCL implementations on Twitter Stream

a parallel fashion, one workgroup per attribute peaking out at 700x speedup for the optimal window size on Spectre integrated GPU. The HSA backend underperformance is consistent with the exhaustive search tests.

6.3.3 Random Projection Tree

The random projection tree performance on Infinimnist is shown in Figures 6.9, 6.10, 6.11 and 6.12.

The random projection tree training has lower speedup than the 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. Figure 6.14 shows comparative training performance of k-d tree and random projection tree methods. The random projection tree training involves less CPU intensive easily parallelised operations and thus while overall performance is better for the random projection tree, the speedup from GPU parallelisation is far worse than in the k-d tree

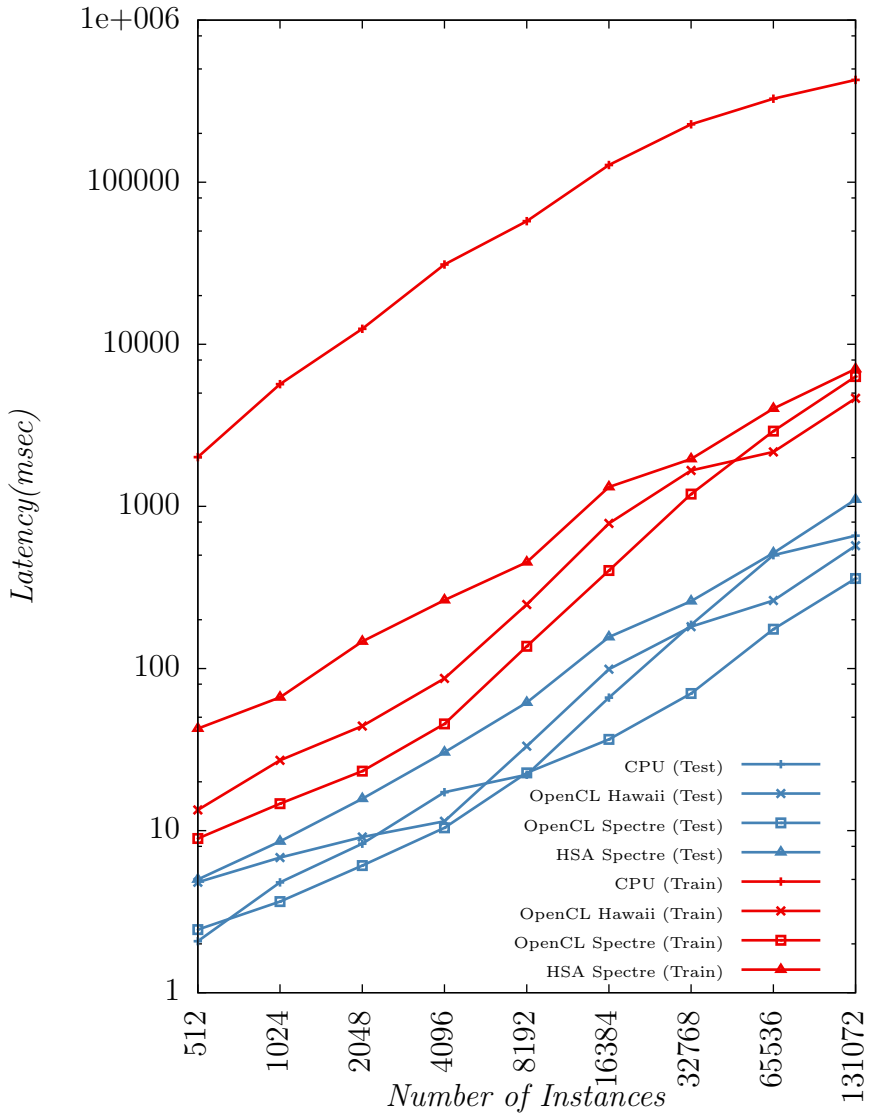


Figure 6.5: k-d Tree training and evaluation latency on Infinimnist

case. The matrix multiplication method of random projection outperformed FJLT in line with the benchmark in Figure 4.10 for the selected number of attributes (784).

6.3.4 Z-Order Search

Z-Order search provides a compromise in regard to testing and training speed between exhaustive search and exact space partitioning methods.The mapping procedure speedup compared to single-threaded CPU implementation is shown in the Figure 6.15. Figure 6.14 shows training times for Z-Order search and tree-based methods described above. The experiment used a simplistic ap-

