Spatial interpolation in massively parallel computing environments

Katharina Henneböhl, Marius Appel, Edzer Pebesma

Institute for Geoinformatics, University of Muenster
Weseler Str. 253, 48151 Münster
{katharina.henneboehl, marius.appel, edzer.pebesma}@uni-muenster.de

ABSTRACT

Prediction of environmental phenomena at non-observed locations is a fundamental task in geographic information science. Often, samples are taken at a limited number of sensor locations and spatial and spatio-temporal interpolation is used to generate continuous maps. The computational cost of the underlying algorithms usually grows with the number of data entering the interpolation and the number of locations for which interpolated values are needed. Thus, real-time provision and processing of large spatio-temporal datasets call for scalable computing solutions. This requires re-thinking of established (sequential) programming paradigms. In this paper, we study the implementation and behavior of inverse distance weighted interpolation (IDW) on a single graphics processing unit (GPU), as an example for spatial interpolation algorithms in massively parallel computing environments. We argue that the underlying ideas can be expanded to a framework providing highly parallel functions for geostatistics.

1. INTRODUCTION

Spatial interpolation is a fundamental task in geographic information science: a limited number of data points are used to predict unknown quantities of a continuous phenomenon for a small or large number of prediction points. The underlying deterministic and/or geostatistical interpolation algorithms are well studied and optimized for application in geographic information systems using today's state-of-the-art hardware. However, real-time provision of large spatio-temporal datasets, the availability of massively parallel hardware such as programmable graphics processing units (GPUs) and the resulting specification of programming models such as CUDA (NVIDIA, 2008) and OpenCL (Munshi, 2010) call for re-thinking established sequential programming paradigms not only in spatial interpolation algorithms but geostatistical mapping and analysis tools in a wider sense.

Since massively parallel processors have been enabled for general purpose computing tasks, GPU computing is part of a general high performance computing (HPC) framework. Yang et al. (2010) consider HPC an important component of a Geospatial Cyberinfrastructure (GCI). One advantage of GPUs is that they provide the possibility to accelerate certain types of computations on virtually each desktop computer. Zhang (2010) discuss some perspectives of GPU technologies for parallel processing of geospatial data in a personal computing environment and introduce the term “Personal HPC-G” (Personal High Performance Geospatial Computing) for this trend. This paper approaches the prospect of GPU computing in geographic information science from a more empirical standpoint and contributes a study on the behavior of spatial interpolation algorithms in massively parallel environments. Our goal is to outline some chances for GPU accelerated tools, particularly for geostatistical analysis.

Research on GPU accelerated geostatistics has been carried out by Srinivasan et al. (2010a) who have re-formulated the classic (ordinary) kriging algorithm in order to benefit from a GPU’s massively parallel computing capabilities. Their formulation relates kriging to training and prediction in Gaussian process modeling (Rasmussen & Williams, 2006). Independently from our work, Huraj et al. (2010) have deployed inverse distance weighted interpolation (IDW) on GPUs to accelerate snow cover depth prediction. In this paper, we study the behavior of IDW on a single GPU depending on the number of data values, the number of prediction locations, and different ratios of data size and prediction locations.

The remainder of this paper is organized as follows. Section 2 describes IDW and its relation to the re-formulated kriging algorithm (Srinivasan et al., 2010a). Section 3 introduces technical properties of GPUs and further details of the IDW implementation. The performance of the IDW implementation is evaluated and discussed in section 4. Finally section 5 concludes with relevant findings.

2. SPATIAL INTERPOLATION

2.1 Inverse distance weighted interpolation (IDW)

Spatially interpolated values at non-sampled locations are computed as a weighted linear combination of a given number data values z_i for $i = 1,\ldots,n$. Usually, the weight w_{ji} of each data value is a function of $d(s_j, s_i)$, the Euclidean distance between a data location s_i and a prediction location s_j with $j=1,\ldots,m$ denoting the spatial index of the prediction locations. For IDW, w_{ji} are given by

$$w_{ji} = 1 / d(s_j, s_i)^p \tag{1}$$

where p denotes the IDW power. A common choice is $p=2$. The traditional sequential formulation to obtain a prediction $\hat{z}_j$ over a set of locations s_j is:

$$\hat{z}_j = \sum_{i=1}^{n} \hat{w}_{ji} z_i \quad \text{with} \quad \hat{w}_{ji} = \frac{w_{ji}}{\sum_{i=1}^{n} w_{ji}} \tag{2}$$

