Point Distribution Tensor Computation on Heterogeneous Systems

Ivan Grasso\textsuperscript{14}, Marcel Ritter\textsuperscript{234}, Biagio Cosenza\textsuperscript{1}, Werner Benger\textsuperscript{345}, Günter Hofstetter\textsuperscript{2}, and Thomas Fahringer\textsuperscript{1}

\textsuperscript{1} Institute for Computer Science  
\textsuperscript{2} Institute for Basic Sciences in Engineering Science  
\textsuperscript{3} Institute for Astro- and Particle Physics  
\textsuperscript{4} University of Innsbruck, Austria  
\textsuperscript{5} AHM Software - Airborne Hydromapping  
\textsuperscript{1} Center for Computation & Technology at Louisiana State University

Abstract

Big data in observational and computational sciences impose increasing challenges on data analysis. In particular, data from light detection and ranging (LIDAR) measurements are questioning conventional methods of CPU-based algorithms due to their sheer size and complexity as needed for decent accuracy. These data describing terrains are natively given as big point clouds consisting of millions of independent coordinate locations from which meaningful geometrical information content needs to be extracted. The method of computing the point distribution tensor is a very promising approach, yielding good results to classify domains in a point cloud according to local neighborhood information. However, an existing KD-Tree parallel approach, provided by the VISH visualization framework, may very well take several days to deliver meaningful results on a real-world dataset. Here we present an optimized version based on uniform grids implemented in OpenCL that is able to deliver results of equal accuracy up to 24 times faster on the same hardware. The OpenCL version is also able to benefit from a heterogeneous environment and we analyzed and compared the performance on various CPU, GPU and accelerator hardware platforms. Finally, aware of the heterogeneous computing trend, we propose two low-complexity dynamic heuristics for the scheduling of independent dataset fragments in multi-device heterogeneous systems.

Keywords:

1 Introduction

Point datasets are present in many scientific domains. Smoothed particle hydrodynamics methods in astrophysics [4], echo sounding in engineering, 3D surface reconstruction [16] and urban reconstruction [20] in graphics are typical examples where data generated by numerical simulations or observations are processed as point primitives. Today’s range-sensing devices are
capable of producing highly detailed point datasets containing hundreds of millions of sample points. Light detection and ranging (LIDAR) technology, in particular, allows collecting millions of data points e.g. from airborne scanners in order to produce high-resolution digital elevation maps. However, depending on the application, large point datasets may require a prohibitively high computational power for processing.

This paper is motivated by the LIDAR surveying application of shallow waters [7] where, starting from a point dataset, a second order tensor field is computed and used as a basis for several other algorithms such as point classification and geometry reconstruction [22]. A first CPU-based parallel implementation of such a point distribution tensor was implemented in the VISH visualization shell [5]. VISH is a productive framework that provides functionalities for both efficient data processing and visualization of big data. However, as surveyed datasets grow from several thousand of points to many millions of points, the tensor computation becomes a bottleneck for data processing.

In this paper we focus on exploiting the computational power of emerging heterogeneous computing systems in order to improve the tensor computation of massive datasets of millions of points. Our study makes the following contributions: First, we implemented a new tensor computation code in OpenCL using a uniform grid space partitioning approach, and evaluated its performance against the current KD-Tree implementation available in VISH. Second, we investigated the performance of the implemented code on 8 different devices, comprising four GPUs, three CPUs and one accelerator, from desktop and server domains. Finally, we proposed two low-complexity dynamic heuristics for the scheduling of independent dataset fragments and compared them with three static scheduling heuristics in two multi-device heterogeneous systems.

2 OpenCL Programming Model

OpenCL [14] is an open industry standard for programming heterogeneous systems composed of devices with different capabilities such as CPUs, GPUs and other accelerators. The platform model consists of a host connected to one or more compute devices. Each device logically consists of one or more compute units (CUs) which are further divided into processing elements (PEs). Within a program, the computation is expressed through the use of special functions, called kernels, that are, for portability reasons, compiled at runtime by an OpenCL driver. A kernel represents a data-parallel task and describes the computation performed by a single thread, which is called work-item in OpenCL. During the program execution, based on an index space (N-Dimensional Range), a certain number of work-items are generated and executed in parallel. The index space can also be subdivided into workgroups, each of them consisting of many work-items. The exchange of data between the host and the compute devices is implemented through memory buffers, which are passed as arguments to the kernel before its execution. In the past few years, OpenCL has emerged as the de facto standard for heterogeneous computing, with the support of many vendors such as Adapteva, Altera, AMD, ARM, Intel, Imagination Technologies, NVIDIA, Qualcomm, Vivante and Xilinx.

