Distributed/Cluster Computing for Data Stream Mining: Draft Notes

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

Vladimir Petko

University of Waikato
2015

Abstract

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

Acknowledgements

Contents

List of Figures

List of Tables

Introduction

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

Chapter 1

General Purpose GPU Computing

Introduction

The modern Graphical Processing Units (GPU) greatly outpace CPUs in arithmetic throughput and memory bandwidth for data-parallel tasks. Since 2003 the efforts were made to port data parallel algorithms to GPUs - first using shader languages such as HLSL, then with the release of Nvidia G80 in 2006 using extensions to C programming language - CUDA[7]. Presently there is a number of programming frameworks targetting specifically GPU architecture such as CUDA[24], OpenCL[**?**], RenderScript[10], DirectCompute[**?**] and more generic parallel-processing frameworks such as OpenMP[**?**] and AMP[2] which provide GPU backend as one of the targets. The differences in the hardware architecture between CPU and GPU is reflected in the programming model of the traditional GPU-specific languagages which contain hardware architecture specific language constructs. This chapter provides an overview of GPU architecture, most known programming frameworks, lists limitations of the traditional GP GPU programming,discusses OpenCL 2.0 standard which addresses some of the limitations and Heterogenous System Architecture (HSA) an optimized platform architecture for OpenCL 2.0.