Alternatively, we can collect the normalized weights $\hat{w}_{ji}$ in a matrix W of dimension $m \times n$ where each row j represents a prediction point and each column i a data value z_i. Thus, an item $\hat{w}_{ji}$ of W is equivalent to the weight of the i-th data value z_i to predict the j-th point. IDW and other common spatial interpolators such as kriging (see section 2.2) can then be written as a simple matrix-vector product:

$$\hat{Z} = Wz \tag{3}$$

where $\hat{Z}$ is the $m \times 1$ vector of predictions at s_j and z the $n \times 1$ vector of data values z_i measured at s_i. In case of IDW, the rows of W sum up to 1.

Three operations need to be considered with regard to the GPU implementation described in section 3. First, we construct the spatial weight matrix W, then sum up the weighted data values, and finally normalize the weights. The implementation is straightforward. The resulting algorithm has complexity $O(nm)$.

2.2 Kriging

Although we focus on studying IDW in this paper we will briefly discuss the kriging interpolator to illustrate that generic strategies for parallelization apply to a wider class of problems. The kriging interpolator also computes the prediction as a weighted linear combination of data values. However, a GPU implementation of kriging requires more complex considerations than a GPU implementation of IDW.

Kriging interpolation is also known as Gaussian process prediction (Rasmussen & Williams, 2006) in the research field of machine learning. Srinivasan et al. (2010a) explicitly describe how to transform the traditional sequential notation for ordinary kriging into Gaussian process prediction notation given by:

$$\hat{Z}_K = C\hat{C}^{-1}z \tag{4}$$

where $\hat{Z}_K$ denotes the $m \times 1$ vector of kriging predictions, C the $m \times n$ covariance matrix between the prediction and data points. $\hat{C}$ represents the $n \times n$ variance-covariance matrix of the data values z_i. An item c_{ji} in C refers to the covariance between prediction location s_j and data location s_i. The construction of C and $\hat{C}$ requires a valid covariance model.

Roughly speaking, the covariances in C "replace" the IDW weights in W because kriging accounts for spatial correlation. In order to identify the matrix-vector multiplication part in the kriging algorithm, it is necessary to solve the system $\hat{C}q=z$ for q. Substituting $\hat{C}^{-1}z$ by q clarifies how kriging prediction can be traced back to simple matrix-vector multiplication. The computational cost of solving an $n \times n$ linear system has complexity $O(n^3)$. Thus, the overall computational complexity for kriging prediction is $O(n^3+nm)$, as opposed to $O(nm)$ in case of IDW. For a more comprehensive discussion

including formulas to obtain the variance associated with a kriging prediction in this form we refer the reader to Srinivasan et al. (2010a) and Rasmussen & Williams (2006).

From the above, we could conclude that compared to IDW kriging also comprises three main operations. That is the construction of the covariance matrix *C*, the solution of an *n x n* linear system and the summation of the weighted data values. Indeed, we have to add the step of finding and fitting a suitable covariance model prior to the construction of the matrix *C*. Thus, the approach described above accelerates kriging prediction but not kriging parameter estimation.

3. TECHNICAL SET-UP AND IMPLEMENTATION

3.1 GPU Basics

Our experiments aim at taking advantage of massively parallel computing capabilities as provided by today's programmable GPUs. At least, the formulation of spatial interpolation in terms of matrix-vector products predestines the prediction part, i.e. the summation of weighted data values, to be executed on the GPU (see section 2). The architecture of a modern GPU follows the single-program-multiple-data (SPMD) paradigm (Owens et al., 2008). It is particularly suited for data parallel applications with high arithmetic intensity, i.e. large number of numeric operations per memory access. Due to its specific properties, the GPU can support the CPU in certain computations rather than substitute it.

As we extend existing CUDA (NVIDIA, 2008) source code, we briefly discuss the CUDA device memory model in order to describe how the GPU architecture appears from a user's point of view and to illustrate how performance tuning works. CUDA distinguishes between the CPU (the host) and the GPU (the device). The host governs the computation and organizes the data transfer from and to the GPU.