3 Tensor Computation

For a set of $N$ points $\{P_i : i = 1, ..., N\}$ the point distribution tensor $S$ at the point $P_i$ is defined as:

$$S(P_i) = \frac{1}{N} \sum_{k=1}^{N} \omega(|t_{ik}|)(t_{ik} \otimes t_{ik}^T),$$

(1)
Figure 1: Input point distribution (left) and output tensor (right) of the river Rhein dataset. For the input points, the height (z-axis) drives a colormap (left) while the output tensor is used for coloring (planarity) and surface shading in VISH (right).

whereby $\omega(x) = \theta(r - x)$ is a threshold function dependent on a radius $r$ [22], $t_{ik} = P_i - P_k$, $^\tau$ is the transpose and $\otimes$ denotes the tensor product. A graphical result of the computation is depicted in Figure 1.

The naive approach for the tensor computation is therefore to test every point with all the others, leading to a quadratic algorithmic complexity. In real models composed of millions of points this approach is not applicable due to the inherent performance problem. To mitigate this problem spatial partitioning methods have been investigated [2, 25, 26].

**KD-Tree Implementation.** The tensor field computation algorithm, currently implemented in VISH, makes use of a KD-Tree data structure to find the neighbors of a certain point. After the tree building phase, in which the points of the dataset are inserted into the KD-Tree, the computation of the tensor distribution is executed for each point with a series of range queries dependent on a given search radius (threshold function). The computational loop over the points was parallelized using OpenMP with dynamic scheduling and packets of 10000 loop iterations. The KD-Tree code was integrated in a computational VISH module, and implemented via C++ templates and STL containers.

**Uniform Grid OpenCL Implementation.** A uniform grid space partitioning approach involves a spatial partitioning of the model system into equally-sized boxes (cells) containing different numbers of points. It is important to ensure that the grid box size is not smaller than the radius size, as this would force the algorithm to check many surrounding grid boxes. On the other hand, if the grid box is larger than the radius, each box would contain numerous points and the process of locating neighbors would once again be checking many points outside the radius area. In our case, the uniform grid approach is effective because the radius is an input parameter of the program and therefore we are able to tune the grid box size accordingly.

We implemented the tensor computation application in OpenCL using a uniform grid space partitioning approach. We used a grid with a cell size of two times the radius, which implies that each point can only interact with points in the neighboring cells (27 in a 3D space). The complete program, described in Algorithm 1, is composed of three phases: initialization, computation and finalization. During the initialization phase, the OpenCL devices are initialized, the OpenCL kernels are compiled and the metadata of the dataset is loaded. The metadata contains information about the number of fragments present in the dataset, the number of points for each fragment, plus other additional information useful for the graphical visualization. The dataset consists of independent fragments of spatially ordered points to facilitate the
Algorithm 1 The OpenCL tensor computation algorithm

```
1: devices_initialization() ▷ Initialization Phase
2: metadata ← load_dataset_metadata()
3: for all fragments in dataset do ▷ Computation Phase
4:   pts_ar ← load_points_data(fragment)
5:   write_points_to_device(pts_ar)
6:   create_uniform_grid(pts_ar, radius){
7:     hash_ar ← compute_hash_values(pts_ar)
8:     index_ar ← sort_points_indices(hash_ar)
9:     begin_end_ar ← compute_interval(hash_ar)
10:   }
11:   compute_tensor(pts_ar, index_ar, begin_end_ar)
12:   tsr_ar ← read_tensor_from_device()
13:   write_tensor_to_disk(tsr_ar)
14: end for
15: devices_finalization() ▷ Finalization Phase
```

data manipulation and visualization. Each fragment contains a small percentage of replicated data necessary for the computation of the tensor algorithm at points close to the border of the fragment. Once the initialization phase is completed the system is ready to schedule the fragments on the available devices and the computation phase will start.