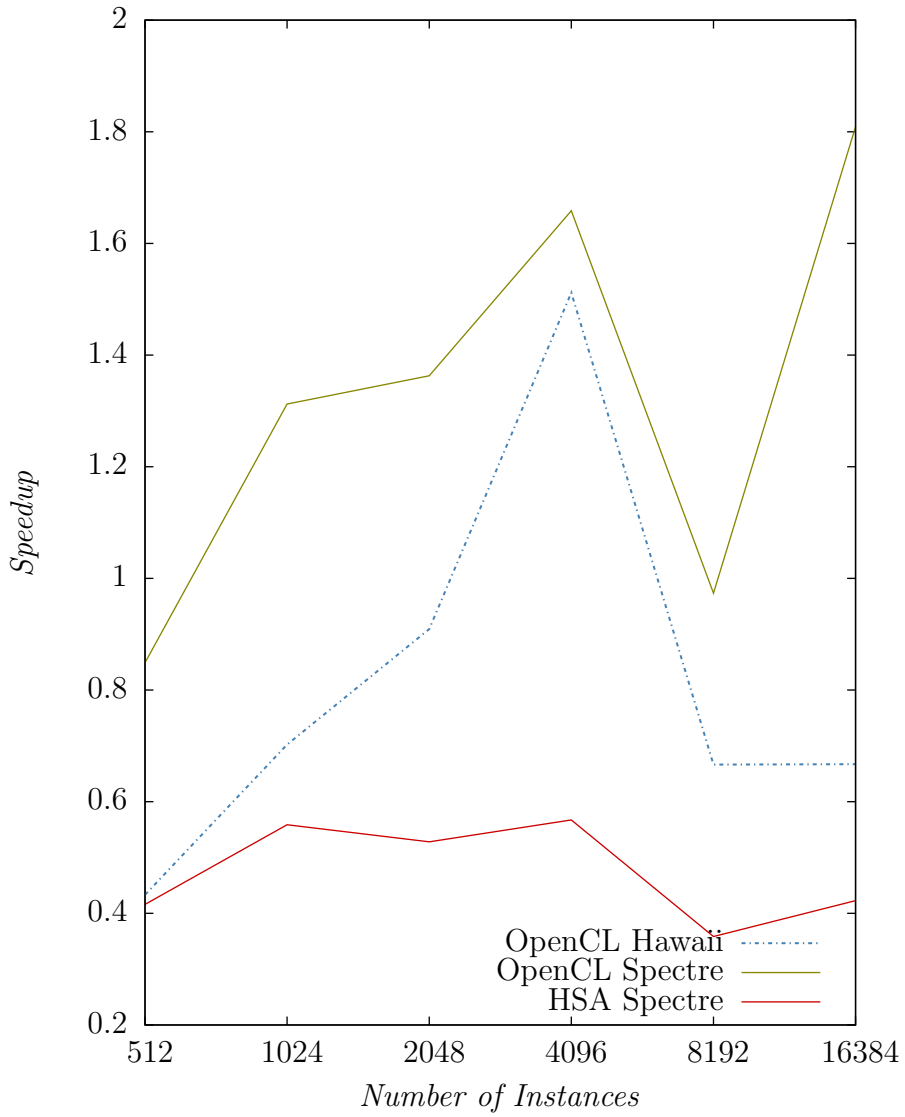


Figure 6.6: k-d Tree evaluation speedup on Infinimnist

proach to searching z-order curves - the data dimensionality was reduced using random projection, a z-order computed and a fixed 256 instances region was sampled around the query location. The results were concatenated to provide the result in line with [57]. The test was performed using 16384 instances sliding window on Infinimnist stream and the results are presented in Figure 6.16. The method accuracy can be improved if instead of the search in the fixed window the curve is iteratively traversed in batch increments until stop conditions such as closest point in the current batch being outside the query radius or a time limit are reached. The test was performed using RandomRBFGenerator data stream and Figure 6.13 summarizes this approach.

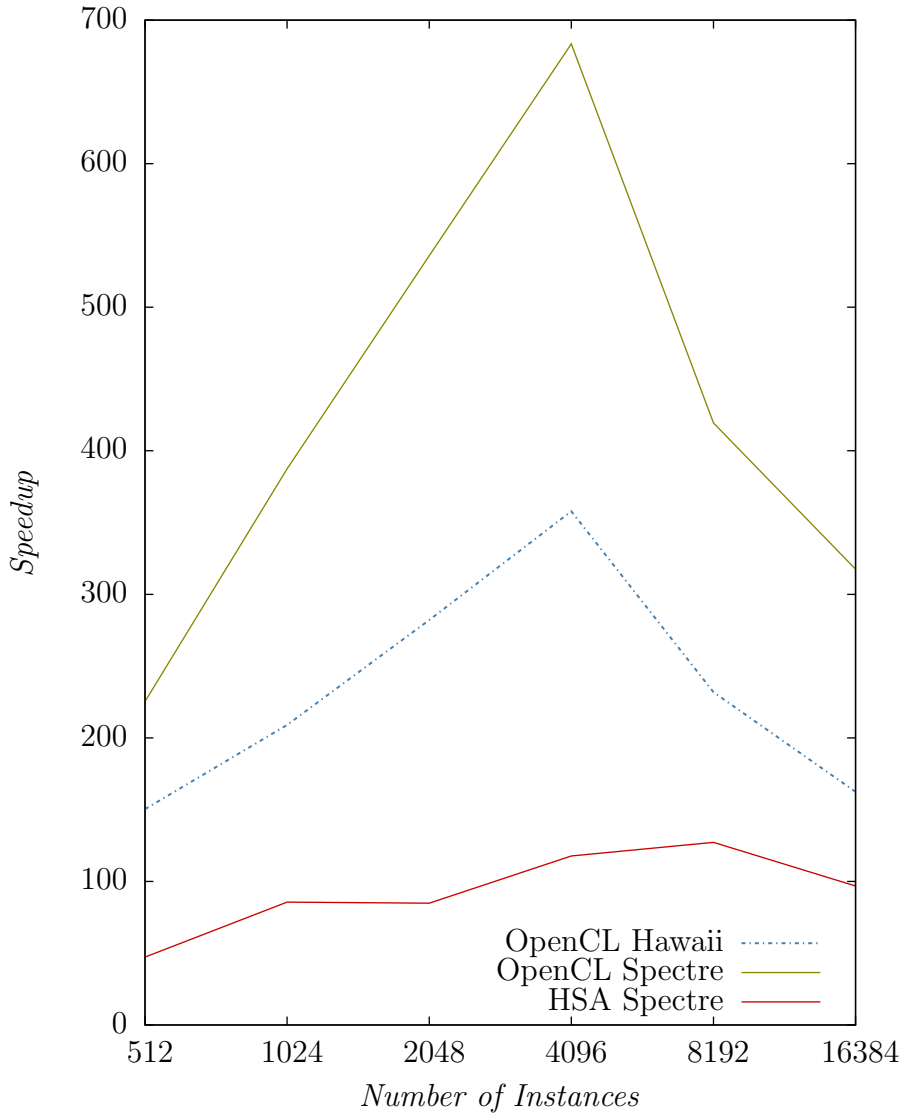


Figure 6.7: k-d Tree training speedup on Infinimnist

6.4 Stochastic Gradient Descent

This work implements Hogwild![67] 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 out, updated and atomically stored. It is possible to loose individual

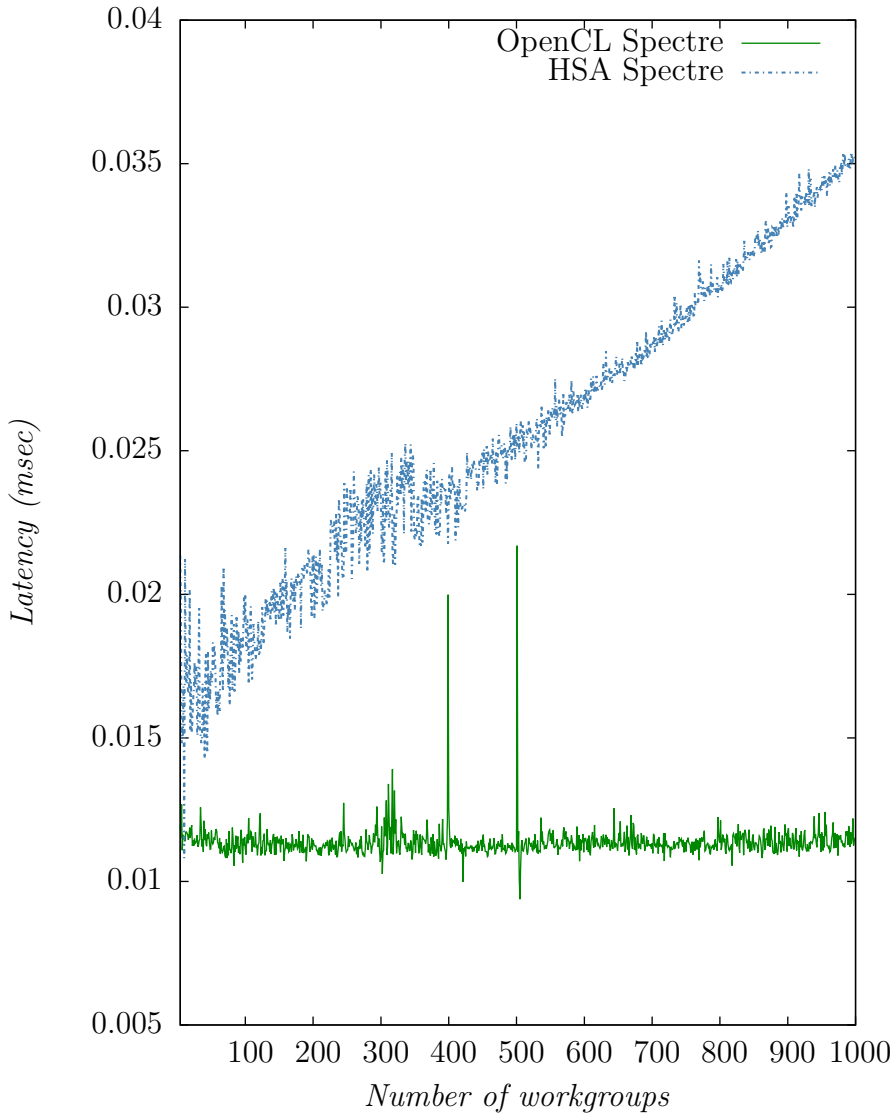


Figure 6.8: No operation kernel execution

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 6.18 shows the 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 sparseness of the data stream. The Figure 6.17 shows that even for dense data the error is not significantly affected even if the workers are not synchronized.

