Published article: V. Mäkinen, T. Sarjakoski, J. Oksanen, and J. Westerholm, "A Multi-GPU Program for Uncertainty-Aware Drainage Basin Delineation: Scalability benchmarking with country-wide data sets", *IEEE Geoscience and Remote Sensing Magazine*, volume 4, issue 3, 2016, DOI: 10.1109/MGRS.2016.2561405

MULTI-GPU PROGRAM FOR UNCERTAINTY-AWARE DRAINAGE BASIN DELINEATION – SCALABILITY BENCHMARKING WITH COUNTRY-WIDE DATASETS

Ville Mäkinen[1], *Tapani Sarjakoski*[1], *Juha Oksanen*[1], *Jan Westerholm*[2]

[1]Finnish Geospatial Research Institute FGI,
National Land Survey of Finland,
Geodeetinrinne 2, FI-02430 Masala, Finland
ville.p.makinen@nls.fi, tapani.sarjakoski@nls.fi, juha.oksanen@nls.fi
[2]Åbo Akademi University, Faculty of Science and Engineering
FI-20014 Turku, Finland
jan.westerholm@abo.fi

ABSTRACT

Processing high-resolution digital elevation models (DEM) can be tedious due to the large size of the data. In uncertainty-aware drainage basin delineation, we apply a Monte Carlo simulation that further increases the processing demand by two to three orders of magnitude. Utilizing graphics processing units (GPU) can speed up the programs, but their on-chip RAM limits the size of DEMs that can be processed efficiently on one GPU. Here we present a parallel uncertainty-aware drainage basin delineation algorithm and a multi-node GPU CUDA implementation along with scalability benchmarking. All the computations are run on the GPUs, and the parallel processes communicate using Message Passing Interface (MPI) via the host CPUs. The implementation can utilize any number of nodes, with one or many GPUs per node. The performance and scalability of the program have been tested with a 10 m DEM covering 390905 km^2, the entire area of Finland. Performing the drainage basin delineation for the DEM with different numbers of GPUs shows nearly linear strong scalability.

Index Terms**—**Geospatial analysis, parallel computing, GPU, MPI

1. INTRODUCTION

In uncertainty-aware geospatial analysis, we compute not only the solution to a given problem, but also estimates of the uncertainty of the solution [1]-[4]. Determining the reliability of the analysis is important because in many cases decisions are made based on the result of an analysis that may have a significant economic impact or even affect human lives. For example, issuing storm warnings will let people prepare for approaching storms in time, but if the predictions are not reliable the false alerts render the warnings useless. When choosing a location for long-term storage for nuclear waste, one wants to make sure that a location the model predicts to be stable is not simply a random artefact that moves or disappears with the slightest change in the input data. Knowing the reliability of the borders of the drainage basins [5], [6] will help proper action to be taken e.g. in the case of accidents where toxic material spills onto the ground. In general, knowledge of the uncertainty of the result of an analysis indicates whether the result can be trusted or if more accurate data or another analysis method are required.

Although the foundation for uncertainty-aware geospatial analysis is rather well established [1], [4], it has received relatively little practical usage. This is partly because the analysis of uncertainty is computationally very demanding, for the implementations use Monte Carlo simulations in which the underlying analysis is repeated typically a thousand times, if not more [1]. It is evident that carrying out uncertainty-aware geospatial analysis with large datasets covering geographically extensive areas pushes computation facilities to their limits.

Large computing clusters are nowadays common, but programs and algorithms must be developed for parallel execution in order to harness the available resources efficiently. Unfortunately, the traditional software packages that users in the application field of geographic information systems (GIS) are used to do not benefit from powerful computing clusters as well as they could [7-10]. For example, in the GRASS GIS package only some of the functionality supports parallelism [11]. In this paper we have designed and implemented an uncertainty-aware drainage basin delineation program that utilizes multiple GPUs to speed up the calculations and to permit efficient processing of large digital elevation models that do not fit into the RAM of a regular workstation.

Some work has been reported where GPUs have been utilized to speed up some common analyses [12-18]. However, they are typically limited to one GPU. This work is continuation to the work reported in [18] where preliminary benchmark calculations of a drainage delineation program utilizing multiple GPUs were presented. We have identified and analysed the main bottlenecks of the implementation and developed the algorithms further.

In the following sections, we describe the principles on which the program is based to achieve good performance and scalability. For benchmarking, we use a country-wide digital elevation model covering 390905 km^2, the area of Finland, in 10 m resolution [19]. To our knowledge, this is the first time that uncertainty-aware geospatial analysis has been carried out for areas covering an entire country. In addition, this was done in a single run.

Based on the benchmarking, we demonstrate that the cost to compute uncertainty-aware drainage basin delineations for country-wide datasets has been reduced to a rather low level. We argue that we have reached a situation in which cost alone is not sufficient a reason to neglect the computation and presentation of uncertainty maps. These statements are based on and apply to the drainage basin delineation task. As will be discussed at the end, our implementation could be used as a framework for other, similar uncertainty-aware geospatial analysis tasks.

In our study, the motivation for fast, scalable computing solutions is based on the need to produce uncertainty maps and on the underlying Monte Carlo simulation, which is a computationally intensive task. The need for fast and scalable programs for geospatial analysis is, though, much more generic: high-resolution data are available in such volumes, velocities, and varieties that they deserve to be called big geospatial data. Efficient utilization of these data fundamentally depends on quick, on-demand computations, in order to be able to produce timely inputs for environmental decision-making processes. At the same time, multi-GPU computing clusters are increasingly being used for scientific and technical computing. In this respect, the presented work can serve as a high performance geocomputing demonstration on utilizing computing resources efficiently.

2. DRAINAGE BASIN DELINEATION ALGORITHM

We begin by describing the process of uncertainty-aware basin delineation and then outline the parallelization of the task to multiple GPUs.

