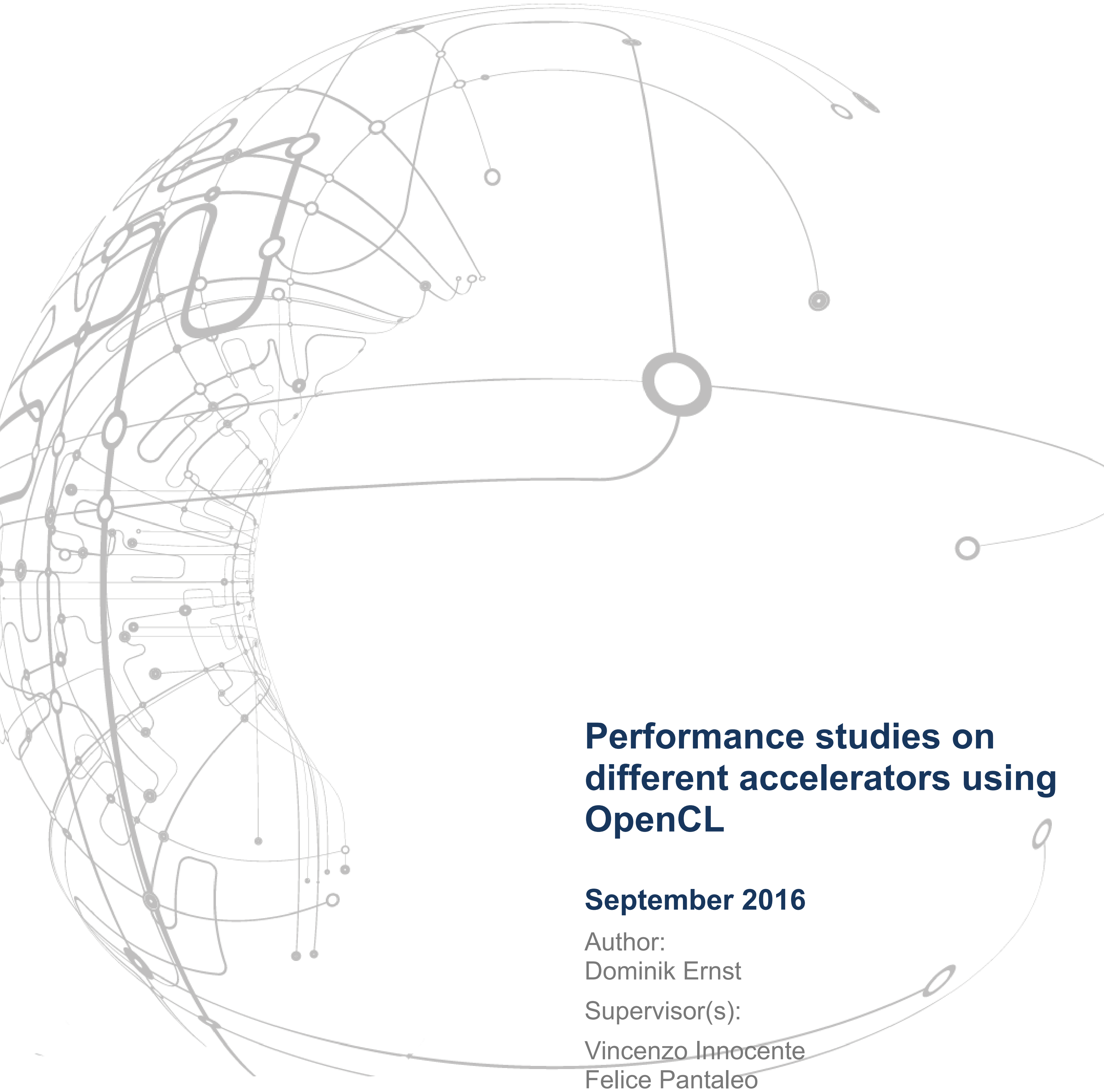

Performance studies on different accelerators using OpenCL

September 2016

Author:
Dominik Ernst

Supervisor(s):

Vincenzo Innocente
Felice Pantaleo

CERN openlab Summer Student Report 2016

Contents

Project Specification

The High Luminosity LHC (HL-LHC) is a project to increase the luminosity of the Large Hadron Collider to 5*1034 cm-2 s-1. The CMS experiment is planning a major upgrade in order to cope with an expected average number of overlapping collisions per bunch crossing of 140. The dataset sizes will increase by several orders of magnitude and so will be the request for larger computing infrastructure. The complete exploitation of a machine capability is desirable, if not a requirement, that should anticipate a request for new hardware resources to the funding agencies. Furthermore, energy consumption for computing is becoming more and more an important voice into European data center's budget. The exploitation of Intel integrated accelerators like graphics processors or FPGAs, which are and will be part of our machines, will allow us to achieve much higher energy efficiency and higher performance. Furthermore, MIMD architectures like Kalray's Massively Parallel Processor Array could prove useful as embedded solutions in real-time environments like the experiment trigger farms. All these accelerators can be programmed using OpenCL. The aim of the project is to study the performance and the power efficiency of these accelerators when executing some kernels which are part of the reconstruction of CMS experiment events using the CMS software framework.

Abstract

This report introduces the OpenCL API and programming language and describes implementations using OpenCL of several kernels used for particle track reconstruction in the CMS software framework. The first part are kernels for construction and search in the context of a k-d tree data structure. The second part is a set of kernels for building possible tracks out of pairs of hits in the silicon tracker. Several OpenCL platforms are tested and benchmarked.