GPU Architecture

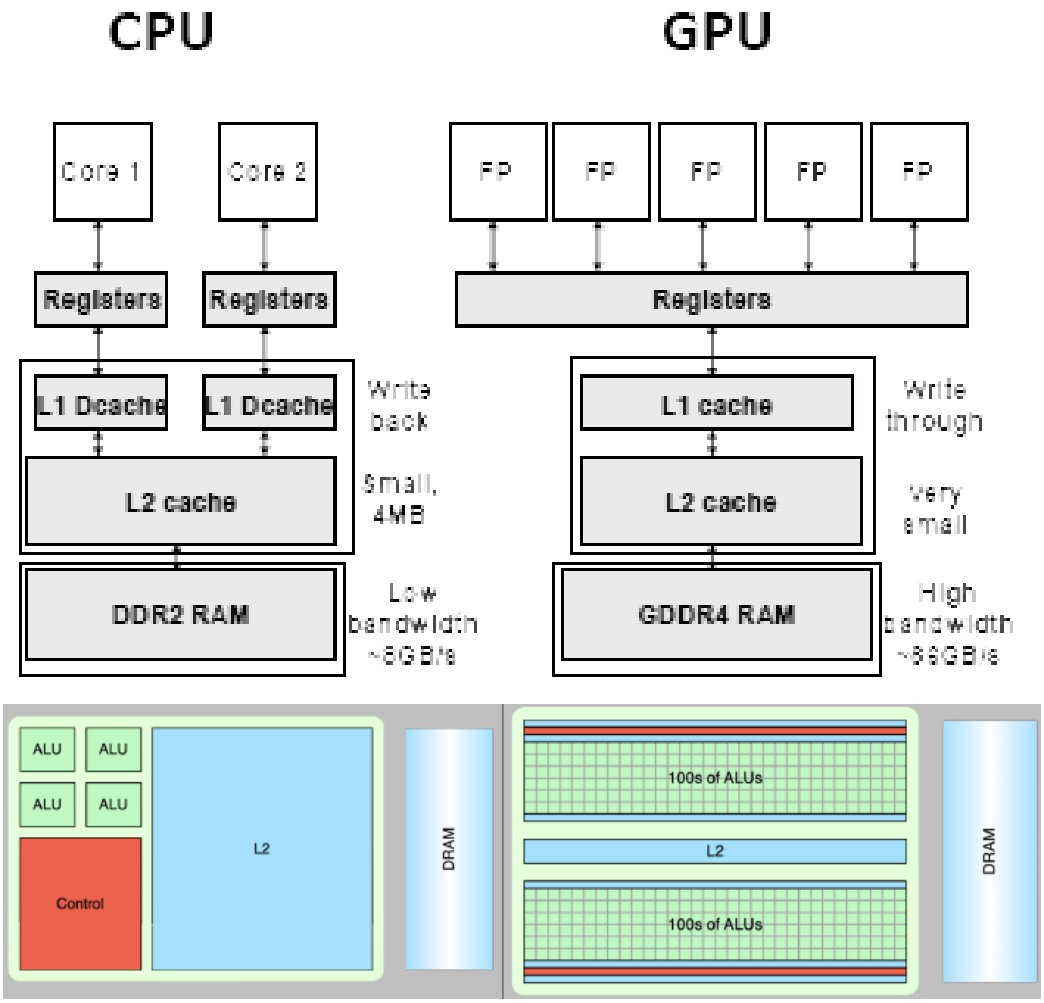

Figure 1.1: CPU versus GPU hardware architecture. Reproduced from NVIDIA GPU ARCHITECTURE and CUDA PROGRAMMING ENVIRONMENT by Alan Tatourian[32]

The main differences between modern CPU and GPU architectures are the level of paralellism and ability to directly address tiered memory. Modern CPU with 2 hex-cores support maximum of 12 threads (24 with hyperthreading) and minimal unit of execution for the NVIDIA GPU (called *wavefront*) is 32 threads. Modern GPUs implement SIMT (Single Instruction - Multple Thread) execution model (AMD/NVIDIA desktop GPUs) first introduced by NVIDIA in G80 model[7]. 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[11][24]. The CPU thread is a heavy-weight entity which is centered around execution of a specific task for an extended period of time. Whenever CPU needs to preempt the thread, the register state is stored and another thread takes over. This makes a context switch a costly operation and operating systems attempt to minimize number of context switches per second. The GPU context switch is an extremely lightweight operation and is routinely used for the latency-hiding - whenever the wavefront is waiting on data, the GPU schedules another wavefront for execution. The GPU registers are private for each thread and are not reallocated until thread execution completes.

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

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

The GPU programming following abstractions:

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

The unit of scheduling is called *wavefront* in AMD terminology or *warp* in NVIDIA and typically consists of 32 threads on NVIDIA and 64 on AMD hardware. The GPU chip is equipped with a number of SIMT cores which execute same instruction for each *warp*. Divergence of control results in underload of the processing units and reduces performance. The branching should be reduced to wavefront granularity to avoid wasting execution cycles[29][24] It should be noted that the wavefront size is a hardware specific feature and the optimization should be performed at the run-time.