2.1 Basic drainage basin delineation algorithm

The drainage basin delineation algorithm is presented in refs. [17], [18], [20], [21]. In short, the basic algorithm that does not take the uncertainty of the DEM into account reads the DEM and the stream data as the input, and provides the borders of the drainage basins as the output. The principal idea is to determine to which stream the surficial flow leads from each cell. The basic algorithm consists of the following parts, which are executed sequentially:

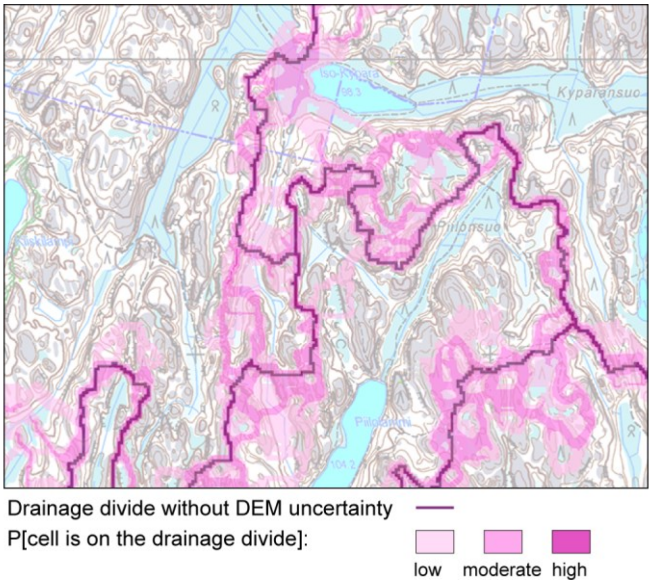

Figure 1: An example of the drainage basin borders determined with and without taking the uncertainty of the DEM data into account. (Background map: National Land Survey of Finland, Basic map raster, 01/2015)

1. Burn the stream data into the DEM.
2. Fill the pits in the DEM.
3. Assign flow directions to the cells.
4. Trace the cells to the streams.
5. Extract the borders of the drainage basins.

The stream burning is needed because otherwise some constructions, such as bridges, erroneously create obstacles for the surficial water flow in the DEM-based flow model. Small depressions in the DEM would stop the tracing of cells to the streams, therefore the pit filling is used to fill them, transforming them into flat areas. After this each cell is assigned a flow direction based on the slope of the DEM (the flat areas are handled separately). Finally, one can start from any cell and end up in a stream by following the flow directions. Knowing which stream each cell flows to makes it easy to determine the borders of the drainage basins.

2.2. Uncertainty-awareness

As all measured data contains some uncertainty, so does the DEM. The question that immediately arises is how much this uncertainty affects the locations of the acquired borders of the drainage basins. One way to take into account the uncertainties in the DEM height values is to run the drainage basin delineation program on the DEM several times, but each time with a different realization of the DEM error model added [4]. The realizations can be generated e.g. using process convolution [6], [22]. In a nutshell, the uncertainty-aware drainage basin delineation algorithm looks like this:

1. Generate an error field of random values.
2. Convolve the error field to reach an a priori specified spatial autocorrelation structure.
3. Add the error field to the original DEM.
4. Perform the basic drainage basin delineation algorithm for the DEM with the error field added.
5. Add the delineation borders to previous results.
6. Repeat steps 1–5 the number of times specified by the user (often in the range of 100–1000).

The results of each iteration are added cell-wise. After N iterations, the probability that the cell is on the drainage divide is the value of the cell divided by N. An example of a probable catchment border is shown in Figure 1.

The procedure is a straightforward Monte Carlo (MC) simulation, and the iterations are called MC iterations. The downside is that none of the calculations inside the MC iterations are reusable and the algorithm run time is proportional to the number of MC iterations.

2.3. GPU implementation

In our program, all the algorithms described in sections 2.1 and 2.2 are implemented as CUDA kernels [23]. Some of them (e.g. the random field generation) are easily implemented to benefit greatly from the fine-grained parallelism of the GPUs. If a thread, operating on one cell, requires the output from other threads, they need to be synchronized in order to avoid data races. This imposes limitations on the design of the algorithms due to the fact that separate thread blocks cannot be synchronized within the CUDA framework. The basic features of the CUDA implementations of the algorithms are explained in [17] where a drainage basin delineation program using a single GPU is reported. We used the implementation in [17] as our starting point and modified the algorithms for multi-GPU environments.

2.4. Parallelization using many GPUs

Incorporating multiple GPUs and using them in parallel is achieved by dividing the DEM into rectangular partitions (Figure 2). Each partition is extended by a region called halo zone that is used to hold copies of the values from the neighbouring partitions. In this way, large sections of the partitions can be processed independently of other partitions, and only the values at the boundary zones must be communicated to the halo zones of the neighbouring partitions. The most straightforward division method is to divide the DEM into partitions of the same size and assign one partition to each GPU, as shown in Figure 2.

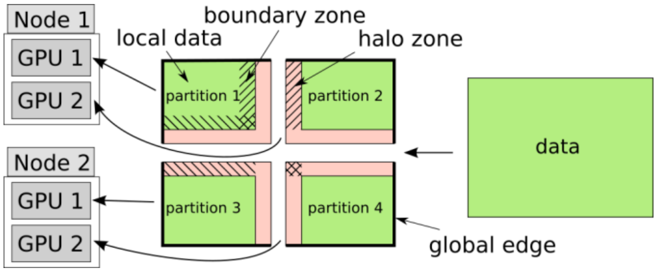

Figure 2: An example of dividing and assigning data in a multi-node, multi-GPU environment. The boundary zone is part of the local data and the partitions are extended with the halo zones. The striped areas in Partition 1 show how the boundary zone is distributed to the halo zones of Partitions 2, 3 and 4.