1 OpenCL

OpenCL is both a programming language and an API for for the data parallel SPMD (Same Program Multiple Data) programming paradigm. Algorithms that perform the same operations for a large amount of items, are formulated as kernels, that each perform an operation on a single work item. The API is then called to instantiate the kernel for each work item. OpenCL calls each kernel instantiation a thread.

For example, the simple data parallel for loop

```
for(int i = 0; i < N; i++) {
    A[i] *= A[i];
}
```

is transformed into the kernel (note: not actual OpenCL syntax)

```
void square_kernel(float* A) {
    int i = get_thread_id();
    A[i] *= A[i];
}
```

which is the instantiated `N` times with an API call:

```
launch_kernel( square_kernel(A), N);
```

The kernel instantiations can be mapped to threads and vector lanes that can be found in GPUs, in all kind of many core architectures and in common multi core CPUs. The underlying OpenCL platform cares about launching the appropriate thread count and generating code for the right vector width for each kind of architecture, e.g. a lot of wide vector threads for GPUs, and just a few, narrower vector threads for multi core CPUs.

There is no guarantee being made as to the order of execution of threads. Synchronization and data exchange between OpenCL threads is restricted to threads that are grouped together within what OpenCL calls a work group. The maximum size for a work group is 1024 threads.

OpenCL defines a memory model with several memory spaces:

- Host Memory, accessible by normal, native code
- Global Memory, accessible only from kernels and with explicit copy functions from/to Host Memory, writes can be observed by all threads
- Local Memory, accessible only from kernels, replicated for every work group, writes can only be observed by threads in the same work group

1.1 Devices

Despite already being 3 years old, support for OpenCL 2.0 is still not widespread. OpenCL version 1.2 is therefore being targeted here. While by far not the biggest addition to OpenCL 2.0, the so called Work-Group Functions introduced in OpenCL 2.0 were the biggest missing feature for the implementation of the described kernels.

The Work-Group Functions contain parallel primitives that perform horizontal operations across the threads of a work group, like prefix sums or reductions. The vendor supplied primitives are potentially the best implementations on each device. Doing a correct and performance portable implementation of these primitives is a hard task.

Multi Core CPUs Intel and AMD both develop OpenCL Platforms that run OpenCL code on normal multi core CPUs. Both platforms are not exclusive to the CPUs of the respective vendor, since they compile and execute OpenCL code as x86 assembly. Both platforms use multi threading and try to use modern CPUs SIMD ISAs (SSE, AVX) if possible and deemed beneficial. The explicit parallelism can lead to better vectorization compared to code generated from normal sequential C code. However, the restrictions imposed on OpenCL code, that enable suitability for GPU like architectures, do not always fit well on CPUs. For example, the algorithms and approaches to expose the amount of parallelism in the range of 10^4-10^5 threads to saturate a large GPU, are unnecessary for even large HPC CPUs, whose parallelism is in the range of around 100.

The biggest advantage of CPU platforms is in debugging, testing, and compatibility, as it does not require accelerator hardware. Widespread support is currently limited to OpenCL 1.2

Many Core CPUs, i.e. Xeon Phi Although definitely CPUs in most of their architectural and operational characteristics, Intels line of Xeon Phi products deserves a separate category in being called many core architectures. The first widely released iteration, codenamed Knights Corner (KNC), found itself still somewhat in between traditional CPUs and newer GPUs both in appearance, with being a PCIe add-in board, and operation mode, running only a very reduced and modified OS, requiring a host system to boot and operate, and executing a modified version of the x86 ISA, with both elements removed and added. The second iteration, Knights Landing (KNL), is much more autonomous. It runs an almost unmodified OS, can boot and operate without a host system, is fully compatible to standard CPUs ISA, and its ISA additions will eventually make their way into standard CPUs.

What they both still retain from GPU design philosophy is less reliance on instruction level parallelism (ILP), more cores, wider vectors and more multi threading (SMT), which all increase the degree of parallelism to several thousand threads.

Similarly to its in-betweeness in system architecture, there is a split in programming models for the Xeon Phi line. KNC could be programmed both in offloading mode, where work is queued from the host either with OpenMP extensions or with OpenCL, or in native mode, where normal multi core code would be compiled and executed as a native binary on the device. Since KNL does not depend on a host system any more, the first option is no longer available.

Intel favors OpenMP both for offloading and native code, and does not support OpenCL any more for KNC. In theory, KNL could just be treated as a normal CPU to execute OpenCL via a CPU platform. In practice, Intels CPU platform does not recognize the currently still unreleased KNL and does not run OpenCL programs on it. With public availability of KNL in the future, the publicly available CPU platform could support KNL.

In conclusion, neither iteration of the Xeon Phi currently run OpenCL. In the future, the situation could become similar to normal CPUs on KNL.

NVIDIA GPUs NVIDIA produces classical GPUs with the aforementioned characteristics: many simple cores, low ILP exploitation, high degree of SMT, wide vectors and always dependent on a host system to boot and queue work. The possible programming approaches are explicitly parallel kernel languages like OpenCL and NVIDIAs CUDA and more implicit offloading annotation models like OpenMP and OpenACC. NVIDIAs promotes its proprietary CUDA, that is conceptually very similar to OpenCL, rather than promoting OpenCL. NVIDIA does offer a OpenCL platform for their hardware, but removed OpenCL support from all of their development tools. NVIDIA currently supports only version 1.2, and has no published plans for version 2.0.

Intel GPUs Intel has included GPUs into their desktop CPUs since the Nehalem series. Integrated CPUs are constrained by much smaller power and die size envelopes compared to discrete GPUs and having to share a smaller memory bandwidth with the CPU. As an advantage, they do use the same DRAM that the CPU uses, so that memory copies from host to device memory are either faster or can be completely omitted. Intel does not have official support for OpenCL on Linux, but backs the Open Source project Beignet, that implements an OpenCL platform for integrated Intel CPUs starting from the Ivy Bridge series. Beignet currently supports OpenCL 1.2, with experimental support for 2.0. The still ongoing development status of Beignet was illustrated by a bug in Beignets on line compiler, that is triggered by the k-d tree kernels. This currently makes Beignet and Intels integrated GPUs unavailable for evaluation.