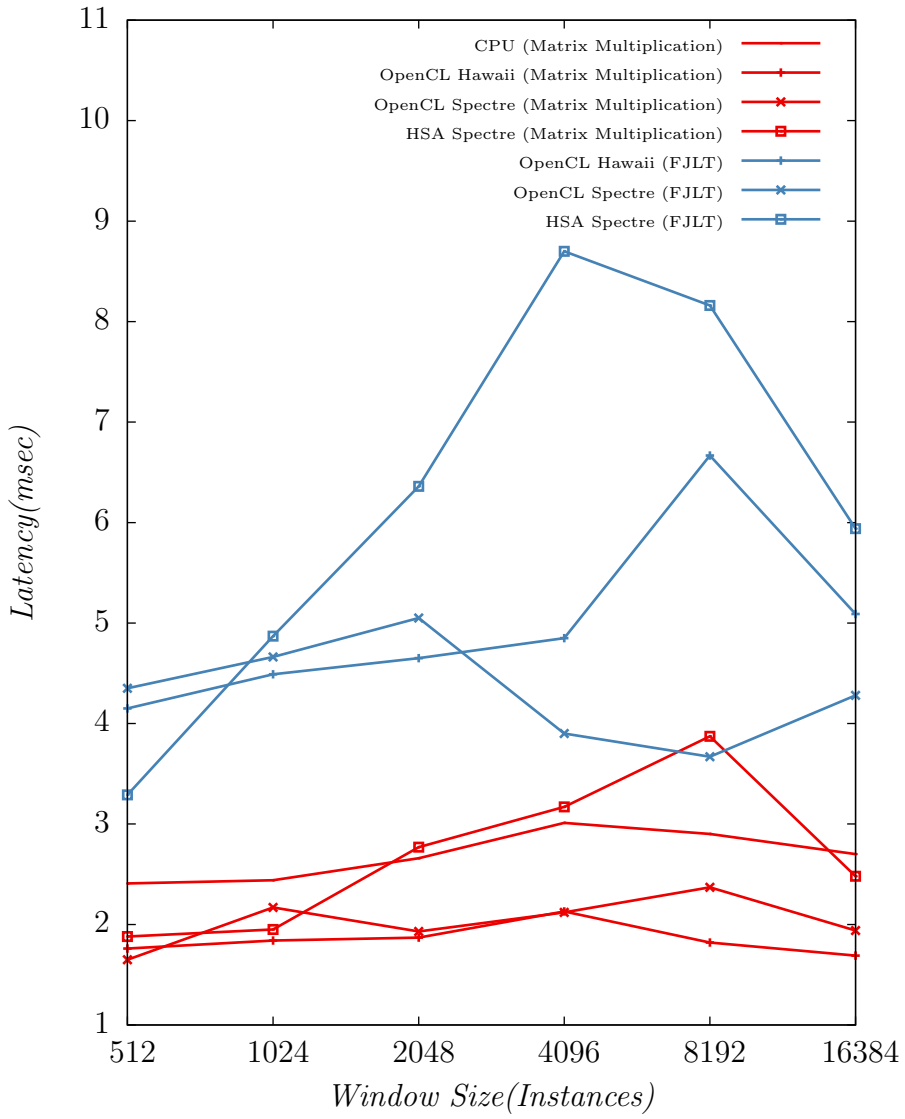


Figure 6.9: Random Projection Tree evaluation latency on Infinimnist dataset

The minibatch size used in further tests was set to 64 as per Figure 6.19. The performance of direct updater and 1bit updater with 1 synchronization per 1000 iterations was identical. Figure 6.20 shows performance depending on the 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.