The drawback of this method is that as the data is split into smaller and smaller partitions, the ratio of the circumference of the partitions to their area grows. At some point, the overhead due to synchronization and MPI communication will become comparable to the actual execution time on the GPUs and thus will degrade the scalability of the program. When this happens exactly is highly dependent on the underlying hardware.

Another parallelization method would be to calculate several MC iterations concurrently. This would be trivial to implement because the individual MC iterations are independent of each other. However, this work concentrates on processing datasets that are so large that the memory of a single GPU is insufficient, thus requiring multi-GPU solutions.

3. MULTI-GPU PROGRAM FOR UNCERTAINTY-AWARE DRAINAGE BASIN DELINEATION

When the drainage basin delineation program is executed, the MPI processes allocate arrays of memory for the partitions of the DEM and the stream data and for the corresponding drainage areas. These arrays are kept in the GPU memory throughout the program execution. We note that they could be stored on the host RAM as well. In that case more GPU RAM would be available for the temporary data and the size of the partitions could be increased. The downside is that the relatively slow transfer of data between the host and the GPU RAM would be required for each MC iteration.

Referring to the computation steps described in sections 2.1 and 2.2, in each MC iteration, the random number generation, the convolution of the random field, the stream burning and the extraction of the borders of the

Published article: V. Mäkinen, T. Sarjakoski, J. Oksanen, and J. Westerholm, “A Multi-GPU Program for Uncertainty-Aware Drainage Basin Delineation: Scalability benchmarking with country-wide data sets”, *IEEE Geoscience and Remote Sensing Magazine*, volume 4, issue 3, 2016, DOI: 10.1109/MGRS.2016.2561405

drainage basins all work in a similar manner: first the width of the halo zones are chosen, then the local data is processed, and finally the halo zones are updated with the boundary values from the neighbours. For example, in the case of the random field generation, the width of the halo zones is the radius of the convolution filter reflecting the range of the DEM error model’s spatial autocorrelation range. Each cell needs to be processed only once and they can be processed in any order. The common factor for these algorithms is that when they are operating on a cell, they only need values from the neighbouring cells inside a predefined radius, which can be zero. In general, these kinds of algorithms can be implemented efficiently for parallel architectures. If the whole analysis consisted only of such operations, it would be possible to divide the DEM into small enough partitions and analyse them sequentially on a single GPU; however, due to the highly non-local nature of the pit filling, the flow routing of the flat areas and the flow tracing algorithms we are required to process the entire DEM simultaneously.

3.1. Parallel pit filling

A pit filling algorithm is needed because the input DEM with the random field added contains small depressions that will stop the tracing of the cells to the streams. The algorithm transforms these depressions into flat areas so that starting from any cell it is possible to reach a stream without going uphill.

Compared to the algorithms mentioned above, the pit filling algorithm is considerably more complex. Our implementation is based on the single GPU implementation introduced in [17], which starts by creating an auxiliary elevation data array where the cells in the streams are marked with zero elevation and others with infinity. The cells in the streams are marked as active. Then the pit filling CUDA kernels are launched to process the data. Each thread that has an active cell assigned to it marks it as inactive, then iterates over its neighboring cells and, when certain conditions are met, lowers their auxiliary elevation values and marks them as active. These kernels cannot finish the algorithm in one run, so they need to be launched again and again until none of the cells are marked as active [17]. Here, in the multi-GPU context, we refer to this process as performing local iterations until the algorithm has converged locally.

With multiple GPUs, the difference to the single GPU case is that after every local iteration the data in the boundary zones may have been updated and the halo zones need to be updated. The principal design of our multi-GPU algorithm is shown in Figure 3. It consists of global iterations in which the local iterations are first repeated at maximum N_{limit} times before updating the halo zones.

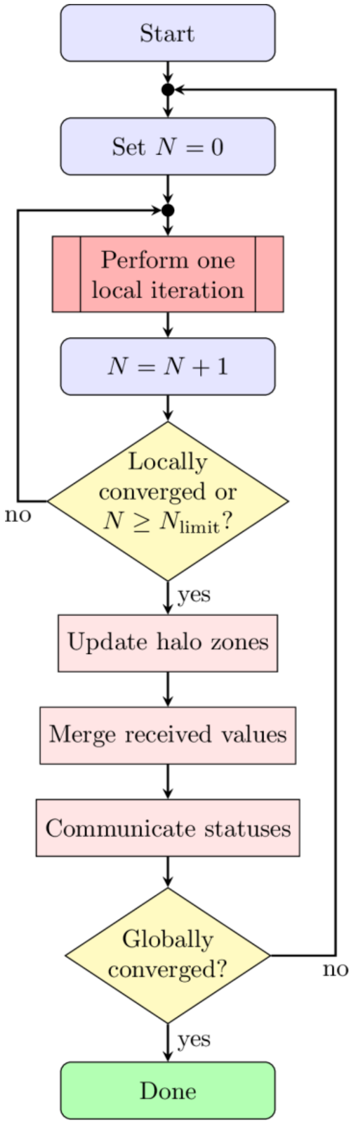

Figure 3: The design of the iterative pit filling and the flow routing of the flat areas algorithms. The value of N_{limit} is chosen based on the hardware used.

After receiving the data from the neighbours, the MPI processes need to evaluate whether they have active cells to process and report this information to all the other MPI processes. The global iterations are performed until all the MPI processes converge locally at the same time, i.e. until the algorithm converges globally.

Note that an MPI process is not allowed to exit from the algorithm after reaching local convergence. This is because it may remain in a locally converged state for several global iterations but then receive data from neighbouring partitions that forces it to do processing again.