AMD GPUs AMD produces both classical and integrated GPUs. Rather than promoting a proprietary solution like NVIDIA, AMD only supports OpenCL. They are therefore much more invested into the API, which also shows in being the first vendor to support OpenCL 2.2. NVIDIA currently enjoys a much larger mind and market share in the HPC market. There was no AMD hardware available for evaluation.

2 k-d Tree

A k-d Tree (short for k -dimensional tree) is a binary space partitioning tree data structure. The problem space is split into two half spaces by an axis aligned hyper plane.

Continuing to split the resulting half spaces hierarchically generates a binary tree. All the hyper planes used within each level of the tree have the same orientation. The orientation of the planes cycles through the k different possible alignments, i.e. on level l, all hyper planes are perpendicular to the dimension $l \mod k$.

With the orientation of the splitting planes being fixed, the position has to be specified. Points inserted into the data structure are used to implicitly specify splitting planes. By

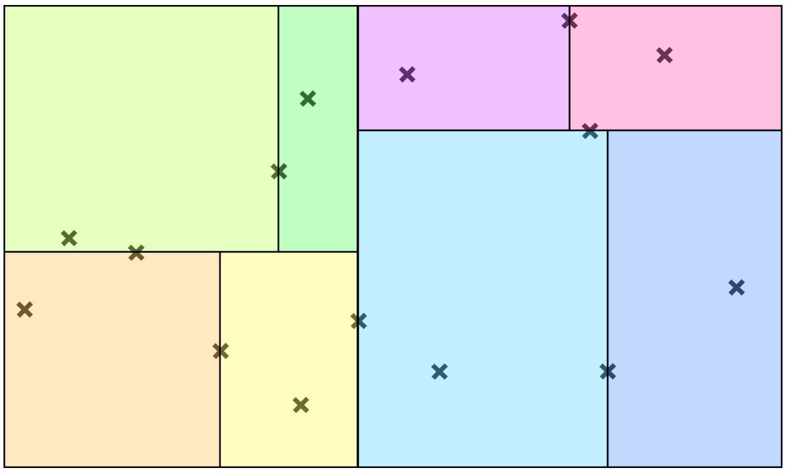

Figure 1: Cloud of 2D Points with overlaid space partitioning hierarchy

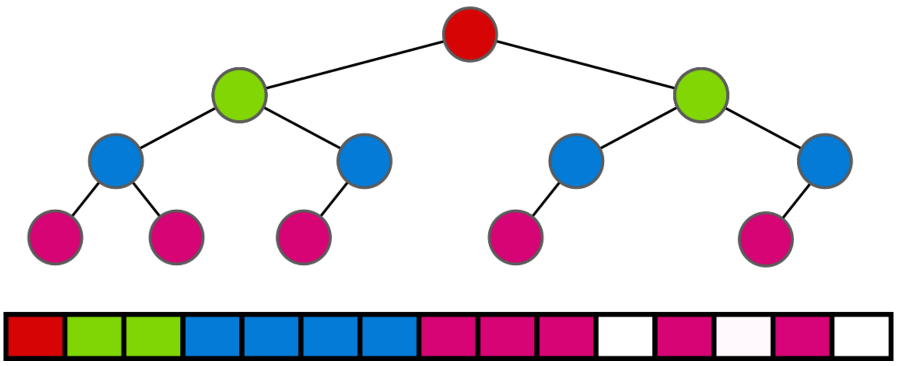

Figure 2: Linearized tree format of a tree that is built with the median point as splitting plane.

ordering the points along the axis perpendicular to the splitting plane and then selecting the median point, a splitting plane is found which separates all points into two groups of points of mostly equal sizes, which generates a balanced tree.

However, by choosing a splitting point that ensures a complete sub tree on the right half instead of the median point, the tree can be linearized without forming holes. The position of a point in the tree is implicitly described by its index, and the children of a point can be found by index calculations. Explicit storage of parent-child relationship is thus not required.

2.1 Usage Scenario of k-d trees

k-d trees allow for fast and efficient spatial queries of geometry data, e.g. a request for all points that lie in a specific axis aligned box. In the particle track reconstruction in the CMS software framework, hits in the inner detector are inserted into a five dimensional

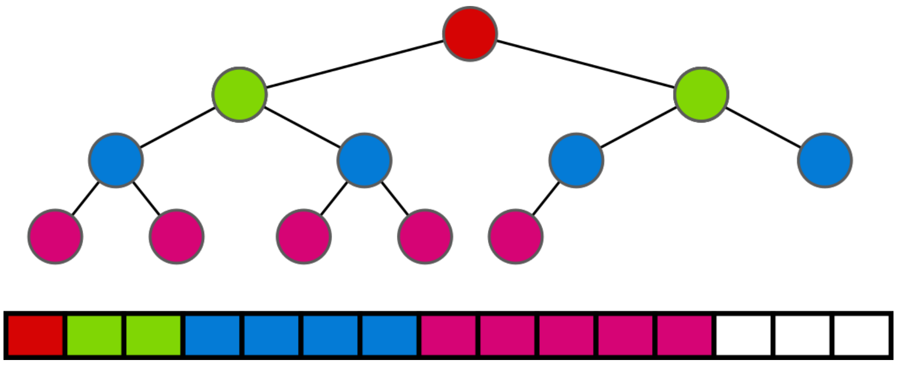

Figure 3: Linearized tree format of a tree where the splitting point is chosen to avoid holes