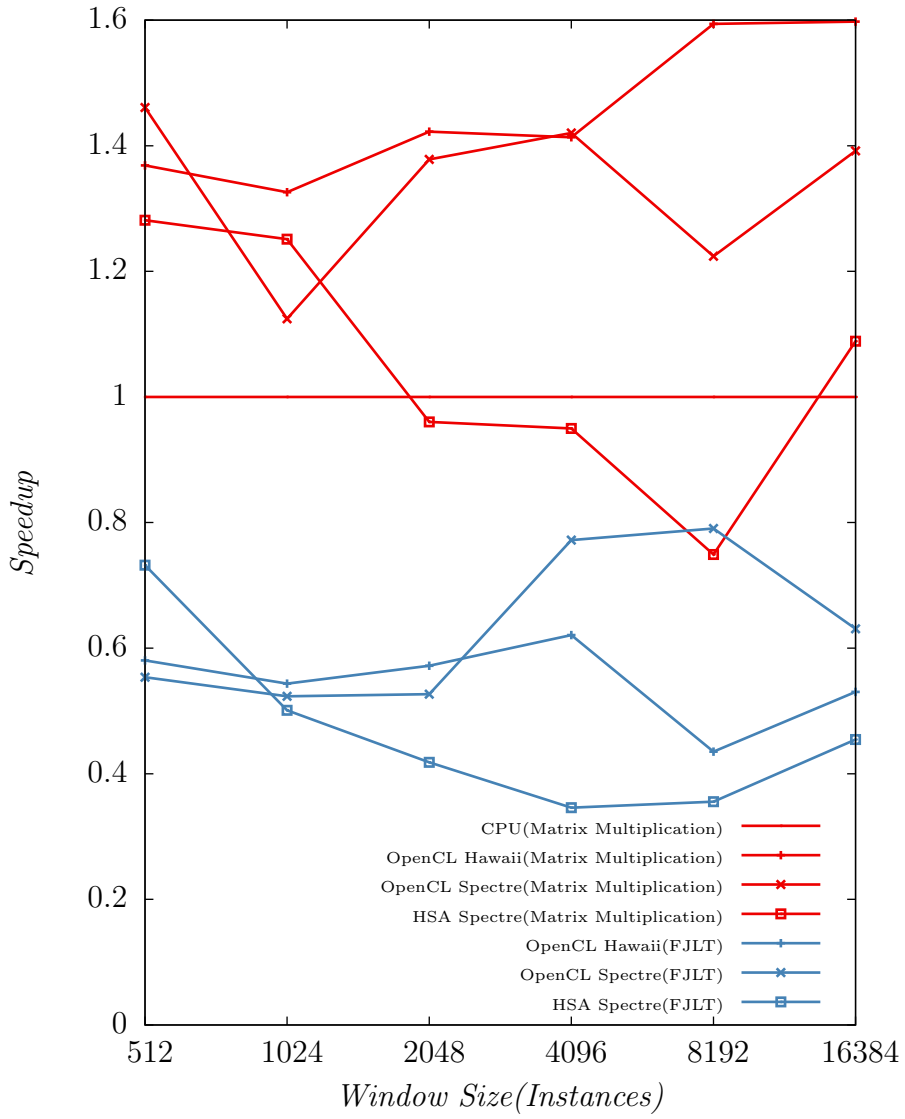


Figure 6.10: Random Projection Tree evaluation speedup on Infinimnist dataset

6.5 GPU Utilization and Estimated Power Draw

The average GPU utilization and estimation of the power draw obtained during training and evaluation on OpenCL platform are shown in the Table 6.2. The utilization tests were performed with window size 8192 on Infinimnist dataset. Z-Order mapping used 64 curves with 256 bytes Morton codes. SGD test was performed using the single worker and 64 instances minibatch size.The comparison with HSA driver was not performed due to the lack of monitoring software.

Table 6.3 shows the comparison between speedup and power draw of Hawaii

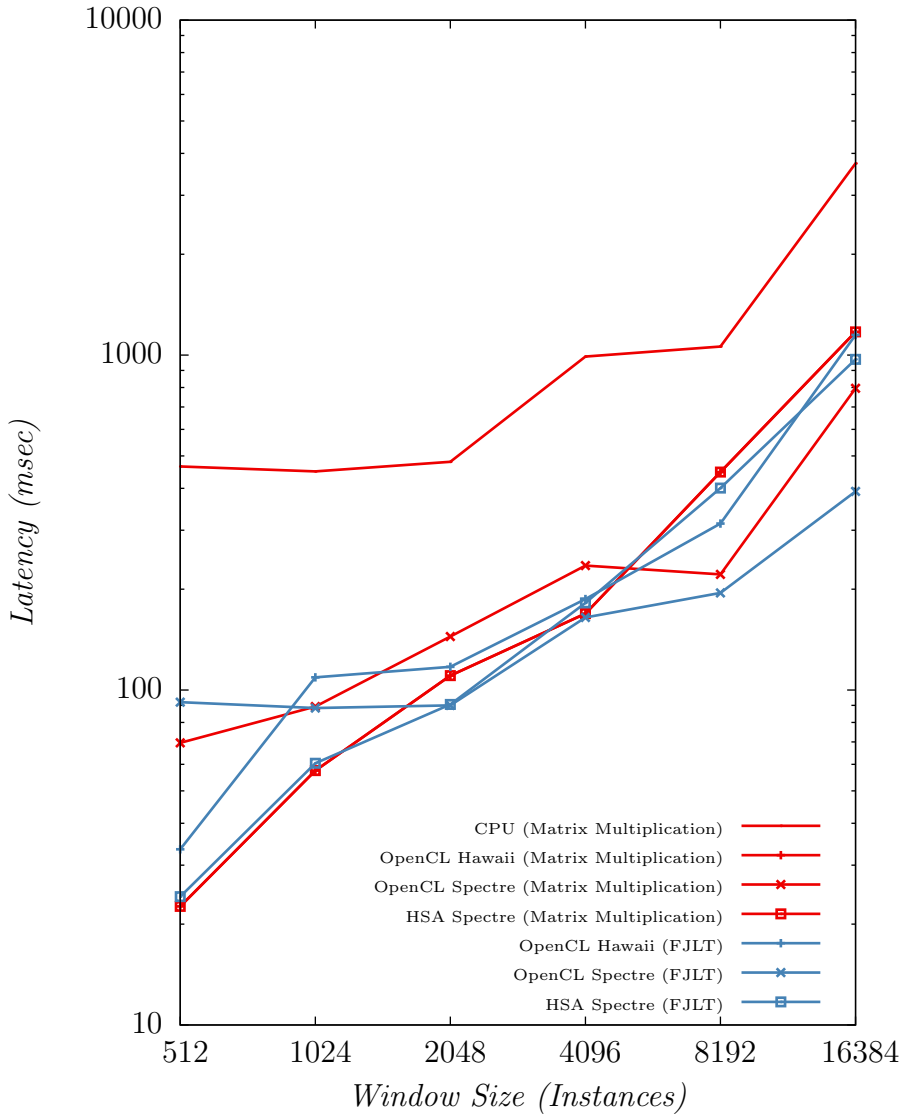


Figure 6.11: Random Projection Tree training latency on Infinimnist dataset

card over Spectre integrated GPU using OpenCL drivers. On average integrated GPU shows 1.6x times more power efficient performance. High power to speedup ratio for SGD test can be explained by the shared memory update bottleneck.

Figure 6.12: Random Projection Tree training speedup on Infinimnist dataset

	Spectre		Hawaii	
Algorithm	Load	Power Draw	Load	Power Draw
exhaustive search	50%	32W	30%	82W
k-d tree train	38%	24W	27%	74W
r-p tree train	38%	24W	27%	74W
z-order mapping	75%	48W	40%	110W
sgd(single worker)	65%	42W	34%	93W

Table 6.2: GPU Load/Power Draw (roughly estimated as percent of Thermal Design Power), Infinimniset dataset

Algorithm	Speedup Ratio	Power Ratio	Power to Speedup
exhaustive search	1.7	2.5	1.5
k-d tree train	1.81	3.0	1.7
r-p tree train	2	3.0	1.5
z-order mapping	1.37	2.2	1.6
sgd(single worker)	1.01	2.2	2.3