Forcing the updating of the halo zones after a fixed number of local iterations, regardless of whether the

Published article: V. Mäkinen, T. Sarjakoski, J. Oksanen, and J. Westerholm, "A Multi-GPU Program for Uncertainty-Aware Drainage Basin Delineation: Scalability benchmarking with country-wide data sets", *IEEE Geoscience and Remote Sensing Magazine*, volume 4, issue 3, 2016, DOI: 10.1109/MGRS.2016.2561405

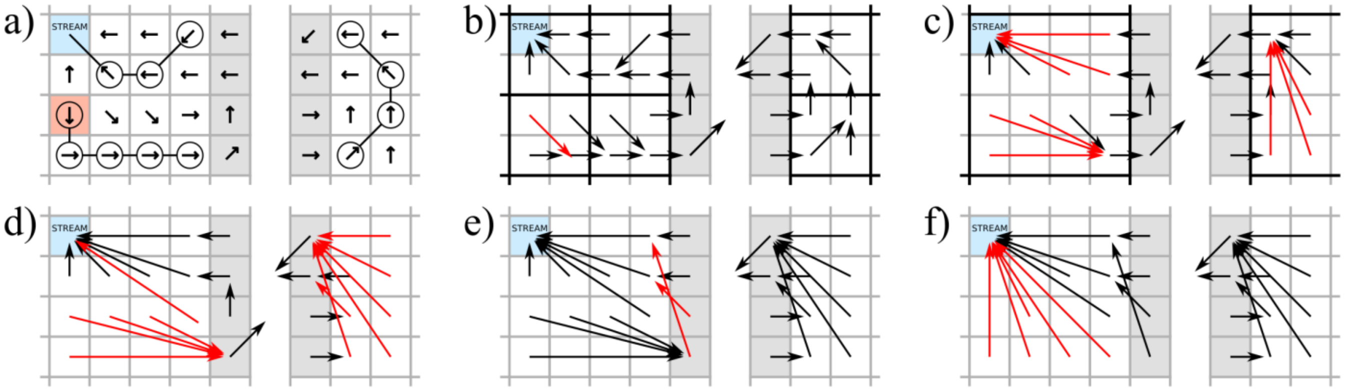

Figure 4: An example case where the flow tracing crosses the partition edges. The shaded columns mark the halo zones and the red arrows indicate a change from the previous configuration. In a) the real flow route is shown for a single cell. In b) the flow directions are reduced to directed links in 2×2 sub-areas, and subsequently in 4×4 areas in c). Subfigures d) and e) show the iterative part of the algorithm where the links are traced for each cell in the local area (d) and the suitable values in the halo zones are updated (e). In this example the algorithm finishes after one iteration and f) shows the final link configuration.

algorithm has converged locally or not, helps to avoid situations where a partition has crucial updated boundary data that its neighbour requires to advance in its processing but has to wait for the partition to reach a local convergence before starting the communication. The optimal value of N_{limit} depends on the speed of the connection between the nodes compared to the processing power of the GPUs. In our simulations we used $N_{\text{limit}} = 5$.

The pit filling algorithm works with multiple partitions because the new (initially infinite) elevation values are always lowered from the previous values and because of the design of the algorithm it is impossible to lower them too far down. Therefore, if some cells have been processed and then lower values are received from the neighbouring partitions, the cells will simply be reprocessed without the need to keep track of and undo the previous work.

3.2. Parallel flow routing

The flow routing for non-flat and flat areas is performed separately. For the non-flat areas, the flow direction is set to the direction of the steepest descent using the D8 method [24]. Then the halo zones are updated.

The nontrivial part is to assign flow directions to the cells that are located in the filled pits. As these pit areas are flat, the method mentioned above does not work. However, the pit filling algorithm we use guarantees that for every flat area, there is at least one cell that has its flow direction outward from the area: a spill point. We have chosen to assign the flow directions for the cells in the flat areas in such a way that each cell flows to a spill point along the shortest path within the flat area.

The flat-area flow direction algorithm starts by creating an integer array. The cells in flat areas next to the spill points are marked with one, then their neighbours with two and so on. As in the pit filling, if lower values are received from the neighbouring partitions, some cells need to be reprocessed. Once each flat cell has been assigned its final value the flow directions are set such that each cell flows to the closest neighbour with a lower value.

The design shown in Figure 3 is also used to implement the flow routing of the flat areas. If a flat area is split by the partition division, the number of required global iterations to reach global convergence increases.

3.3. Parallel flow tracing

With the help of the flow directions, the cells can be traced to the streams or to global edges. When a cell is traced to a stream, the cell is marked with the ID value of that stream. The output is a grid with a stream ID at each cell point indicating to which drainage basin the cell belongs.

The flow tracing algorithm presented in [17] is designed for a single partition and does not work optimally if the tracing of a cell leads to another partition. An example case is shown in Figure 4a where the tracing of the cell marked with red leads to another partition twice before reaching a stream cell. In the first iteration only the upper part of the left partition can be traced, in the second iteration only the right partition can be traced, and in the third iteration the rest of the left partition can be traced. The demerit of the approach is that unless the destinations of the tracings that do not reach a stream cell are recorded, e.g. the chain starting from the red cell must be traced to the halo zone three times. As the partition size grows both the number of the tracings and their chain lengths grow, leading to unnecessary work and a slower program.

In our approach we treat the flow directions as directed links between the cells. First we divide the local area into non-overlapping N × N sub-areas and reduce the flow directions into links inside the sub-areas as shown in Figure 4b. Then the sub-areas are quadrupled and the links are further reduced inside the larger sub-areas. This is continued until the whole local area has been reduced (Figure 4c). A natural choice for the initial sub-area size in the CUDA implementation is the size of the thread block.

Only after the reduction step we consider the halo zones and the neighbouring partitions. For each cell in the local area we trace the links until either

- we reach a stream cell, in which case the starting cell is linked to the found stream cell, or
- the tracing leads out of the partition, in which case the starting cell is linked to the last cell in the chain that is inside the partition.