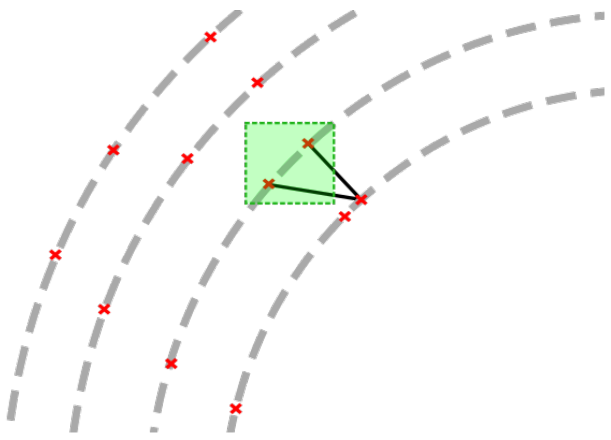

Figure 4: Example of a box query that searches for points on the second detector layer that could belong to the same track as one of the hits on the innermost detector layer

k-d tree data structure. The five dimensions of each particle are the three spatial dimensions, time and the angle of incidence. The data structure is then used to form pairs of hits or doublets that could belong to the same particle track. For each hit, prospective candidates to form doublets are obtained by querying the data structure for all hits in an appropriate five dimensional box.

2.2 Construction

2.2.1 Algorithm

A key element in constructing a k-d tree is the quick select or `nth_element` algorithm. The C++ reference `cppreference.com` describes this algorithm as:

```
void nth_element( RandomIt first, RandomIt nth, RandomIt last );
```

> `nth_element` is a partial sorting algorithm that rearranges elements in `[first, last)` such that:
>
> - The element pointed at by `nth` is changed to whatever element would occur in that position if `[first, last)` was sorted.
> - All of the elements before this new nth element are less than or equal to the elements after the new nth element.
>
> (`http://en.cppreference.com/w/cpp/algorithm/nth_element`)

If as a comparison operator a comparison of the coordinate axis perpendicular to the splitting plane is chosen, this algorithm both delivers the sought splitting point at `*nth`, if the position of `nth` is chosen as `first + median`. It also partitions the remaining points into the two half spaces, as all points `[first, nth)` belong to the left half space and the points in `(nth, last)` belong to the right half space. Sorting the range would have the same effect. However, because the points need not be sorted within the half spaces, `nth_element` does less work, which is reflected by `nth_element` having linear complexity instead of the $n \log n$ complexity of a complete sort.

After performing the `nth_element`, the point at `nth` is inserted at its index in the linear tree data structure, and the same procedure is recursively performed on the resulting half spaces `[first, nth)` and `(nth, last)`.

```
partition(first, last, dimension) {
    nth_element(first, nth, last, order by dimension);
    insert *nth into tree
    partition(first, nth, (dimension+1) % k)
    partition(nth + 1, last, (dimension+1) % k)
}
```

2.2.2 Implementation

Rather than implementing the Depth First (DFS), recursive approach as in the pseudo code, an iterative, Breadth First (BFS) algorithm is chosen. Each level of the tree is a separate kernel launch. Since OpenCL 2.0, it is possible to enqueue new work from the device, which would enable the different tree branches to run asynchronously, but this has not been examined in this report due to support and time constraints. The different branches on each level are processed in parallel. As the construction progresses from the root of the tree to the deeper branches, the branch count goes up, starting at a single node, but the amount of points under each branch decreases down to just one point per node. Two different algorithms and parallelisation strategies for different depth have therefore been implemented.

The first strategy launches a workgroup per node. This makes the cooperative work on each branch easier, as communication and synchronization is possible in a workgroup. The `nth_element` primitive is implemented with a truncated radix sort. In contrast to a full radix sort, only the bucket that contains the nth element is sorted further. A moving bit mask of four bits, starting from bits 27-31 down to 0-3, is used to assign points to 16 buckets. Each thread creates a thread local histogram of points in the partition.

```
for(int i = threadId; i < size; i += workGroupSize) {
    localBuckets[ bucketIndex(pointsSrc[i], lowBit, highBit) ]++;
}
```