For each fragment the point’s coordinates will be loaded in main memory and transferred to the device memory where the computation will take place. On the device, the uniform grid will be created and used during the tensor computation in the search for the neighboring points. Once the computation is done, the computed tensor data is transferred back to the host’s main memory and finally saved to the disk. The finalization phase releases all the devices and the used memory.

The steps necessary for the creation of the uniform grid are described in Algorithm 1 (lines 6-10). The algorithm consists of multiple OpenCL kernels. The first kernel (line 7) calculates a hash value for each point based on its cell ID and stores them in an array in device main memory (hash_ar). The array is then sorted based on the cell IDs while updating at the same time the order of the point IDs. Sorting is performed using a bitonic algorithm. The result of this computation is an array of point IDs sorted by cell (index_ar). The last kernel (line 9) is then executed to find the begin and the end position of any given cell. The kernel generates an OpenCL work-item for each point and compares the cell ID of the current point with the cell ID of the previous one in the hash_ar array. If the two indices are different, the current work-item ID is used as start index of the current cell and the begin_end_ar array is updated using a scattered write operation. During the execution of the compute_tensor kernel (line 11), using the begin_end_ar and index_ar arrays, we calculate the neighbor cells for each point in the fragment and for each point present in the cells we compute the difference to the current point in each dimension (x, y, z). If the length of the difference vector is less than the radius, the tensor array and the points counter are updated. Finally, in the last step, each element of the tensor array is divided by the points counter.

4 Scheduling Independent Fragments
As previously mentioned in Section 3, the tensor computation is applied on single fragments that compose the complete dataset. The fragments are completely independent of each other and can be computed in parallel using the available devices present in the system. During program
execution a scheduler is responsible for the allocation of the fragments among the heterogeneous
devices. The scheduling problem has been extensively investigated and numerous methods have
been reported in the literature [6, 18, 17]. In our program we implement two low-complexity
scheduling heuristics: SimpleH and SimpleHS. SimpleH analyzes the dataset metadata and
sorts the list of fragments based on the number of points contained in each of them. The
algorithm then proceeds by dynamically assigning the fragment with the smallest number of
points to the slowest device and the fragment with the biggest number of points to the fastest
device. Following this pattern, the scheduler continues to dynamically assign fragments until
all of them are processed. SimpleHS follows a similar pattern. A fragment is assigned to the
slowest device, if the predicted execution time of the fragment on that device is lower than the
predicted execution time of all the remaining fragments on the fastest device. The execution
time for each fragment is predicted with a quadratic regression model using the number of points
of the fragment. During the program execution, information regarding number of points per
fragment and execution times are stored. These information will then be used to build a more
accurate model whenever the slowest device is ready to compute a new fragment. Although
for simplicity the heuristic algorithms are described taking into consideration only two devices,
they can be applied to heterogeneous systems composed of a single slow device (CPU) and
multiple equally fast devices (e.g. GPUs). In Section 6 we evaluate and compare SimpleH and
SimpleHS with three heuristics which are widely used to address the problem of scheduling
independent tasks in heterogeneous computing systems: Min-Min [13, 6], Max-Min [13, 6], and
Sufferage [19]. Because these are static heuristics, it is assumed that an accurate estimation
of the expected execution time for each fragment on each device is known prior to execution
and contained within an ETC (expected time to compute) matrix. The Min-Min heuristic
proceeds by assigning a previously unassigned fragment to a device in every iteration. The
assignment is decided based on a two-step procedure. In the first step, the algorithm computes
the minimum completion time (MCT) of each unassigned fragment over the devices in order
to find the best device which can complete the processing of that fragment at earliest time.
This decision is made taking into account the current loads of the devices and the execution
time of the fragment on each device. In the second step, the algorithm selects the fragment
with the minimum MCT among all unassigned fragments and assigns the fragment to its best
device found in the first step. The Max-Min heuristic differs from the Min-Min in the fragment
selection policy adopted in the second step of the fragment-to-device assignment procedure.
Unlike Min-Min, which selects the fragment with the minimum MCT, Max-Min selects the
fragment with the maximum MCT and then assigns it to the best device found in the first
step. Sufferage is also similar to Min-Min but adopts a different fragment selection policy. In
the first step of the process, the algorithm computes the second MCT value in addition to the
MCT value for each fragment. In the second step, the sufferage value, which is defined as the
difference between the MCT and the second MCT values of a fragment, is taken into account.
Sufferage selects the fragment with the largest sufferage and assigns it to the best device found
in the first step.