This is depicted in Figure 4d. After this step the links in the boundary zone are communicated to the neighbouring partitions. Only the received links that point back to the partition and that are different from the existing links in the halo zone are updated (Figure 4e). This reduction and communication cycle is repeated until the merged links are the same as the links in the local data (Figure 4f).

After the reduction phase has converged the actual tracing is performed. The cells that flow to a stream cell in the same partition can be traced via a single link and are marked with the ID of the stream. Then the halo zones are updated and the cells without a stream ID are traced again. This is repeated until every cell has been traced to a stream or to a global edge.

3.4. Extracting the borders of the drainage basins

The borders of the drainage basins are extracted from the output of the flow tracing algorithm simply by marking all the cells that have a neighbour with a lower stream ID. The extracted borders are then added to the border array allocated at the beginning of program execution.

4. HIGH PERFORMANCE COMPUTING ENVIRONMENT FOR TESTING AND EVALUATION

The program is written in C++ with NVIDIA CUDA extensions [23]. MPI [25] is used for communication between the processes running on the CPUs on the separate nodes.

We are currently using the Bull supercomputer of CSC – IT Center for Science Ltd., a non-profit computing centre for universities and research institutes in Finland (www.csc.fi). The Bull is a cluster with 38 nodes that are connected by InfiniBand, each node furnished with two NVIDIA K40 cards [26]. A single K40 card has 12 GB of RAM, resulting in a total of 912 GB of GPU memory.

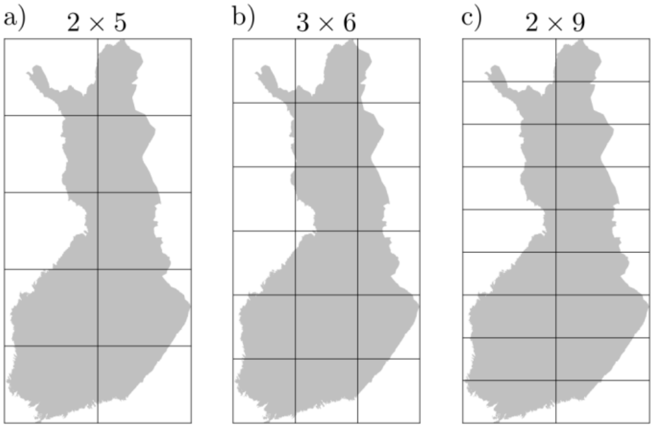

Figure 5: Examples of data partitioning to a different number of equivalent rectangular blocks. The outer box shows the area for which the analysis was performed and the grey the area for which elevation data exists. The subfigures b) and c) show two possibilities for dividing the data into 18 partitions.

4.1. Single and multi-threaded CPU implementation

For comparison we have also implemented CPU versions of the presented algorithms. The main difference to the GPU versions is that the pit filling and the flow routing of the flat areas are priority queue based rather than using a separate raster to keep track of the cells that are to be processed next. The algorithms are implemented only for a single thread execution so in order to parallelize the computation for N threads the area must be divided into N partitions. The partitions are processed in parallel using OpenMP and the communication between the nodes is handled via MPI.

The CPU program was benchmarked in Taito [27], another cluster available at CSC. The Taito cluster includes also "fat" computing nodes with large memory capacity. This allows us to make such reference computations that the whole test data resides on a singe node used e.g. for the fully serial CPU implementation. All the computations were performed on nodes with two Intel Haswell 12-core E5-2690v3 processors, running at 2.6GHz.

5. TIMINGS

For benchmarking, we used the country-wide DEM of the entire area of Finland, which is available in 10 m resolution [19]. For our purposes, a bounding box of 55,000 × 114,000 grid cells was needed to cover the whole of Finland (shown in Figure 5). With the current implementation, we need at least ten NVIDIA K40 GPUs to process the data efficiently, the GPU memory being the limiting factor. We used the drainage basin delineation with 50 MC iterations as the benchmark calculation. As the calculation environment consisted of nodes with two GPUs on each node, we

Published article: V. Mäkinen, T. Sarjakoski, J. Oksanen, and J. Westerholm, “A Multi-GPU Program for Uncertainty-Aware Drainage Basin Delineation: Scalability benchmarking with country-wide data sets”, *IEEE Geoscience and Remote Sensing Magazine*, volume 4, issue 3, 2016, DOI: 10.1109/MGRS.2016.2561405

benchmarked our program using up to 20 nodes (40 GPUs). This may be considered as a strong scaling test [28], [18].

A regular block of data can be divided into *p* partitions simply in row-wise or column-wise order, i.e. into *p* × 1 or 1 × *p* partitions. There are more possibilities if *p* is not a prime number. The optimal division depends on several factors. If the communication between the partitions is slow, minimizing the circumference of the partitions may result in the fastest execution of the program. However, one partition scheme may leave some partitions virtually empty and others full of data (Figure 5b), while another scheme can provide a more balanced solution (Figure 5c). A significant imbalance in workload leads to longer execution times as the GPUs assigned to the empty partitions are not actually calculating anything. The partition schemes used and the benchmark execution times are reported in Table 1.