Table 6.3: Hawaii/Spectre power to speedup comparison, Infinimnist dataset

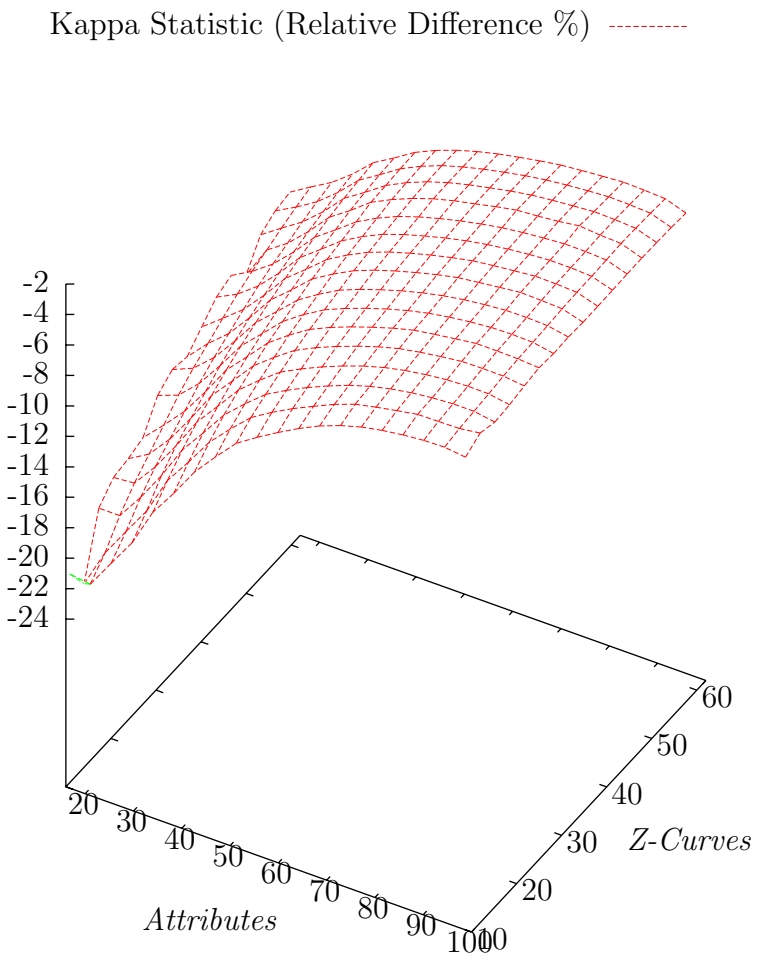


Figure 6.13: RandomRBFGenerator Z-Order compared to Exhaustive Search Kappa Statistic (Relative Difference %)

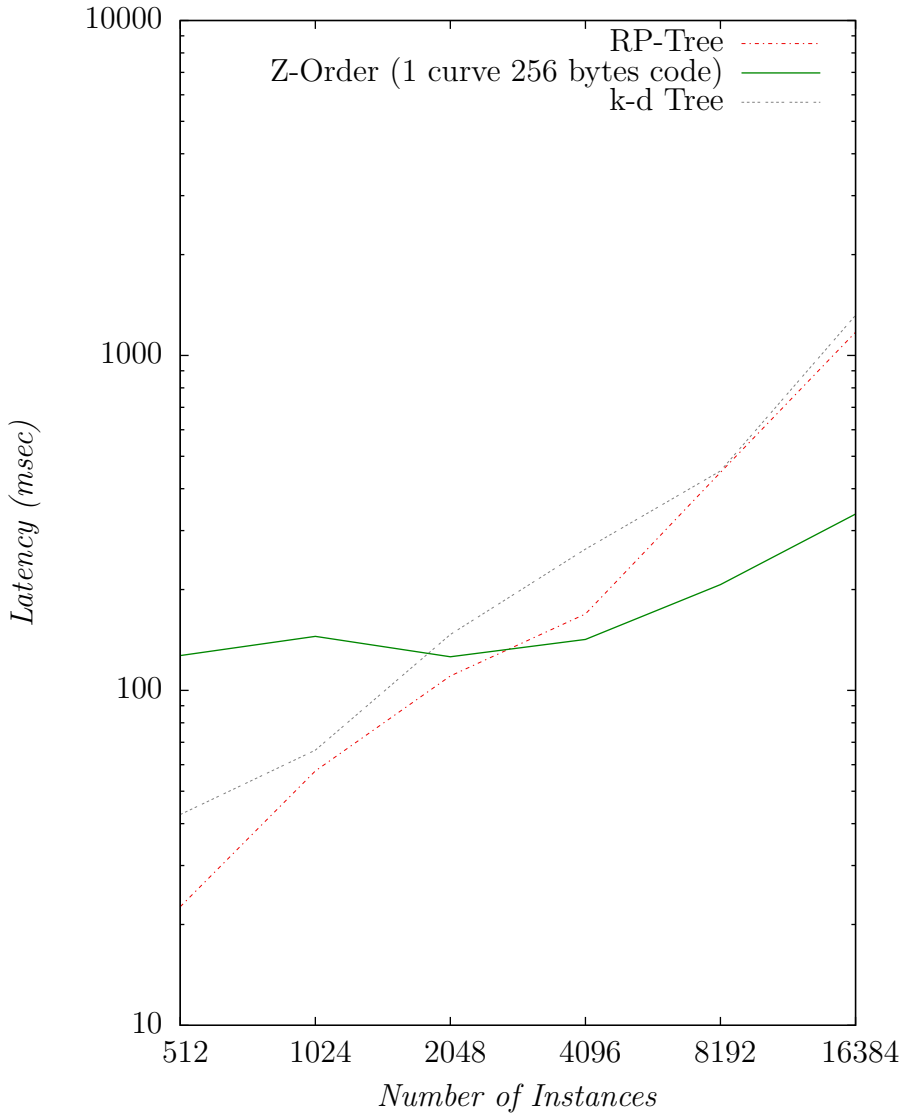


Figure 6.14: Training Times for k-Nearest Neighbours methods (HSA)

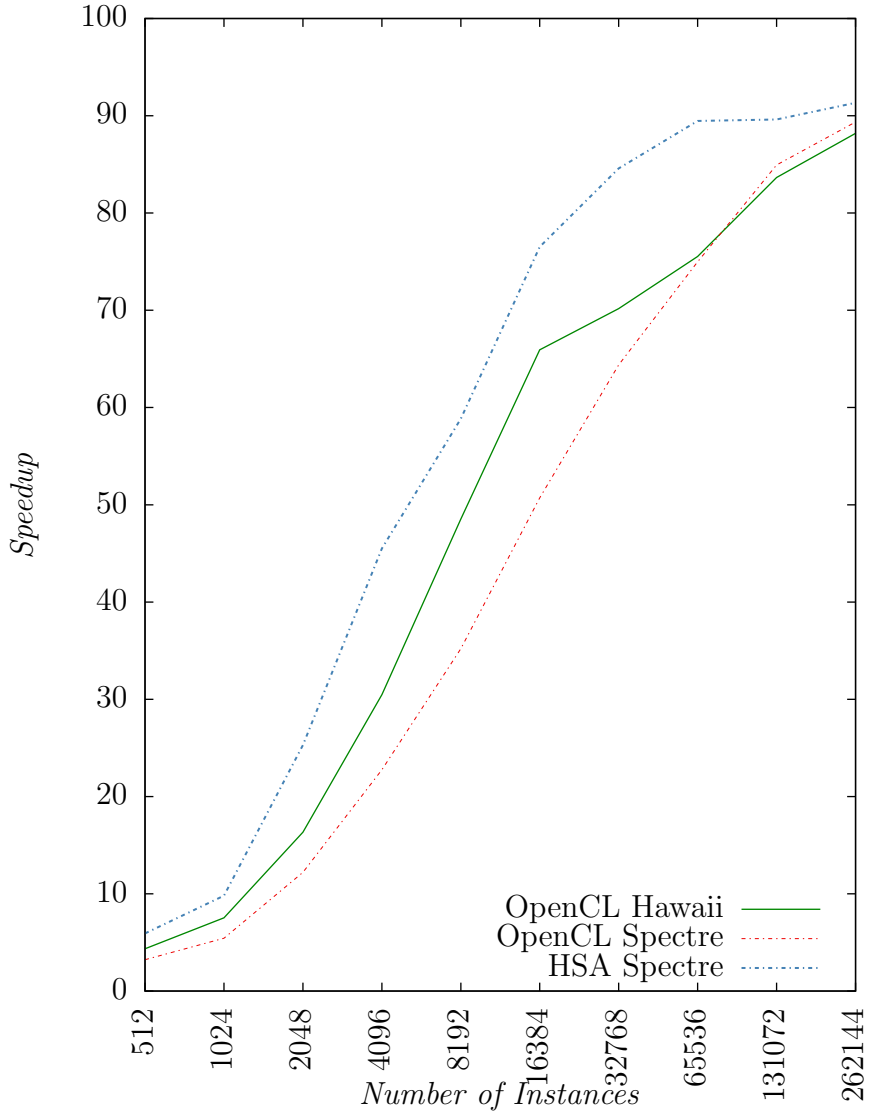


Figure 6.15: Z-Order mapping speedup (256 byte code)