5 Experimental Environment

In order to evaluate the performance of the KD-Tree and OpenCL implementations presented
in Section 3, we use a dataset of 58 million points, generated using a combination of LIDAR
and echo sounding data captured at the river Rhein in Rheinfelden [7]. The dataset is stored in
the HDF5 [23] format, based on the scientific data format F5 [21, 3], to be easily manipulated
with the VISH infrastructure. The dataset is composed of 65 fragments that contain between
one thousand and 3.5 million points each.
To represent the broad spectrum of OpenCL-capable hardware we selected eight devices, comprising four GPUs, three CPUs, and one accelerator. Their device characteristics as reported by OpenCL are summarized in Table 1. To exploit the computational capabilities of heterogeneous machines, we evaluated different scheduling heuristics. The experiments were performed on two different heterogeneous target architectures composed of three OpenCL devices: two GPUs and one CPU. The first platform, \( mc_1 \), consists of an Intel i7-2600K CPU and two NVIDIA GTX 480, while the second, \( mc_2 \), holds two Intel Xeon E5-2690 v2 CPUs (reported as a single OpenCL device) and two AMD Fire Pro S9000 GPUs. For the static scheduling heuristics we utilized, as estimation time for each fragment (ETC matrix), the actual time that the fragment will take to be computed on the different devices. Differently, for the computation of the coefficients in the SimpleHS heuristic, we used the multi-parameter fitting present in the GNU Scientific Library.

All the benchmarked programs were compiled with GCC version 4.8.1 with the -O3 optimization flag. In each different device, the OpenCL kernels were compiled by the respective vendor compilers at runtime during the program initialization. All the experiments were conducted on the previously described dataset. The measurements were collected for the computational phase of the program, excluding the initialization and finalization phases. We repeated each experiment 10 times and we computed the mean value and the standard deviation of the measured performance. In all the presented experiments, the standard deviation is negligible, thus we do not report it.

## Performance Analysis

### KD-Tree and OpenCL Implementations.

To compare the performance of the KD-Tree version and our OpenCL implementation, we executed the tensor computation on the input dataset on the same multi-core CPU (Intel i7-2600K). Both implementations are parallel: the KD-Tree version uses OpenMP to parallelize the loop over all points, while the OpenCL approach is inherently parallel. The building phase of the tree in the KD-Tree implementation is sequential, however, it represents a very small part of the overall run time. The OpenCL version of the program experiences a significant speedup (24×) over the currently implemented VISH KD-Tree version, reducing the execution time from 1 hour to 150 seconds. The performance improvement comes from different reasons. First, grid data structures are more suited for range queries (all the particles around a point in a given radius) while KD-Tree structures are more suited for k-nearest neighbors queries (first N-points close to a given point). Second, vectorization is rather hard in KD-Tree codes where many data-dependent branches are present. In contrast, the uniform grid OpenCL code can be more easily autovectorized by compilers.
Third, we applied a few code optimizations that improve the performance of the OpenCL code. However, the optimizations only partially affect the speedup over the KD-Tree version, which remains significant even in their absence (12.9×).

Heterogeneous Devices. Since OpenCL supports heterogeneous devices, we analyzed the performance of our OpenCL code on a set of heterogeneous architectures described in Table 1. Figure 2a depicts the percentage of execution time spent in the different phases of the OpenCL tensor computation described in Algorithm 1. The blue color represents the transfer of the fragment points to the device (line 5), the green color represents the time spent building the uniform grid structure (line 6-10), the yellow color indicates the time spent in the computation (line 11), while the red color identifies the transfer of the tensors to the host device (line 12). In all the tested hardware the movement of data does not represent an important part of the execution time. Write and Read functions are always under 5% of the total time. The only exception is the AMD Fire Pro S9000 where the data transfers represent 9.0% and 11.6% of the execution time, respectively. This is mainly due to the small amount of time spent in the tensor computation thanks to the strong computational capabilities of the device.