To measure the scalability of the program, we need to compare some characteristic values. Comparing the total execution times is not ideal because they contain all the activities that are needed only once at the beginning and at the end of the analysis, including disk I/O, whose bandwidth may vary noticeably. Also, comparing the individual MC iterations is not meaningful because the random fields generated are different in each benchmark calculation. Therefore, we define the ideal execution time that we derive from the average MC iteration time to be used as the metric for the scalability of the program. We denote the average MC iteration time using *p* GPUs with $\left\langle T_p \right\rangle$ and the standard deviation with $\delta\left\langle T_p \right\rangle$. These are obtained from the log files of the benchmark calculations. With these quantities, we can define the ideal execution time of the analysis with *N* MC iterations using *p* GPUs as

$$T_p^N = N\left\langle T_p \right\rangle, \tag{1}$$

where *N* is the number of MC iterations used to calculate the average MC iteration time. Standard deviation of the ideal execution time is

$$\delta T_p^N = \sqrt{N}\,\delta\left\langle T_p \right\rangle. \tag{2}$$

Since the MC iterations do not depend on each other, we can join the standard deviations of the individual MC iterations quadratically [26] above in the formula (2).

The speedup for an analysis with *N* MC iterations is calculated from the ideal execution times using the equation

$$S_p^N = \frac{T_{10}^N}{T_p^N} = \frac{\left\langle T_{10} \right\rangle}{\left\langle T_p \right\rangle}. \tag{3}$$

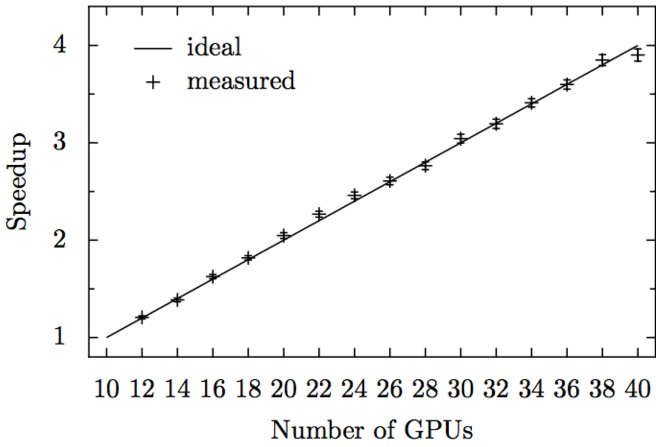

Figure 6: The speedup as a function of the number of GPUs for the analysis of the whole Finland, calculated from the benchmark calculations with 50 MC iterations. The error bars show the standard deviation for the derived speedups for the analyses with 49 MC iterations.

In an ideal case $S_p^N = p/10$, since we here use ten GPUs as our reference case. The fluctuations in the individual MC iteration times will induce variations in the speedup achieved as well. We can estimate this variation by applying the general formula for error propagation [29] to the S_p^N. This gives the standard deviation

$$\begin{aligned}\delta S_p^N &= \sqrt{\left(\frac{\partial S_p^N}{\partial T_{10}^N}\delta T_{10}^N\right)^2 + \left(\frac{\partial S_p^N}{\partial T_p^N}\delta T_p^N\right)^2} \\ &= \frac{1}{\sqrt{N}}\sqrt{\left(\frac{\delta\left\langle T_{10}\right\rangle}{\left\langle T_p\right\rangle}\right)^2 + \left(\frac{\left\langle T_{10}\right\rangle}{\left\langle T_p\right\rangle^2}\delta\left\langle T_p\right\rangle\right)^2}.\end{aligned} \tag{4}$$

These formulae show that the achieved speedup can vary considerably with small numbers of MC iterations, but as the number of iterations increases, the fluctuations in the individual MC iterations average out.

Another commonly used quantity is the efficiency E_p^N, which in this case is defined as

$$E_p^N = S_p^N \cdot \frac{10}{p} = \frac{10 \cdot \left\langle T_{10}\right\rangle}{p \cdot \left\langle T_p\right\rangle}. \tag{5}$$

A value close to one means that the scaling is efficient, whereas a value closer to zero means inefficient scaling. The standard deviation for the efficiency is given simply by

$$\delta E_p^N = \frac{10}{p}\delta S_p^N. \qquad (6)$$

We excluded the first MC iteration and calculated the average MC iteration time from the subsequent 49 iterations of the benchmark calculations, as in the first MC iteration the algorithms need to perform some initialization. The average MC iteration times, their standard deviations and the derived speedups and efficiencies are reported in Table 1, and the derived speedup results are shown in Figure 6. The scaling of the program is very close to ideal. One source of variation is that the amount of imbalance in the workload varies slightly with the number of GPUs used.

For comparison we performed 10 MC iterations using the CPU version of the program first in a fully serial mode and then parallelized over 2, 10, 24 and 48 threads. The timings results are shown in Table 2. Again, the first MC iteration was excluded from the calculation of the average values. The values are calculated using the equations (3) - (6) but using the case $p=1$ as the reference case. The CPU implementation has not been optimized to the same extent as its multi-GPU counterpart and it is possible that adjusting the parameters such as the N_{limit} shown in Figure 3 more carefully could improve the scaling.

A direct comparison of the average MC iteration times indicates that the multi-GPU program using e.g. 10 GPUs is ~100 times faster than the serial CPU version. In [17] the single GPU program was found to be roughly 10 times faster than the serial CPU program. Therefore the comparison of multi-GPU program using 10 GPUs is expected to be two orders of magnitude faster than the fully serial CPU version, and our measurements fit into this expectation well.

The early benchmarkings of our multi-GPU program were reported in [18]. At that stage, the scalability for multiple computing nodes was not ideal. In the current work we have shown good, nearly linear scalability. Based on the benchmarking results, we can estimate that using ten GPUs, uncertainty-aware computation of a drainage basin delineation, based on 1000 MC iterations would take about 12.6 h for the whole of Finland. With 40 GPUs, the computing time is less than 3.3 h.

According to CSC's pricing for academic and public sector [30], GPU cost is 0,30 €/h. The cost for the job used as a reference above (12.6 h on 10 GPUs) would be 38€. To run the same job using our single core CPU implementation would take 1640 h and the cost would be 36€, based on CSC's 0.022 €/h price for CPU core usage. For GPU implementation cost is invariant with respect to the number of GPUs, because the efficiently is always close to 1.0 (Table 1), whereas for CPU implementation the cost increases when more cores are used, due to the decreasing efficiency (Table 2).