Listing 1: thread local histograms created in a grid stride loop

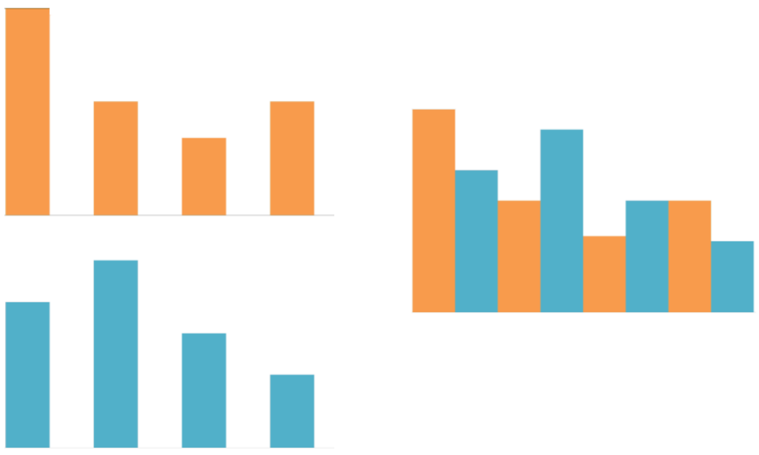

Figure 5: Interleaved Thread Local Histograms with two Threads

The thread local histograms are then interleaved, so that the counts of all threads for the first bucket are after another, than those of the second bucket and so on. The exclusive prefix sum of this interleaved histogram buffer is computed, which computes the count of all values smaller than the current bucket plus the preceding values of the same buckets of threads with smaller ids. The bucket with the nth element in it is the one where the count of values smaller than the bucket is smaller than the nth position, but the count of values smaller or equal is larger than the nth position. All threads again iterate cooperatively over all points, select all that fall into the chosen bucket with the nth element in it, and put them into a new buffer. The positions that each thread assigns the values it found to are determined by the values of the prefix sum. It contains the sum of all threads counts smaller than a specific thread, so that that thread can safely assign its first value to one position above, and has enough space to assign all of its values, since the next bigger thread will leave exactly enough head room.

```
for(int i = threadId; i < size; i += workGroupSize) {
    if( bucketIndex(pointsSrc[i], lowBit, highBit) == chosenBucket) {
        pointsDst[prefixSum[threadId]++] = pointsSrc[i];
    }
}
swap(pointsSrc, pointsDst);
```

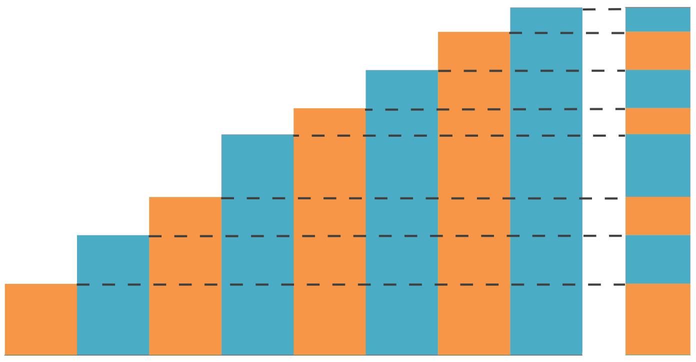

Figure 6: Exclusive Prefix Sum Indicates Buffer Positions

The source buffer and the destination buffer now only containing the values of the chosen buffer are swapped, and the procedure repeats with the next lower bit mask. This repeats either down to the lowest four bits, or until the chosen bucket only contains one value, which must then be the nth element.

A similar count, scan and scatter operation is then performed to partition all points into on of the three buckets smaller, equal or greater than the nth element.

The second strategy for deeper levels uses just a single thread per node, and uses a simple bubble sort full sorting scheme to find the nth element and to partition the range. Since the point count at this stage should be less than 32, often just two or three values, bubble sort, while not optimal, is not prohibitively slow.

2.3 Search

2.3.1 Algorithm

In order to perform a box query, the root node of the tree is checked whether it lies in the box and added to the result set if it does. It is then checked whether the box overlaps at least partly the left and the right half space, and if so, the same procedure is hierarchically repeated for the child nodes of the root nodes.

```
search( rootNode, box) {
    if( rootNode in box ) add rootNode to results
    if( box overlaps left half space)
        search(leftChild, box)
    if( box overlaps right half space)
        search(rightChild, box)
}
```

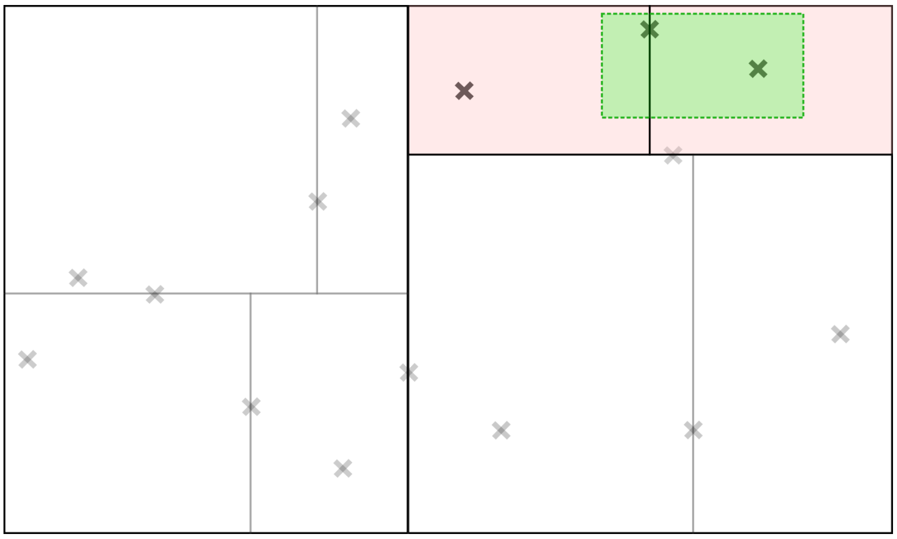

Figure 7: A partily progressed box query on the second level. The root node, the left half space, and the lower half of the right half space are already discarded, and only the upper half of the right half space is still considered

2.3.2 Implementation

Similar to the Construction implementation, BFS is preferred over DFS. A large independent count of queries already supplies a lot of parallelism. However, there might not be enough queries to saturate a larger GPU, so exploiting parallelism inside of a query is desirable. A configurable thread count or gang work collaboratively to perform a single search query. The gang size can be as low as one, if enough parallelism is available and the architecture favors it, and as big as the maximum work group size, as all threads in a gang need to be in the same work group. Choosing the gang size similar to the vector width can be advantageous, as this keeps the granularity of thread divergence above the vector level. Good values where found in the range of 4-32.

```
srcBuf[0] = rootNode;
for( int level = 0; iter < maxLevel; level++) {
  for( int branch = threadId; branch < branchCount; branch += gangSize) {
    if( nodeInBox( box, srcBuf[branch] )
       resultSet.add( srcBuf[branch]);
    if( overlapsBox( srcBuf[branch]->leftChild , box) )
      dstBuf.add(srcBuf[branch]->leftChild);
    if( overlapsBox( srcBuf[branch]->rightChild , box) )
      dstBuf.add(srcBuf[branch]->rightChild);
  }
  barrier();
  swap(srcBuf, dstBuf);
}
```

The implementation keeps two buffers with branches that are to be evaluated, `srcBuf` for the current level and `dstBuf` for the next Level. `srcBuf` initially only contains the root node. The outer loop iterates over all tree levels. In the inner loop, the gang of threads then collaboratively iterates in a grid stride loop with the gang size as grid size over the current list of branches, `srcBuf`. Each branch is checked whether the node itself is a point in the box, if so, it is added to the result set. The left and right half spaces are checked whether they are overlapped by the box, if so, they are added to the list of branches to evaluate on the next level, `dstBuf`. After the inner loop, all threads synchronize and the two buffers are swapped.