In Figure 2b we present the performance comparison of the heterogeneous architectures. The speedup of the CPUs respects the characteristics of the hardware. The AMD Opteron, with a higher number of compute units but a lower clock rate, experiences a 1.6× speedup over the Intel i7 while the Xeon, with 40 compute units and a similar clock rate, reaches a 4.5× speedup. All the GPUs show significant improvements in performance compared to the Intel i7. The desktop GPUs AMD Radeon 5870 and NVIDIA GTX 480 reach a speedup of 4.7× and 12.0×, respectively. The server GPUs NVIDIA K20m and AMD Fire Pro S9000, designed for the HPC market, show a speedup of 14.5× and 23.8×, respectively. It is worth underlining that although the NVIDIA K20 offers higher theoretical peak performance, in our test the AMD Fire Pro S9000 is around 1.5 times faster. The only accelerator present in our test is the Intel Xeon Phi. Although its peak performance is comparable with the tested server GPUs, it reaches only a speedup of 4.4× compared to the Intel i7. The difference in performance between the GPUs and the Xeon Phi is difficult to investigate as it derives from the differences in the architecture and from the different maturity of the OpenCL toolchains.

In conclusion the results show that the problem is well-suited for massively parallel GPU architectures, reducing the processing time of the complete input dataset to 6.3 seconds in case of the AMD Fire Pro S9000.
Table 2: Performance of the different scheduling heuristics in two heterogeneous systems

<table>
<thead>
<tr>
<th>Device</th>
<th>Mc1-CPU1</th>
<th>Mc1-GPU1</th>
<th>Mc1-GPU2</th>
<th>Mc2-CPU1</th>
<th>Mc2-GPU1</th>
<th>Mc2-GPU2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sufferage</td>
<td>5976.34</td>
<td>5830.16</td>
<td>5974.84</td>
<td>2708.42</td>
<td>2706.41</td>
<td>2709.77</td>
</tr>
<tr>
<td>Min-Min</td>
<td>0.00</td>
<td>5993.26</td>
<td>6502.75</td>
<td>1341.58</td>
<td>2986.75</td>
<td>2986.75</td>
</tr>
<tr>
<td>Max-Min</td>
<td>5980.55</td>
<td>5988.02</td>
<td>5988.23</td>
<td>2711.67</td>
<td>2712.34</td>
<td>2712.34</td>
</tr>
<tr>
<td>SimpleH</td>
<td>6124.42</td>
<td>5962.76</td>
<td>5984.76</td>
<td>2758.30</td>
<td>2797.53</td>
<td>2838.69</td>
</tr>
<tr>
<td>SimpleHS</td>
<td>4807.08</td>
<td>6014.41</td>
<td>6049.29</td>
<td>2758.30</td>
<td>2797.53</td>
<td>2838.69</td>
</tr>
<tr>
<td>Ex. time (ms)</td>
<td>5976.34</td>
<td>5993.26</td>
<td>5988.02</td>
<td>2711.67</td>
<td>2712.34</td>
<td>2712.34</td>
</tr>
<tr>
<td>Norm. to Suff.</td>
<td>100.00%</td>
<td>91.90%</td>
<td>99.80%</td>
<td>100.00%</td>
<td>99.91%</td>
<td>99.45%</td>
</tr>
</tbody>
</table>

Fragments Scheduling. As previously described in Section 4, we conducted a set of experiments with scheduling heuristics in two heterogeneous machines. The objective of our scheduler is to find a fragment-to-device assignment that minimizes the total execution time (makespan). Table 2 shows, for each device in the two systems, the time spent to execute the number of assigned fragments (in square brackets) for the particular scheduling policy. The table also presents for each heuristic the makespan and the normalized result to the Sufferage heuristic.

In both systems Sufferage reaches an almost perfect load balancing between the three available devices, fully utilizing the entire machines.