6. CONCLUSIONS

In this work we have introduced improved methods for reaching a better scalability in our uncertainty aware drainage basin delineation program running on multiple GPUs, reported in [18]. The test runs now show linear scalability with respect to the number of GPUs used. In this work we also compared the program with the reference implementation using CPUs only. These tests confirmed our expectations that use of GPUs speeds up processing at least ten times compared to a single core CPU implementation.

Our comparison of the costs for running jobs either in

Table 1: The partitioning scheme, total execution times, average Monte Carlo iteration times $\langle T_p \rangle$, their standard deviations $\delta\langle T_p \rangle$ and derived speedup S_p^{49} and efficiency E_p^{49} values with their uncertainties δS_p^{49}, δE_p^{49}, obtained from the 49 MC iterations of the benchmark calculations with 50 MC iterations using *p* GPUs.

p	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40
Partitioning	2×5	2×6	2×7	2×8	2×9	2×10	2×11	2×12	2×13	2×14	2×15	2×16	2×17	2×18	2×19	2×20
Total time [s]	2312	1899	1666	1415	1273	1124	1019	936	885	834	755	721	673	640	600	589
$\langle T_p \rangle$ [s]	45.6	37.8	32.9	28.1	25.1	22.3	20.1	18.5	17.5	16.5	15.0	14.3	13.4	12.7	11.9	11.7
$\delta\langle T_p \rangle$ [s]	3.2	2.2	2.3	1.7	1.3	1.7	1.2	1.3	1.4	1.2	1.1	1.1	0.8	0.8	0.9	1.0
S_p^{49}	-	1.21	1.39	1.63	1.82	2.05	2.27	2.46	2.61	2.76	3.04	3.19	3.41	3.60	3.85	3.90
δS_p^{49}	-	0.02	0.02	0.02	0.02	0.03	0.03	0.03	0.04	0.04	0.04	0.05	0.04	0.05	0.06	0.06
E_p^{49}	-	1.01	0.99	1.02	1.01	1.02	1.03	1.03	1.00	0.99	1.01	1.00	1.00	1.00	1.01	0.98
δE_p^{49}	-	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.01	0.01	0.02	0.01	0.01	0.01	0.02

GPU or CPU environments show that neither of the environments offers any significant advantage in this sense. From the point of view practical analysis tasks, we consider the price tag of about 38 € for our reference job to be a reasonable cost for tasks that have to be carried out only occasionally. We also argue that we have reached a situation in which the cost alone should not be a reason for neglecting the computation and presentation of uncertainty maps.

Our benchmarks on scalability with a larger number of GPUs indicate that our implementation would be able to handle much larger datasets than the current 10 m resolution DEM covering the whole of Finland. A larger capacity will be needed in the future when high-resolution DEMs will be available; according to plans, the NLS of Finland will have a new laser scanning-based DEM with 2 m resolution around 2020.

Regarding our implementation, there is still room for improvement. As Figure 5 shows, the workload is not balanced between the GPUs as some partitions have very little data to process compared to some other partitions. Depending on the shape of the computing block's outline, the situation can be even worse. More sophisticated methods for data partitioning could be developed to improve the situation in this respect.

In this work, the focus has been on performing drainage basin delineation for large datasets efficiently using high performance computing environments. However, the ability to perform the same analysis interactively for small areas is also important. Current GPUs have so much computational resources that if the area to be processed is small, a big part of those resources may be left unused. For interactive use, performing several MC iterations in parallel may provide additional speedup and bring interactive uncertainty-aware geospatial analysis closer to reality.

Regarding future work, our program can also serve as a model or framework for the implementation of programs for other, similar uncertainty-aware geospatial analysis tasks. Using Map Algebra terms, random number generation is a local operation, whereas convolution, stream burning, flow routing of the non-flat areas and border extraction are focal operations with different neighbourhoods [31]. Pit filling, flow routing of the flat areas and flow tracing resemble global cumulative functions [32]. Therefore, it can be foreseen that a computationally efficient and scalable uncertainty-aware Map Algebra program could be implemented using our algorithms as a starting point.

7. ACKNOWLEDGEMENTS

Funding from the Academy of Finland (grants 251987, 259557 and 259995) is gratefully acknowledged. In addition, we are thankful to MSc David Eränen for the initial implementation of several parts of the program.

Table 2: Timing results of the CPU implementation of the drainage delineation algorithm, obtained from the computation of 10 Monte Carlo iteration using p threads and partitions. The case p=48 was computed using two nodes, other cases were computed in a single node. The average values are calculated from the last nine MC iterations.

p	1	2	10	24	48
Partitioning	1	1×2	2×5	2×12	2×24
Total time [s]	59506	34380	8973	4582	2426.5
$\langle T_p \rangle$ [s]	5904.9	3399.6	867.4	423.4	225.8
$\delta\langle T_p \rangle$ [s]	49.9	42.0	13.5	6.8	1.7
S_p^9	-	1.74	6.81	13.95	26.2
δS_p^9	-	0.01	0.05	0.09	0.10
E_p^9	-	0.87	0.68	0.58	0.54
δE_p^9	-	0.01	0.01	0.01	0.01

8. REFERENCES

[1] S. Openshaw, M. Charlton, and S. Carver, "Error propagation: A Monte Carlo simulation", In: I. Masser, M. Blakemore, (Eds.), *Handling Geographical Information: Methodology and Potential Applications*, Longman, London, UK, 1991, pp. 78-101.

[2] G. Heuvelink and P. Burrough, "Propagation of errors in spatial modelling with GIS," *International Journal of Geographical Information Systems*, vol. 3, no. 4, pp. 303–322, 1989.