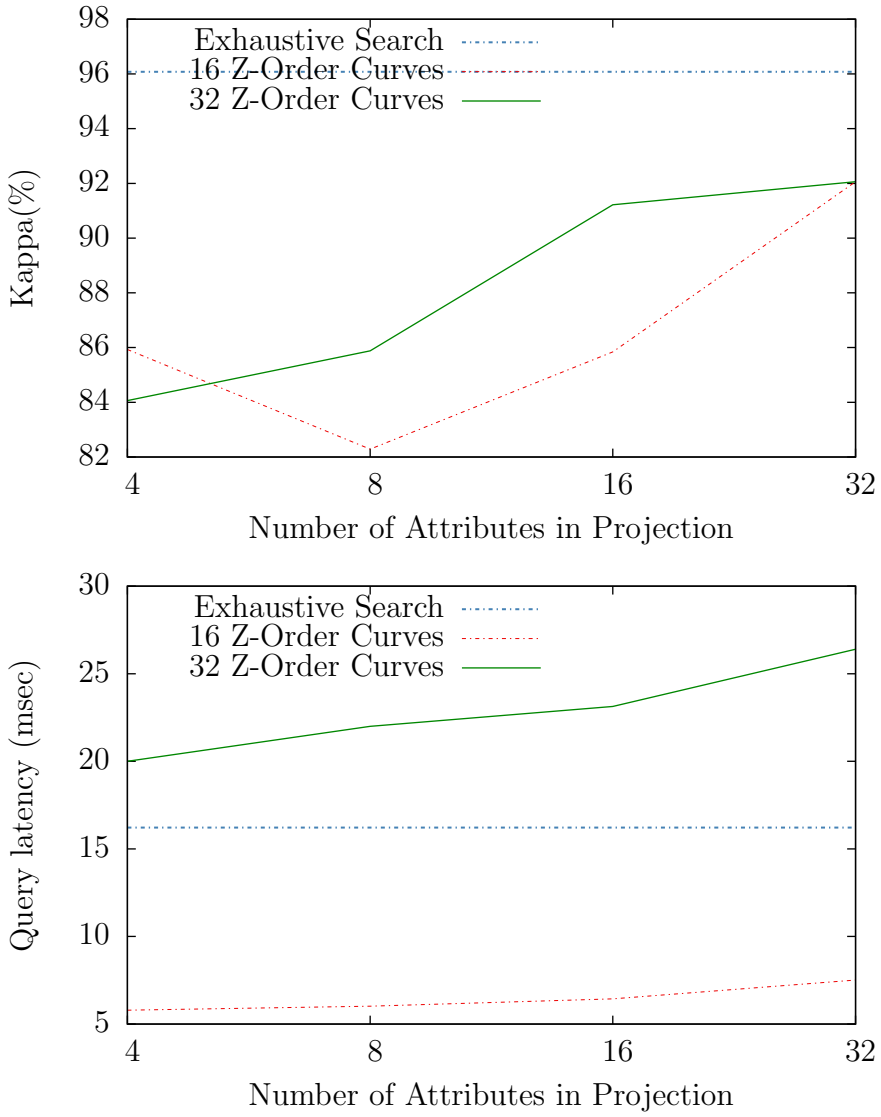


Figure 6.16: Z-Order search compared to Exhaustive Search (OpenCL Spectre)