The three add operations are realized with an atomic counter increase. Since both the next buffer and the result buffer are private to each gang and only a part of the threads will push data back, the number of threads that hit the same counter is low.

```
void add(T* buf, int bufCount, T val) {
    int oldIndex = atomic_inc( bufCount );
    buf[oldIndex] = val;
}
```

2.4 Results

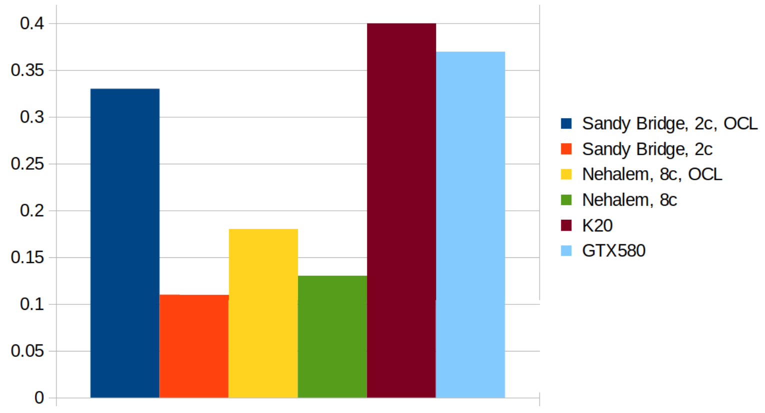

Figure 8: Time in seconds to construct a k-d tree from a point cloud of 500000 points. OpenCL measurements include transfer time. Less is better

The results show the OpenCL implementations to be slower in general. On the small two core machine, the native implementation is 3 times faster, on the larger, 8 core machine the parallel OpenCL version still takes longer than the sequential native version. On the two tested NVIDIA GPUs, the construction takes about four times longer than for the native CPU implementation.

The native CPU version uses the `nth_element` function from the standard library. While not parallel, it is still supposed to be well optimized. This function had to be specifically implemented for this project, and a lot of the trade-offs between performance and implementation time had to be decided in favour of implementation time. OpenCL version 2.0 contains some function primitives, like work group prefix sums, that would have saved implementation and execution time, but were not available in most of the targeted platforms.

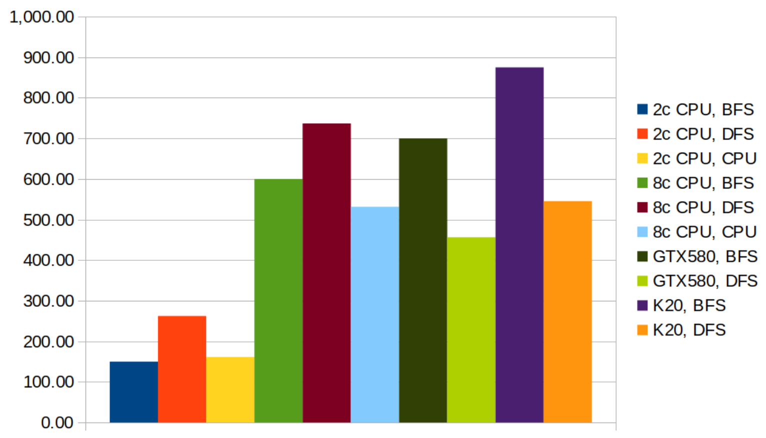

Figure 9: Search performance in Megaqueries per second in a k-d tree consisting of 500000 points. Higher is better. BFS and DFS use OpenCL, CPU is native code

On the CPU systems, the previously existing depth first CPU implementation is faster than both the for this project implemented breadth first approach and the native CPU code. The two core and the eight core CPU do not have the same architecture and clock rates, so that precise scaling behaviour cannot be determined from the data. On both GPUs, the BFS variant outperforms the DFS variant.

That DFS favours CPUs and BFS favours GPUs is somewhat expected. The BFS approach was chosen specifically to avoid code patterns that do not run well on GPUs. CPUs do not have these limitations and rather benefit from higher data locality which helps to make their caches useful.

2.5 Excursus: Beignet Bug Report

All k-d tree kernels employ a code pattern, that reliably triggered a segmentation fault in Beignets online compiler compiled from the git commit `cb5506`. A reduced sample of one of the kernels was produced, that retained only what was necessary to trigger the crash.

```
kernel void crashing_sample(global uint* A, global uint* B,
                                    uint nPoints, uint nQueries) {
  for (uint id = 0; id < nQueries; id++) {
    A[0] = 0;

    uint maximum_depth = nPoints;
    for (uint depth = 0; depth <= maximum_depth; depth++) {
        global uint* temp = A;
        A = B;
        B = temp;
      }
    }
}
```

The common element in the crashing kernels is this swap pattern, that helps to iteratively process intermediate data. The kernels have two temporary working buffers, where `A` is the buffer where the source data is read from and `B` is where the intermediate results are written to. At the end, the two buffers are swapped, and the newly created data is now the source data, and the old source data is overwritten by new data. In the sample, no actual data processing takes place, the buffers are just swapped.