[3] P. Fisher, "First experiments in viewshed uncertainty: The accuracy of the viewshed area," *Photogrammetric Engineering and Remote Sensing*, vol. 57, no. 10, pp. 1321-1327, 1991.

[4] G. Heuvelink, *Error Propagation in Environmental Modelling with GIS*, Taylor & Francis, London, UK, 1998.

[5] D. R. Miller and J. G. Morrice, "Assessing uncertainty in catchment boundary delimitation." Third International Conference/Workshop on Integrating GIS and Environmental Modeling, Santa Fe, NM, 21–26 January 1996. Available online at: http://escholarship.org/uc/item/43x094z3. Accessed on: Feb. 26, 2016.

[6] J. Oksanen and T. Sarjakoski, "Error propagation analysis of DEM-based drainage basin delineation," *International Journal of Remote Sensing*, vol. 26, no. 14, pp. 3085-3102, 2005, doi:10.1080/01431160500057947

[7] ArcGIS, http://www.esri.com/software/arcgis. Accessed on: March 8, 2016.

[8] QGIS, http://www.qgis.org. Accessed on: March 8, 2016.

[9] Manifold GIS, http://www.manifold.net. Accessed on: March 8, 2016.

[10] GRASS GIS, http://grass.osgeo.org/#. Accessed on: Feb. 4, 2016.

[11] GRASS GIS wiki on parallelization: http://grasswiki.osgeo.org/wiki/Category:Parallelization. Accessed on: June 17, 2015.

[12] L. Ortega and A. Rueda, “Parallel drainage network computation on CUDA,” *Computers & Geosciences*, vol. 36, pp. 171-178, 2010, doi:10.1016/j.cageo.2009.07.005

[13] C.-Z. Qin and L. Zhan, “Parallelizing flow-accumulation calculations on graphics processing units – From iterative DEM preprocessing algorithm to recursive multiple-flow-direction algorithm,” *Computers & Geosciences*, vol. 43, pp. 7-16, 2012, doi:10.1016/j.cageo.2012.02.022

[14] M. Steinbach and R. Hemmerling, “Accelerating batch processing of spatial raster analysis using GPU,” *Computers & Geosciences*, vol. 45, pp. 212-220, 2012, doi:10.1016/j.cageo.2011.11.012

[15] A. Osterman, L. Benedicic and P. Ritosa, “An IO-efficient parallel implementation of an R2 viewshed algorithm for large terrain maps on a CUDA GPU,” *International Journal of Geographical Information Science*, vol. 28, pp. 2304-2327, 2014, doi:10.1080/13658816.2014.918319

[16] J. Zhang, S. You and L. Gruenwald, “Large-scale spatial data processing on GPUs and GPU-accelerated clusters,” *SIGSPATIAL Special*, vol. 6, pp. 27-34, 2014, doi:10.1145/2766196.2766201

[17] D. Eränen, J. Oksanen, J. Westerholm, and T. Sarjakoski, “A full graphics processing unit implementation of uncertainty-aware drainage basin delineation,” *Computers & Geosciences*, vol. 73, pp. 48-60, 2014, doi:10.1016/j.cageo.2014.08.012

[18] V. Mäkinen, T. Sarjakoski, J. Oksanen, and J. Westerholm, “Scalable uncertainty-aware drainage basin delineation program using digital elevation models in multi-node GPU environments,” Proceedings of the 2014 conference on Big Data from Space, pp. 267-270, doi:10.2788/1823

[19] Elevation model 10 m, National Land Survey of Finland, http://www.maanmittauslaitos.fi/en/digituotteet/elevation-model-10-m. Accessed on: June 17, 2015.

[20] S. K. Jenson and J. O. Domingue, “Extracting topographic structure from digital elevation data for geographic information system analysis,” *Photogrammetric Engineering and Remote Sensing*, vol. 54, pp. 1593–1600, 1988.

[21] D. R. Maidment and W. K. Saunders, *A GIS Assessment of Nonpoint Source Pollution in the San Antonio - Nueces Coastal Basin*, Center for Research in Water Resources, University of Texas, Austin, TX. 1996.

[22] H. J. Thiébaux and M. A. Pedder, *Spatial Objective Analysis with Applications in Atmospheric Science*, Academic Press, London, UK, 1987.

[23] NVIDIA CUDA Parallel Computing Platform, http://www.nvidia.com/object/cuda_home_new.html. Accessed on: March 4, 2016.

[24] J. F. O’Callaghan and D. M. Mark, “The extraction of drainage networks from digital elevation data,” *Computer Vision, Graphics and Image Processing*, vol. 28, pp. 323-344, 1984, doi:10.1016/S0734-189X(84)80011-0

[25] Message Passing Interface (MPI) forum, http://www.mpi-forum.org. Accessed on: Feb. 4, 2016.

[26] CSC - Bull B715 cluster, https://research.csc.fi/csc-s-servers#bull. Accessed on: Feb. 4, 2016.

[27] CSC – Taito supercluster, https://research.csc.fi/csc-s-servers#taito. Accessed on: Feb. 4, 2016.

[28] Sharcnet: Measuring Parallel Scaling Performance, https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance. Accessed on: Feb. 4, 2016.

[29] John R. Taylor, *An Introduction to Error Analysis*, University Science Books, 55D Gate Five Road, Sausalito, California, 1982.

[30] CSC, Pricing of Computing Services, https://research.csc.fi/pricing-of-computing-services. Accessed on: March 10, 2016.

[31] C. Dana Tomlin, *GIS and Cartographic Modeling*, Esri Press, 380 New York Street, Redlands, California, 1990.

[32] M. N. DeMers, *GIS modeling in Raster*, John Wiley & Sons, Inc., 605 Third Avenue, New York, 2002.