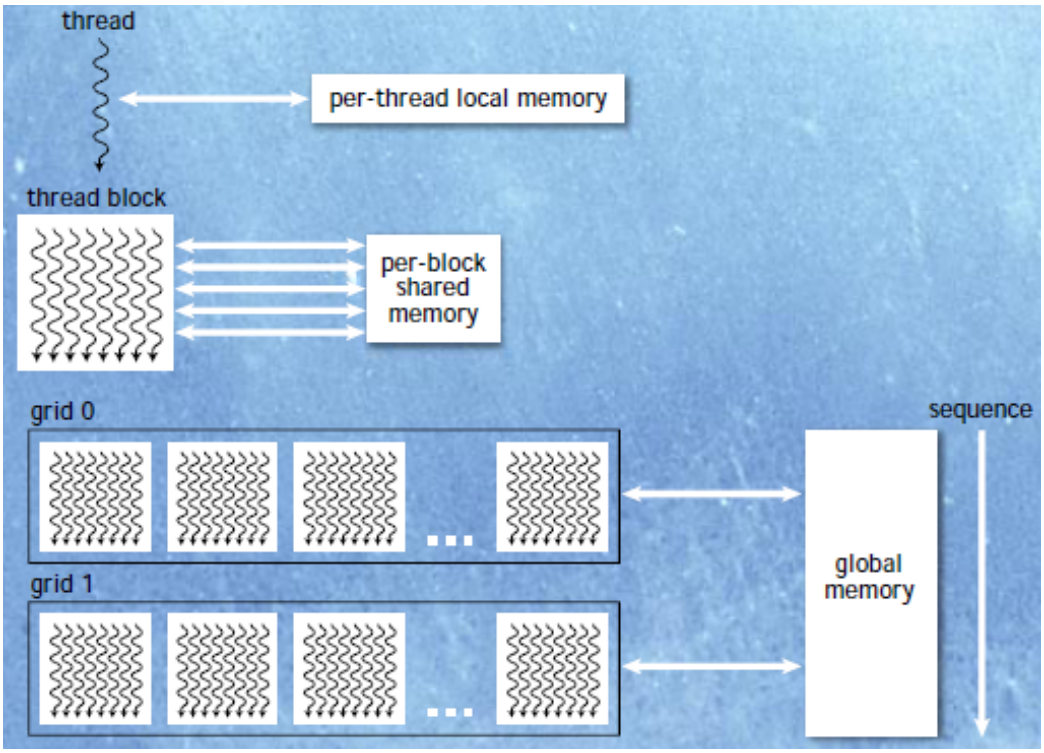

Figure 1.2: GPU Memory Tiers. Reproduced from NVIDIA GPU ARCHITECTURE and CUDA PROGRAMMING ENVIRONMENT by Alan Tatourian[32]

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[8], traditional GPU languages such as CUDA[24] expose task scheduling and memory management giving the expert user fine-tuning capabilities, low level languages provide intermediate binary format compatible with multiple hardware targets. The tree of the GP-GPU technologies is presented in the Figure 1.

High Level Languages

OpenACC and OpenMP are high level parallel programming frameworks that specify a set of annotations, environment variables and library routines for shared memory parallelism in C/C++ and Fortran programs[**?**][**?**]. Microsoft

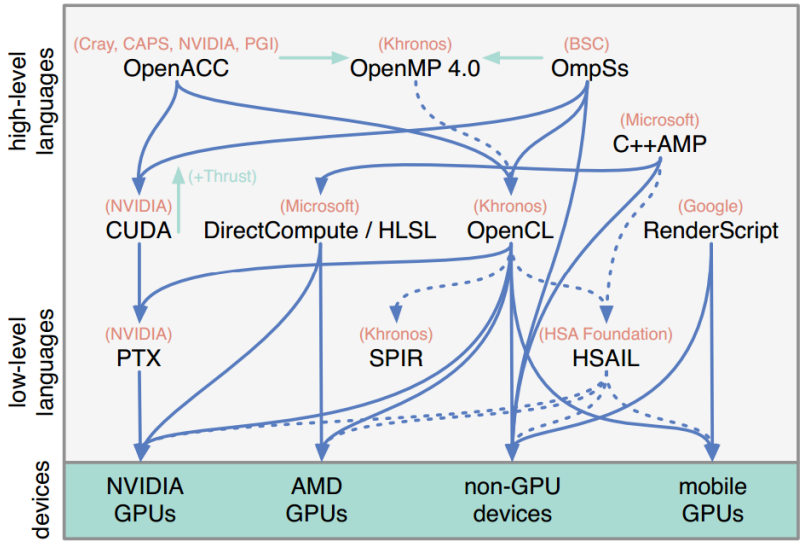

Figure 1.3: GP GPU technologies tree. Reproduced from C. Nugteren, Improving the Programmability of GPU Architecture, p. 21 [26]

C++ AMP[2] is a C++ library which enables parallel computations for CPU and GPUs (using Microsoft DirectX Shading Language) Rootbear GPU compiler provides a transparent compilation of Java code into CUDA[27]. Aparapi provides a way to generate OpenCL kernel code from Java, theoretically allowing code which can be executed on CPU and offloaded to GPU if needed[1]. Project Sumatra is a OpenJDK project which focuses on development of the Hotspot virtual machine capable of offloading JDK 8 Stream API[5] computations to the GPU[8].