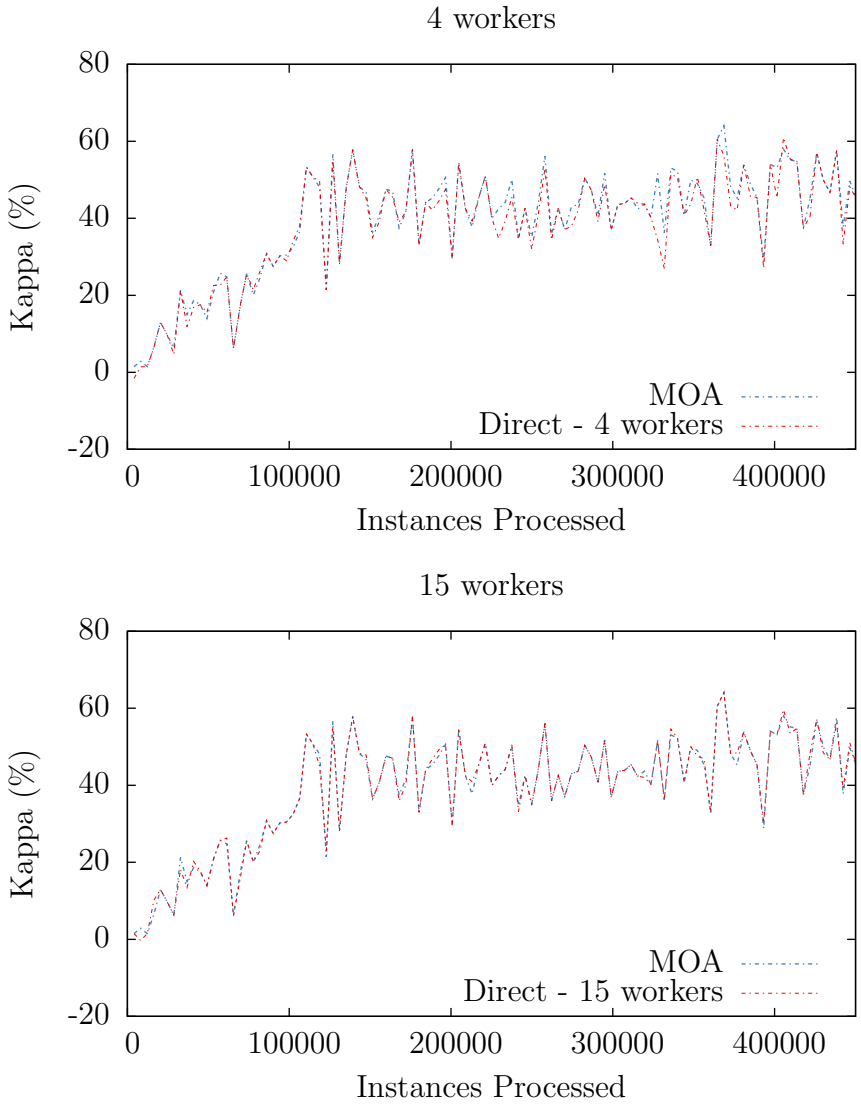


Figure 6.17: Direct update number of workers/Kappa statistic

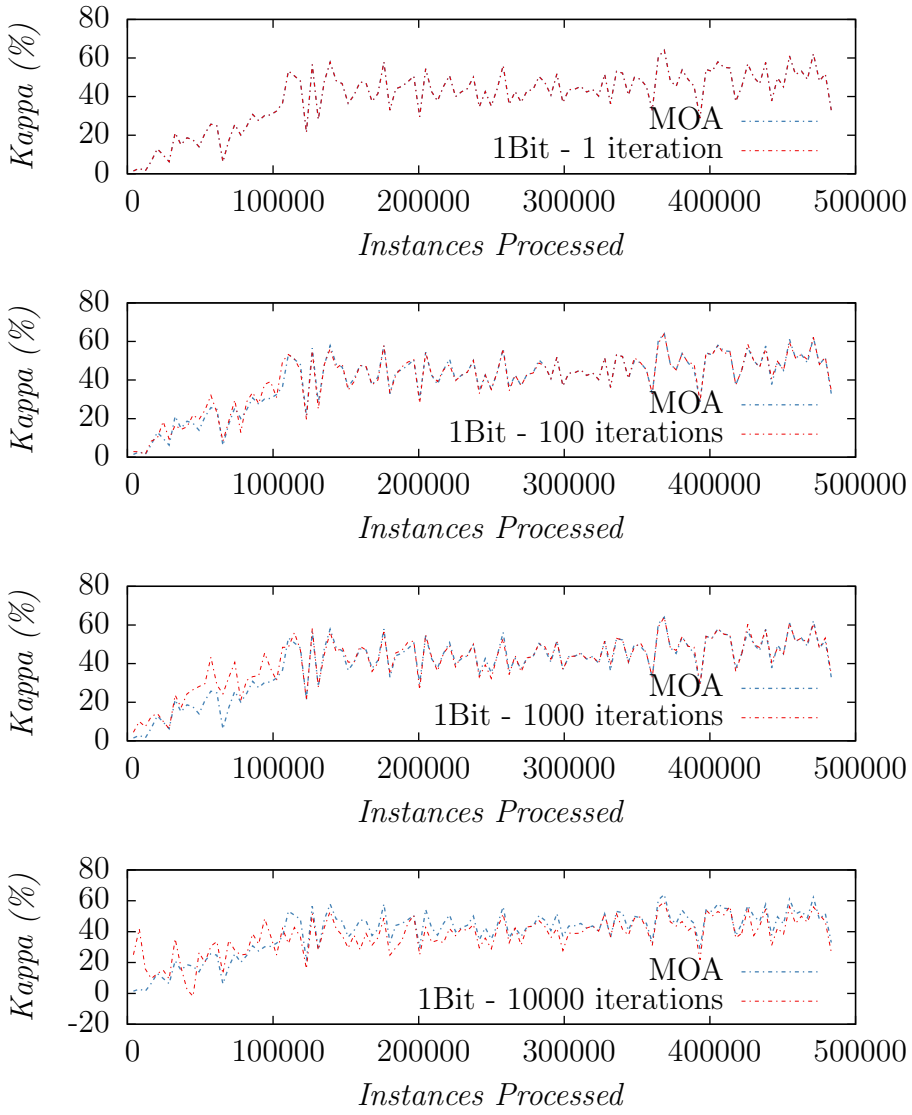


Figure 6.18: 1-Bit SGD quantization delay effect on Kappa statistic

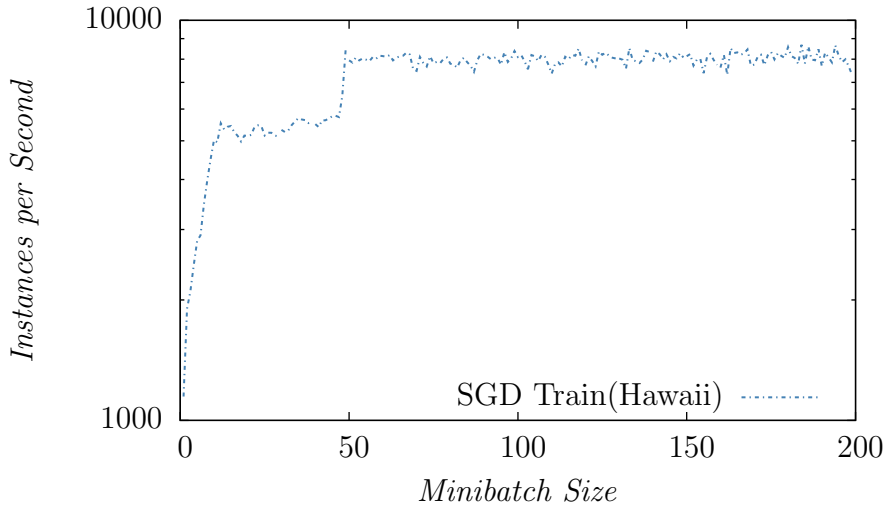


Figure 6.19: Training Speed depending on batch size (Infinimnist dataset)

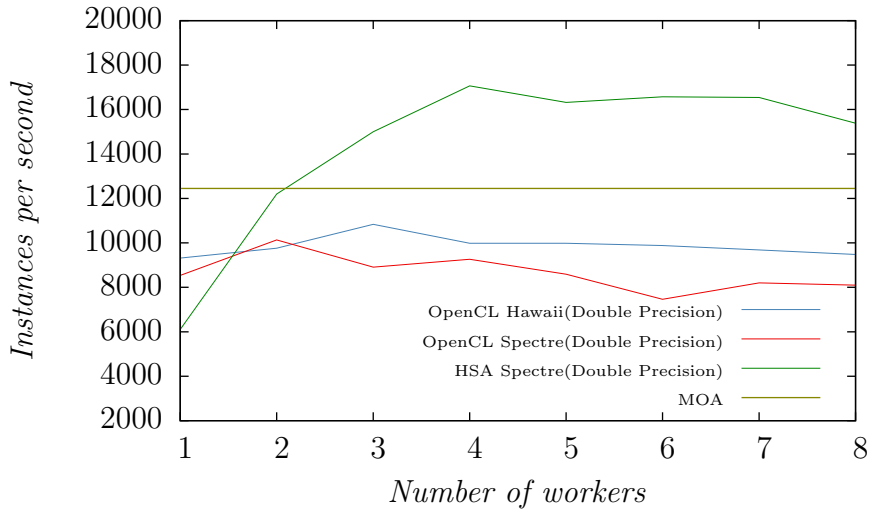


Figure 6.20: Training Speed depending on number of workers (Infinimnist dataset)

Chapter 7

Conclusions and Future Work

We have provided implementations of k-nearest neighbours algorithms and stochastic gradient descent for OpenCL platform on Windows 8 with Catalyst drivers and Linux HSA platform. We have compared their respective performance compared to the single-threaded MOA implementation.