The GPU is seen as a grid consisting of blocks. Each block houses a number of threads that perform the computations. Each thread can read from and write to a local register. All threads within a block have access to a shared memory and all blocks share the global memory. The maximum number of threads within a block depends on the computing capability of the GPU as does the (limited) capacity of the shared memory. Among the different kinds of memories on the GPU, global memory has the largest capacity but also the slowest access time. However, regular access patterns, i.e. each thread within a block reads and writes to consecutive global memory locations, reduce the effort (memory coalescing). Thus, optimization of algorithms for the GPU has two main targets:

1. Define regular patterns to control global memory accesses.
2. Maximize usage of local registers and shared memory.

All our experiments (section 4) were performed on a Quadro FX 4800 NVIDIA GPU with 1.5 GB global memory and an Intel Xeon Quad-Core 2.67 GHz machine with 6 GB RAM and an Ubuntu linux operating system version 10.04.

3.2 IDW implementation on the GPU

Our IDW implementation builds upon the generic approach for kernel summation of the open source GPUML package (Srinivasan et al., 2010b). This illustrates that overarching programming principles for massively computation algorithms are applicable for a wider class of spatial interpolation algorithms. We extended the GPUML package for IDW and for the joint interpolation of several datasets with one GPU call (Henneböhl et al., 2011). According to Srinivasan et al. (2010b)[1] the major steps of the CUDA coded IDW algorithm can be summarized as follows:

1. Each thread within a GPU block is assigned to compute the interpolated value for one prediction location.

[1] See also: http://www.umiacs.umd.edu/~balajiv/GPUML.htm

2. To reduce accesses to global memory, the coordinates and data values are loaded into shared memory. Each thread reads the information of one data point into shared memory thus taking advantage of coalesced memory access.
3. If the number of data points exceeds the capacity of shared memory, which is likely to occur, the data is divided into subsets and processed sequentially.
4. Each thread computes the IDW weights for the data points currently present in shared memory on-the-fly and updates the sum of weighted data values and the sum of weights.
5. Once all subsets of data points are processed, the resulting weighted sum is divided by the sum of weights (normalization) and written to global memory.

Computing the IDW weights on-the-fly in step 4 means that matrix W in equation (3) is not explicitly transferred from the host to the device, thus reducing the amount of data transfer between both and avoiding storage problems on the GPU. This is possible due to the functional representation of the $\hat{w}_{ji.}$ This mechanism applies to all kinds of matrices whose entries are given by a function, e.g. the covariance matrix between the data and prediction locations C (equation 4) in case of kriging.

4. EVALUATION

4.1 Performance tests

The computational effort for spatial interpolation grows with data size and number of prediction locations. Dimensionality of the data may also influence the performance but geospatial data usually does not exceed four dimensions (3d space and 1d time). Thus, we disregard the influence of dimensionality in the performance tests and focus on data size and the number of prediction locations. As for accuracy of the interpolation results in all settings, we use single precision on the GPU and double precision on the CPU. How much this affects performance depends on the hardware equipment. In general, single precision may favor the GPU in performance, i.e. computation time, comparison as a GPU's theoretical single precision capability usually is higher than its double precision capability. We believe that our conclusions are still reliable as for instance NVIDIA's new Fermi architecture closes the theoretical capability gap significantly: Fermi provides 256 FMA ops/clock[2] double precision floating point capability as opposed to 512 FMA ops/clock[2] for single precision (NVIDIA, 2009).

All test inputs were simulated random data assuming two data dimensions, i.e. coordinates in 2d space, and an IDW power of 2. Table 1 and 2 compare the computation times of IDW on the GPU and CPU. IDW interpolation for a single CPU was implemented in C++. In table 1, the number of prediction locations is held constant and data size varies while the opposite applies to table 2. Both tables indicate that significant speed-up factors can be achieved although the absolute values are specific for our experimental set-up.

Data size	Computation time sec CPU	Computation time sec GPU	Speed-up factor
10^2	0,005	0,004	1,4
10^3	0,052	0,006	8,4
10^4	0,513	0,033	15,6
10^5	5,147	0,301	17,1
10^6	53,18	2,974	17,9

Table 1: IDW computation times in sec averaged over 25 iterations on CPU and GPU depending on data size. The number of prediction locations was held constant at 10^4. All measured computation times are specific for the test setting and hardware equipment. The computation times include the time needed for data transfer from and to the GPU.