GPU-specific Languages

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

- CUDA - A programming language for NVIDIA hardware based on C language. Kernels are expressed as C-functions for one thread with parallelism defined at run-time by specifying dimensions of execution grid and thread blocks[24]

- OpenCL 1.X builds upon ideas implemented in CUDA by adding de-

vice management APIs and providing hardware-agnostic programming specification. OpenCL gives *write once-run anywhere* guarantee but does not give any performance consistency guarantees across different hardware[31].

- RenderScript - Android GPU computing component which uses OpenCL with Java binding programming model - C-style kernels and Java-based control code. RenderScript does not provide any APIs for the *work-group* size control in the bid to provide performance portability between different devices[10].

- DirectCompute/HLSL - Microsoft Parallel Computing.

Low-Level Languages

The low level assembly representation is used to abstract compiler implementation from the actual hardware since each model or even revision may have a different instruction set. The translation is performed by *Just-In-Time* compiler before the kernel execution. Each vendor provides different low level specifications: NVIDIA CUDA uses Parallel Thread Execution and Instruction Set Architecture (PTX ISA)[9], Khronos Group specifies Standard Portable Intermediate Representation(SPIR)[**?**], and HSA Foundation specifies Heterogenous System Architecture Intermediate Language (HSAIL)[**?**].

Limitations

Input Size The massively parallel nature of GPU platforms require a certain amount of data to be passed to the kernel to achieve maximum performance. Table 1.1 shows execution time of a kernel which assigns index to each array element $X_i = i$ on AMD A8-7600. The execution time starts to increase when input size is above 1024 and remains constant for lower values. To maximum performance on AMD A8-7600 will be achieved when input size will exceed 1024 elements.

Global Size	256	512	768	1024	1280	2560	3072	3584	4608	4864
Execution Time (μsec)	8	8	8	8	9	9	10	11	11	12

Table 1.1: Input Size and Execution Time

GPU Memory Size and Host-GPU Transfer The discrete GPU requires transfer of data from the host to the GPU memory which adds additional overhead to the computations and requires task partitioning according to the memory specification of GPU[28]. Memory transfer is a bottleneck for Aparapi and its developers allow explicit memory management[1]. This effectively reduces framework which promises CPU-GPU interoperability to the Java wrapper of the OpenCL API.

Kernel Launch There is a constant time needed to setup kernel launch which might offset any gain from parallelization if the data can be processed sequentially faster.(NB. Amdahl's law) It is impossible to schedule kernel execution from within the kernel itself requiring a mix of kernel and host code if several iterations are required.

OpenCL 2.0

OpenCL 2.0 standard[6] introduces several features which attempt to address limitations of GPU programming:

- Shared Virtual Memory - both host and kernel code share same address space thus either hiding memory transfers (discreet GPU driver stack) or if backed by the hardware architecture such as HSA eliminate its need[12]
- Dynamic Parallelism - OpenCL 2.0 allows scheduling of kernels from within a kernel without host interaction reducing host CPU bottleneck.
- Pipes - pipes feature allows passing data from kernel to kernel without processing the whole input which allows to obtain the results of computation faster.

HSA Platform

AMD introduced Heterogeneous System Architecture plaftorm as an optimized platform architecture for OpenCL 2.0. Its specifiction introduces a set of requirements that allow both GPUs and CPU share same memory space, synchronize execution using signals and atomics and schedule execution both from GPU and CPU[12]. Task execution is performed by *agents* which represent CPU or GPU nodes. The task execution is scheduled via *queues* and synchronized using *signals*. HSA memory model guarantees sequential consistency for the correctly synchronized programs.

Software Available: At the moment (Feb 2014) there is a OpenCL 2.0-¿HSAIL compiler available[4] and a Linux-based runtime environment[3].

HSA Queues

HSA uses queues to schedule code execution. A HSA *queue* is a ringbuffer which contains *packets* with either call or synchronization parameters. The queue maintains two indexes - read index and write index. Write index is modified by the user and used to submit packets to the queue. The read index is updated by the packet processor whenever the packet is taken for execution. As soon as packet is written to the queue the ownership is taken by the HSA packet processor and it may change packet contents at any time[12]. Compared to traditional dispatch where the execution is scheduled via user-mode and kernel-mode driver layers the HSA dispatch intends to be lightweight and source-agnostic way of scheduling execution. The HSA Queues support work-stealing that is several HSA agents may be attached to the queue to share the workload.