In both systems the static scheduling heuristics obtain similar results, with Max-Min that reaches almost the same performance of Sufferage, while Min-Min shows 91.90% and 90.73% of the performance, respectively. These results are justified by the structure of the dataset. Usually, datasets collected with LIDAR technology contain few fragments with a big number of points and many small fragments with fewer points. Due to the fragment selection policy, Sufferage and Max-Min perform the assignment of the large fragments in early iterations resulting in a better load balancing between the devices. Differently, Min-Min favors the assignment of fragments with lower cost in early iterations, not reaching the same performance in terms of makespan. It is noteworthy that the three static scheduling heuristics use a perfectly correct estimated execution time for the fragments (ETC matrix) that will not be available at scheduling time. The resulting performance of the heuristics is therefore only useful as a comparison parameter for our low-complexity heuristics SimpleH and SimpleHS, which are only based on information available at scheduling time. SimpleH, based on the assumption that the GPUs devices are always faster than the CPU, dynamically assigns the fragments with more points to the GPUs and the one with fewer points to the CPU. This simple mechanism facilitates the devices load balancing by avoiding assigning large fragments to slow devices. Although SimpleH is capable of reaching good performance, it also shows its weakness with our input dataset. The heuristic does not take into account the number of remaining fragments to assign and, when few are left, continues to distribute them to the CPU. This behavior can lead to load imbalance if the GPUs have to wait for the CPU that received one of the last fragments. This issue is solved with the SimpleHS heuristic, previously described in Section 4. SimpleHS tries to predict the approximate execution time of a new fragment based on the execution time of the previous ones. Although at the beginning the prediction error is high, it rapidly decreases during the
scheduling of fragments. It is noteworthy that the overhead introduced by the prediction model is negligible and does not impact the performance of the scheduler. In our tests, SimpleHS is able to correctly predict when to stop the assignment of fragments to the CPU, obtaining a better load balancing between the devices. As depicted in Table 2, SimpleHS, scheduling less fragments to the CPU, always achieves better or equal performance compared to SimpleH, reaching 98.79% and 95.45% of the Sufferage performance in the two systems.

These results validate the success of the proposed heuristics which, using only information available at scheduling time, show comparable performance to more sophisticated methods which require an accurate estimation of the expected execution times.

7 Related Work

The study of the interaction of millions of points, present in modern datasets, requires scalable systems capable of supporting the large computational demands. In order to actually improve the scalability of such systems many spatial partitioning methods were proposed and investigated [2, 25, 26]. Some of these approaches are suitable for simulations which frequently have high density in one or several spatial locations and some perform best with uniformly distributed points. In recent years, among such methods, uniform grid data structures have received great attention from the research community. Erra et al. [8], leveraging the GPU processing power, implemented an efficient framework which permits to simulate the collective motion of high-density individual groups. Aaby et al. [1] presented the parallelization of agent-based model simulations (ABMS) with millions of agents on multiple GPUs and multi-core processors. Viguera et al. [24] proposed different parallelization strategies for the collision check procedure that takes place in agent-based simulations. Green [11] described how to implement a simple particle system in CUDA using a uniform grid data structure. Husselmann et al. [12] presented single- and multiple-GPU solutions for grid-boxing in multi-spatial-agent simulations. While the uniform grid approach of these works is similar to ours, they are restricted by the language of choice to some specific hardware. Differently, using OpenCL, our work is not limited to a single platform and can be executed in multiple heterogeneous devices. This advantage allows us to compare different platforms and fully exploit the computational performance of heterogeneous systems as shown in other recent work [15].

8 Conclusions

This paper proposes an OpenCL implementation of the second order tensor field computation of massive point datasets. Compared with an existing KD-Tree parallel approach which uses OpenMP, our approach is $24 \times$ faster on an Intel i7-2600K. Since OpenCL supports heterogeneous devices, we investigated the performance of our implementation on a set of heterogeneous architectures, showing a remarkable reduction of the execution time. Furthermore, aware of the heterogeneous computing trend, we investigated different scheduling policies on two heterogeneous machines. The obtained results validate the success of the proposed SimpleHS heuristic, which shows comparable performance to more complex static heuristics, only using information available at scheduling time. In the future, we plan to extend our work to distributed environment using the libWater library [10, 9].

9 Acknowledgment

This research has been supported by the FWF Austrian Science Fund as part of project I 1523 “Energy-Aware Autotuning for Scientific Applications” and by the FWF Doctoral School CIM Computational Interdisciplinary Modelling under contract W01227.
References