Beignets online compiler gets stuck in an infinite recursive loop while doing data flow analysis, runs out of stack space, and crashes. A patch posted by one of the developers fixed the segmentation fault. However, in the course of further testing and benchmarking, it was found that the search kernel under counted the number of found points very significantly. It was possible to provoke similar wrong results in the reduced sample.

```
kernel void wrong_output_sample( global uint* A, global uint* B,
                                 uint nPoints, uint nQueries) {
  printf("\n\n");
  for (uint id = 0; id < nQueries; id++) {
    A[0] = 0;

    uint maximum_depth = nPoints;
    for (uint depth = 0; depth <= maximum_depth; depth++) {
      for (uint i = 0; i < nPoints; i++) {
        B[i] = A[i] + 1;
        printf("%u␣", B[i]);
      }
      printf("\n");
      global uint* temp = A;
      A = B;
      B = temp;
    }
    printf("\n");
  }
}
```

Actual data processing is added to the swap pattern loop, where the previous buffer is simply counted up by one. If buffer `A` is completely initialized with 7, the expected result, that was produced by four other OpenCL platforms, would be:

```
1 8 8 8
2 9 9 9
```

```
3 10 10 10
4 11 11 11
5 12 12 12

1 13 13 13
2 14 14 14
3 15 15 15
4 16 16 16
5 17 17 17
```

[. . .]

Instead, the result is:

```
1 1 1 1
2 2 2 2
1 1 1 1
2 2 2 2
1 1 1 1

1 2 2 2
1 1 1 1
2 2 2 2
1 1 1 1
2 2 2 2
```

[. . .]

This does once again have the look of a bug in the dataflow analysis. The bug (`https://bugs.freedesktop.org/show_bug.cgi?id=97190`) is still open as of the time of writing.

3 Linked Doublet Graph Path Enumeration

3.1 Usage Scenario of the Doublet Path Enumeration

A previous stage of the detector data analysis forms doublets of hits in adjacent detector layers that could belong to the same particle track. Each hit usually belongs to several different doublets. The Doublet Path Enumeration Algorithms links several (in this particular case three) doublets to form prospective particle tracks.

3.2 Algorithm

The doublet linking algorithm has three phases:

1. For each hit, create a list of doublets this hit is used in

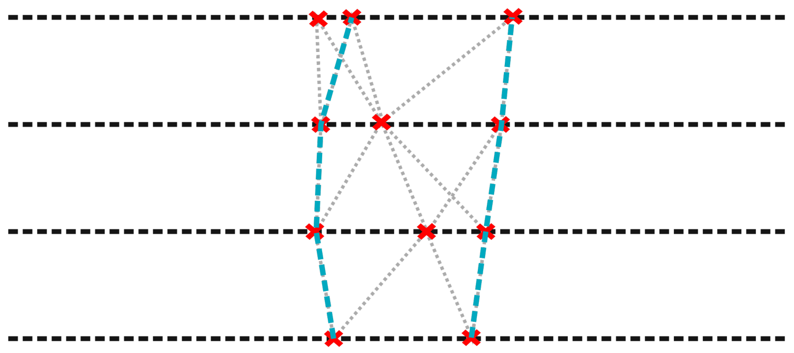
Figure 10: Doublets between hits on different layers of the detector (in grey) are connected to form complete prospective particle tracks (in teal)

2. For each doublet, create a list of doublets in the next layer that this doublet could be linked to. Conditions are that the two doublets share a points, have similar RZ-alignment, and have similar curvature

3. Find all paths through the graph of linked doublets

3.3 Implementation

The first phase is parallelized over all doublets. A grid stride loop iterates over all doublets. For each doublet, the doublet id is added to the list of doublets the inner point belongs to.

```
for( id = tid; id < doubletCount; id += threadCount) {
  doublets[innerPointIds[id]].push_back(id);
}
```

The second phase is parallelized over all doublets again. For each doublet, via the inner point id, a list of doublets, that use this inner point as an outer point, can be looked up. All these pairs of doublets are checked whether they are aligned in the radius/depth plane, and whether they have a similar curvature. If so, the second doublet is added to the list of doublets the first doublet is connected to. This creates a connection list of the unidirectional graph spanned by the doublets. In this graph, points on each layer are only connected with points on the next upper layer.

```
for( id = tid; id < doubletCount; id += threadCount) {
  for( otherDoublet = 0...doubletCounts[innerPointIds[id]] ) {
    if( RZ alignment and similar curvature ) {
      connectedDoublets[id].push_back(otherDoublet);
    }
  }
}
```

The third phase enumerates all possible paths from the lowest to the highest layer through the unidirectional graph. Since connections between points are always towards points on the next higher level, in case with four layers, each path has exactly four nodes.

The implementation starts out by enumerating all paths with three nodes, or two doublets. The parallelization is over the doublets in the lowest layer. Each thread then iterates over the second layer doublets linked to its first layer doublet, and adds all combinations to the result set. All threads push back into the same result set. The `atomic_inc` operation will quickly become a bottleneck here, as all threads need to increase the same counter for every push back operation. To reduce the contention on the result sets size counter, each thread reserves the complete amount of space it needs for its elements in one `atomic_add` operation. Each thread can easily determine the amount of elements it will push back. This cuts contention down by the average fan out of each node.