HSA Signals

HSA uses *signals* to perform synchronization between host and kernels being executed or to signal completion of the task. A *signal* is essentially a shared

memory variable modified by the HSA agent. Runtime environment provides a way to check the value of the signal or wait for the specific value.

HSA Memory Model

The sequential consistency was first defined by L. Lamport as " ..the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." Modern processors (ARM, x86, Itanium, POWER) introduce relaxed memory model to allow a range of the hardware optimizations to provide better performance by reodering load and store operations[22]. Platform specification states

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

[12] Similiar to Java Memory Model[**?**] it guarantees sequential consistency for the correctly synchronized programs, that is 'synchronizing operations meet the requirements for sequential consistency within each scope/segment instance'[12]. The specification introduces several memory segments: "

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

" Each particular memory location is always associated with one and only one segment and all operations apply to only one segment with the exception of fence operations[12]. In addition to memory segments HSA memory model

introduces *scopes* : wavefront, work-group, component and system. They can be used to reduce visibility of the memory operation compared to the default supported by the segment.The global segment may use any of the specified scopes and group segment is limited to wavefront and workgroup scopes[12]. Different workgroups accessing a global variable with the workgroup scope will work with different instances of the variable. The write serialization only applies to the operations within the segment/scope that they specify.

Implementation Notes

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

Conclusion

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

Chapter 2

k-Nearest Neighbours

Problem Statement

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

- Forward k-NN - for an given sliding window of examples find classification of the variable query
- Reverse k-NN - for a given fixed query form a window of examples nearest to it. This approach is discussed in Efficiently Processing Continuous k-NN Queries on Data Streams[15].

There is a number of implementations of the offline k-NN algorithm for GP-GPU: brute force approach[16][20][19][28], kd-tree[18][33], approximate [23][21]. The brute force implementation consists of distance calculation and sorting phase. The distances to the query a computed as a vector-matrix multiplication or if several queries are processed at once as a matrix-matrix multiplication. GPU implementation of those routines is available as a part of cuBLAS library[**?**]. Sorting phase finds k nearest of all the computed distances[28]. Sismanis et. al[28] provide time complexity of reduced sort algorithms and evaluates their performance on GPU, proposes to interleave distance calculation and sorting phases to hide latency - the data for the distance calculation should be offloaded to GPU while it performs the sorting phase. The input data in the brute-force approach is partitioned according to the GPU memory capabilities and does not use examples's spatial information. The kd-tree approach presented by Gieske et. al focuses on parallel execution of nearest neighbour queries in a lazy fashion. The query points are accumulated in the leaf nodes of the kd-tree until enough of them is present and then processed as batch. This solves an issue of the GPU underutilization and low performance if leaf nodes are processed sequentially for each example[18]. The parallel kd-tree construction is explored by Zhou et. al[33].

TODO: Continue

Algorithm Implementations

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

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

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

```
1  // input – query vector
2  // samples – sliding window, matrix of window_size instances
3  // ranges – min/max values for each attribute
4  // result – resulting distance vector
5  // window_size – size of the window
6  // element_count – number of attributes in each instance
7  // numerics_size – number of numeric attributes
8  distance(double* input, double* samples,      _double2* ranges, double* result, int
       window_size, int element_count, int numerics_size)
9  {
10   forall result_offset ( 0 < result_offset < window_size) do in parallel
11     int vector_offset = element_count * result_offset;
12   double point_distance = 0;
13   double val;
14   double width;
15   int i;
16   for (i = 0; i < numerics_size ; i ++ )
17   {
18     double2 range = ranges[i];
19     width = ( range.y – range.x);
20     val = width > 0 ? (input[i] – range.x) / width – (samples[vector_offset + i] – range.
           x)/width : 0;
21     point_distance += val*val;
22   }
23
24   for (; i < element_count; i ++ )
25   {
26     point_distance += isnotequal( input[i] , samples[vector_offset + i]);
27   }
28   result[result_offset] = point_distance;
29 }
```

The optimal implementation depends on the size of the window and number of attributes present[30]. For the small instance size (¡100) and windows less than 10^4 elements naive implementation will provide the best solution. Best all around distance calculation should apply different strategies depending on the windowsize and number of attributes.[30].