7.1 k-Nearest Neighbours

The exhaustive search is the basic primitive used in all other kNN implementations. Figure 6.1 highlights a well-known issue in GP GPU programming — the algorithm lacks performance portability as the specific distance calculation methods show significant degradation for different work sizes. The algorithm and scheduling have to be chosen dynamically in line with the dimensionality of the task and hardware capabilities to obtain best execution speed. High-performance GP GPU computing libraries address this issue by either keeping a hardware database of parameters [74] or auto-tuning [63]. We have hardcoded the threshold values for the algorithm choice and scheduling based on the tests performed on the target platform, but it is desirable to add an auto-tuning procedure to the software.

The single float precision is sufficient for k-nearest neighbours classification on chosen datasets. The further improvement should concentrate on data quantization to investigate the low-precision representation of data as it can

significantly reduce required memory bandwidth.

This work did not evaluate sparse representation of the data, but a test of sparse versus dense matrix multiplication has shown that the break-even point for the sparse matrix multiplication is 1% of the non-null values present in the matrix. Matrix multiplication of CSR matrices in ViennaCL with the higher fill ratio will be outperformed by dense representation.

k-d tree evaluation has not shown any significant speed-up and in fact worse than exhaustive search due to the additional time required for the tree traversal and instances not being rearranged for the sequential access. Random Projection tree shows better performance due to the GPU buffers rearranged for sequential instance access within the tree node and tree structure adjusted to the manifolds of the data that minimizes the number of node traversals compared to the *k-d* tree.

k-d tree training has shown a significant speedup compared to the serial implementation due to the parallelisation of the min-max computation as MOA implementation computes min-max over a full sliding window for each level of the tree. Random Projection tree performs parallelisable operation (projection) only once per instance in the sliding window resulting in lower overall speedup.

Z-Order search is a family of approximate methods that shares similarities with locally sensitive hashing and tree search. The data locality preserving regions of the z-order curve can be considered as tree node leaves and they are commonly used as input for the tree generation. The projection on the z-order curve can also be viewed as a hash code calculation and data locality preserving the property of space filling curves allows thinking that locality sensitive condition holds. The algorithm provides flexibility to the user allowing to compromise for quality or runtime. It is also one of the most GPU friendly algorithms resulting in highest utilization. The bottleneck of the current implementation is the computation of the intersection set of the data points found in different curves as it is performed on CPU. The more efficient version might

use atomics to flag the instances that should be considered for the short list and then either apply scan and scatter primitive to compose a list of instances for the one thread per instance distance function or abort computation when the flag is set for one workgroup per instance kernel.

7.2 Stochastic Gradient Descent

Stochastic Gradient descent was implemented with double floating point precision. A number of works suggest that double precision is unnecessary [33] and single floating point or even 16-bit fixed point is sufficient for common applications of SGD such as neural network training [53][39][76]. The main bottleneck of the implementation was the access to the shared memory segment to update model weights. The future implementation could benefit from lower precision calculations and weights cached in the GPU memory.

There was no difference in performance between 1-bit and direct updaters due to the staggered weight updates as observed by debug timestamps of the updates in each worker. The HSA implementation that directly updated shared memory from within the GPU kernel has shown the same behaviour. The algorithm might behave differently on discrete GPU if the weight update would be parallelised inside GPU memory by using multiple queues within the same OpenCL context.

7.3 Discrete and Integrated GPU Comparison

This works compares performance of R9 390 Hawaii discrete GPU and AMD A8-7600 Spectre integrated GPU. The collected benchmarks show that despite discrete GPU begin 6 times more powerful core-wise (384 vs 2560)and with 8 times more memory bandwidth (128-bit vs 512-bit bus) it does not directly translate into an algorithm speedup. Best results for the discrete GPU are obtained when an optimal batch of data is processed with little or no synchronization with the CPU. Example would be the exhaustive search with best

speedup for Spectre 10x with 512 instances window size and 70x for Hawaii with 8192 instances window. The CPU-GPU communication-intensive task such as k-d tree evaluation with work sizes underutilization both GPUs results in single-digit best (1.5 Hawaii and 1.3 OpenCL Spectre) speedups. At small worksizes (up to 8192) the performance on the benchmarked tasks of discrete GPU and integrated one are comparable due to a number of reasons — discrete GPU underutilization and higher scheduling and data transfer overhead per instance processed. At certain tasks and work sizes the integrated GPU provides even better speedups, e.g. 2x over discrete GPU in 1024 window size k-d tree training benchmark or on par with the discrete GPU, e.g. z-order curve mapping. It should be noted that integrated GPU consumes considerable less power as shown in Tables 6.2, 6.3. Thus, the integrated GPU provide a power-efficient alternative to the deployment of discrete GPU for small batch sizes common for the data stream processing.

7.4 Heterogeneous System Architecture

The HSA platform provides means for the efficient offload of parallel computation to the GPU device where tight coupling with the CPU is required. Unfortunately, the current driver implementation underperforms compared to traditional OpenCL drivers on exhaustive search and k-nearest neighbours benchmarks but shows positive speedup in a shared memory parallel SGD implementation. Current ViennaCL-based implementation followed OpenCL execution model and benchmarking was limited to direct comparison of memory access and scheduling performance of HSA and OpenCL drivers. The features of the platform such as possibility to enqueue CPU computation from the GPU kernel should be further investigated as they allow for the possibility of seamless lock-free data stream processing.

7.5 Conclusions

The integrated GPUs are suitable for the latency constrained data stream processing tasks and provide a better performance return on deployment and operating costs due to the lower power consumption and presence in most modern embedded, desktop and server CPUs (e.g. Intel Xeon E3). While discrete GPU shows better performance on larger work sizes, integrated GPUs have better performance-to-power ratio and perform on-par with discrete GPUs for small work sizes.

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] Cntk - computational network toolkit. http://www.cntk.ai/, 2016.

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

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

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

[26] 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.

[27] 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.

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

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

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

[31] 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.

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

[33] Olivier Bousquet and Léon Bottou. The tradeoffs of large scale learning. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 161–168. Curran Associates, Inc., 2008.

[34] 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.

[35] 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.

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

[37] 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.

[38] Edward Ching, Norbert Egi, Masood Mortazavi, Vincent Cheung, and Guangyu Shi. Unleashing the hidden power of integrated-gpus for database co-processing. In *GI-Jahrestagung*, pages 1755–1766, 2014.

[39] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014.

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

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] Ivan Grasso, Petar Radojkovic, Nikola Rajovic, Isaac Gelado, and Adrian Ramirez. Energy efficient hpc on embedded socs: Optimization techniques for mali gpu. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 123–132. IEEE, 2014.

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

[54] 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.

[55] Jonatan Jansson. Integrated gpus: how useful are they in hpc? 2013.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

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

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

[62] N Marz. Apache storm, 2014.

[63] 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.

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

[65] 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`.

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

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

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

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

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

[71] 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.

[72] 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.

[73] 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.

[74] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In *Intl. Workshop on GPUs and Scientific Applications*, pages 51–56, 2010.

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

[76] 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`.

[77] 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.

[78] 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.

[79] James G. Shanahan and Laing Dai. Large scale distributed data science using apache spark. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 2323–2324, New York, NY, USA, 2015. ACM.

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

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

[82] 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.

[83] 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`.

[84] 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.

[85] 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.

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

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

[88] 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.

[89] 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.

[90] 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.

[91] 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.