After that first pass, each further pass creates a new result set of paths with one more node. In the case with four layers, each paths has only three doublets, and the algorithm is finished after one further pass. The passes are parallelized over the the paths created by the previous pass. A thread iterates over all possible 3 doublet continuations of its 2 doublet input path.

```
for( path = tid; path < pathCount; path += threadCount ) {
  threadStart = atomic_add(newPathCount, connectedDoubletCount[path]);
  for( i=0...connectedDoubletCount) {
    threadStart[i] = newPath(path, doublet[i]);
  }
}
```

3.4 Results

Figure 12 shows the results of measuring the execution time of the implementation. While the GPU results were very repeatable, he CPU results fluctuated around 30%. The plotted results are the median results over several measurements. Both the high fluctuations, the generally bad results little speedup 8 core CPU versus the 2 core CPUs might be attributable to a bad implementation of the atomic operations. The timing of phase 1 also includes both host and device allocation and copy operations, which might be a reason for the large difference between the two CPU systems.

The GPUs perform much better than the CPUs. However, the newer and more potent K20 is slower than the older GTX580. The reason for this is the heavy use of atomics. More concurrent threads mean more contention, which limits parallelization speedup. Generally, the older Fermi architecture of the GTX580 is narrower in terms of execution units, but runs at higher clock speeds, and also expends more hardware towards ILP extraction. The much wider Kepler design normally offsets somewhat simpler and slower clocked hardware. In this case, the use of atomics reduces the scaling to more threads, that would have been necessary for the K20 to run at its full speed.

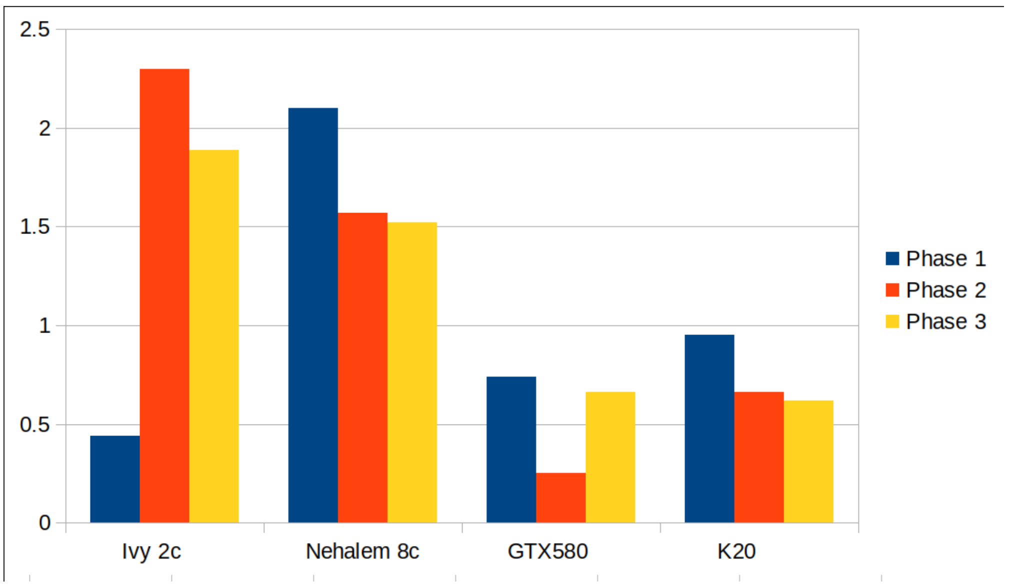

Figure 11: Time in ms to execute the three phases of the Path Enumeration Algorithm

4 Conclusions

OpenCL version support OpenCL is held back by the slow adoption of newer versions of the standard by hardware vendors. OpenCL version 2.0 is 3 years old as of the time of writing, yet all of the examined platforms only support OpenCL 1.2. Memory management, built in work-group functions and dynamic parallelism are OpenCL 2.0 features that would have increased implementation and execution speed for the projects reported about here.

OpenCL device support The initial goals expressed in the project description encompassed evaluation of the implemented kernels on a wider variety of parallel devices. A usable state of support was found only for two device classes, conventional multi core CPUs and discrete GPUs.

Parallel HEP algorithms The typical algorithms in High Energy Physics, of which a few are presented in this report, are much harder to adopt for large scale parallelism and thread coherence required by GPU like devices. It is easy to rewrite an algorithm for OpenCL in the same parallel style that works on multi core CPUs. However, to extract the kind of parallelism that the devices that OpenCL targets require, algorithms need to have threads work collaboratively and data parallel with maximum coherence in data and control flow between each other. A lot of patterns, that are easy to implement sequentially or already standard library functions like `nth_element`, have to be reimplemented in parallel.

Irregular data sizes require flexible parallelism. A frequent pattern is many threads adding values to a list. This can be solved by either a counting/interleaving/scanning pattern to find partition positions for each threads values in a compact array or with an atomic add operation. The first is fast but requires more work initially, the other scales bad with the parallelism of push backs.

OpenCL on CPUs For the kernels examined in this report, the OpenCL implementations are usually competitive with the native CPU implementations. This however already includes multi core parallelization, which comes for free with OpenCL, but is not always implemented for the CPU versions. It has not been examined whether the CPU platforms generate vectorized code. The CPUs simpler SIMD instruction set requires even more control and data coherence than GPUs, and it is not unlikely that the on line compiler does not generate vector instructions because correctness cannot be guarenteed.

OpenCL on GPUs For classical discrete GPUs, the results sometimes show good speedups compared to a small single socket CPU, but nowhere near the ratio of their raw potential. In some cases, a wider, newer GPU (K20) is slower than is predecessor (GTX580), which hints a lack of scaling with more parallelism. Both the large degree in parallelism and the thread coherence which are required are probably not sufficiently present. Having to resort to atomics to to guarantee correctness also hurts scalability. This is an inherent problem of HEP algorithms, but also an issue of implementation quality. While the efforts going into the presented implementations largely center about providing the aforementioned properties, they are far from the optimal approaches. Much more tuning and performance engineering needs to be expended here.