Selection Alabi, et.al evaluated different selection strategies based on bucket sort algorithm and Merril-Grimshaw implementation of radix sort[13].

This implementation provides only select based bitonic sort[14] algorithm which is suboptimal as we can only truncate sorting at the last stage of the algorithm.

*TODO:*The work needs to provide several alternative selection strategies. While Merril's algorithm may be too complex for implementation, we might use k-bucket Selection[13] to provide alternative selection strategy.

KD-Tree based k-Nearest Neighbours Search The KD-Tree structure was implemented as a sequentially updated structure. The distance calculation for the leaves of KD-Tree was offloaded to GPU. The OpenCL implementation required transfer of the leaf nodes contents to the GPU memory and has shown performance worse than serial implementation due to the transfer overhead *Need table*

Some ideas to try: when updating tree do not remove instances - each instance has a timestamp and distance calcuation assigns max distance when instance is too old. Old instances are replaced by the new ones (see FIFO buffer in brute-force implementation). The NN-Search can be batched - to reduce kernel launch overhead accumulate N query instances at the leaf node and only then perform distance calcuation.[17]

LHS based k-Nearest Neighbours Search *TODO*

Chapter 3

Stochastic Gradient Descent

Problem Statement

TODO

Algorithm Implementation

The implementation relies on the fact that the weights vector can be updated without locking since the training instances are sparse and each instance contributes to a different part of the weights vector[25].

Chapter 4

Experimental Results

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

k-Nearest Neihbours

TODO

Stochastic Gradient Descent

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

Chapter 5

Conclusions and Future Work

TODO

References

[1] aparapi - api for data parallel java. allows suitable code to be executed on gpu via opencl.

[2] C++ amp overview.

[3] Hsafoundation.

[4] Hsafoundation/cloc.

[5] java.util.stream (java platform se 8).

[6] Khronos opencl registry.

[7] Nvidias next generation cuda(tm) compute architecture:fermi.

[8] Openjdk: Project sumatra.

[9] Ptx isa :: Cuda toolkit documentation.

[10] Renderscript—android developers.

[11] Single instruction, multiple threads.

[12] Hsa platform system architecture specification, 2014.

[13] 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.

[14] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[15] C. Bohm, Beng Chin Ooi, C. Plant, and Ying Yan. Efficiently processing continuous k-nn queries on data streams. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 156–165, April 2007.

[16] A. Dashti. Efficient computation of k-nearest neighbor graphs for large high-dimensional data sets on gpu clusters.

[17] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using GPU. *CoRR*, abs/0804.1448, 2008.

[18] 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.

[19] Kimikazu Kato and Tikara Hosino. Solving k-nearest neighbor problem on multiple graphics processors. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 769–773, Washington, DC, USA, 2010. IEEE Computer Society.

[20] Quansheng Kuang and Lei Zhao. L.: A practical gpu based knn algorithm. In *In Proceedings of the Second Symposium on International Computer Science and Computational Technology (ISCSCT 09) (Dec. 2009), Academy Publisher*, pages 151–155.

[21] Tieu Lin Loi, Jae-Pil Heo, Junghwan Lee, and Sung-Eui Yoon. Vlsh: Voronoi-based locality sensitive hashing. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 5345–5352, Nov 2013.

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

[23] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.

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

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

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

[27] 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.

[28] 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.

[29] T. Aila S.Laine, T. Karras. Megakernels considered harmful: Wavefront path tracing on gpus.

[30] 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.

[31] 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.

[32] Alan Tatourian. Nvidia gpu architecture and cuda programming environment.

[33] 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.