[2] Fused multiply-add operations per single clock

Number of prediction locations	Computation time sec CPU	Computation time sec GPU	Speed-up factor
10^3	0,005	0,005	1,2
10^4	0,052	0,006	8,2
10^5	0,521	0,030	17,5
10^6	5,154	0,25	20,6

Table 2: IDW computation times in sec averaged over 25 iterations on CPU and GPU depending on the number of prediction locations. The data size was held constant at 10^3. All measured computation times are specific for the test setting and hardware equipment. The computation times include the time needed for data transfer from and to the GPU.

4.2 Ratio between data size and number of prediction locations

The performance tests in this section look at the interaction between data size and number of prediction locations. We compare GPU accelerated IDW to its CPU implementation for different ratios *n/m* of data size and number of prediction locations. For each trial, the number of entries (i.e. *m x n*) in the spatial weights matrix was kept constant while the ratio *n/m* varied.

For *n>m* the interpolation is dominated by the amount of data. In the extreme case, a large number of data values would be used to predict one point and the spatial weights matrix would degenerate to a *1 x n* vector. A realistic scenario for *n>>m* would be the prediction for multipoint geometries. The case *n<<m*, i.e., *n<m* is likely to occur when interpolation aims at very high spatial resolutions. However, a ratio of *n/m=1/100* may represent a realistic scenario when mapping from monitoring networks, e.g. based on 100 data values and targeting 10000 prediction locations. A more balanced ratio may be the case for spatio-temporal interpolation taking into account several temporal instances of measured data.

In figure 1, the GPU computation time averaged over 25 iterations is plotted against different ratios *n/m*. The individual curves represent the behavior for a given number of entries in the spatial weights matrix. Figure 2 shows the same for the CPU implementation of IDW. Note that all measured computation times are specific for our test setting and hardware equipment. Generalization requires further investigation.

For *n<m,* CPU computation time exceeded GPU computation time in our test setting when the overall problem size *m x n* became large enough which in our example was 10^6, 10^7 *and* $5*10^7$. On the GPU, we observed a considerable growth in computation time for *n>m* and a slight growth for *n<<m.* As expected, the computation time of the single CPU implementation of IDW varied very little around a given level.

The growth of computation time for *n>m* on the GPU can be explained by the structure of the IDW implementation. Data dominated interpolation pronounces the use of shared memory. When the data size exceeds the shared memory capacity the summation of the weighted data values needs to be processed in chunks. Repeated thread synchronization within the blocks and global memory accesses then enhance the computation time. For *n>>m*, in our test setting the CPU at some point overtook our GPU, given that the overall problem size was constant. Prediction dominated interpolation pronounces parallelization via threads. For *m>>n,* expensive global memory accesses increase while the usage of the fast shared memory decreases. This may explain the slight growth in computation on the GPU for such cases.

The main difference between IDW on the GPU and CPU is that IDW on the GPU exhibits a performance peak depending on the ratio between data size and the number of prediction locations.

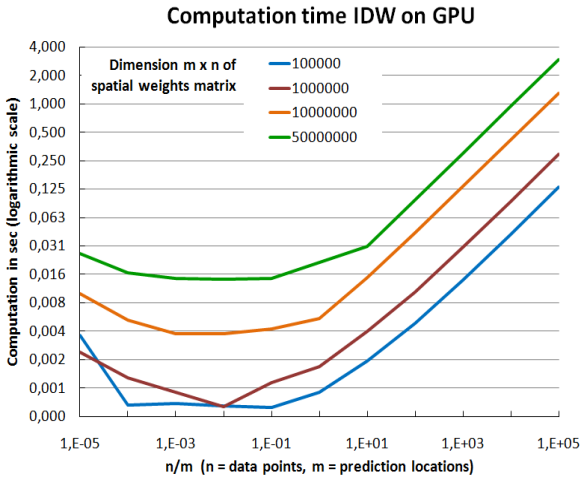

Figure 3: Performance of the IDW algorithm on the GPU for different data/prediction location ratios assuming coordinates in *2d* space. Note that the computation time in sec on the y-axis and *n/m* along the x-axis are in logarithmic scale. The computation times include the time needed for data transfer from and to the GPU. Each curve represents the behavior for a given number of entries (i.e. *m x n*) in the spatial weights matrix.

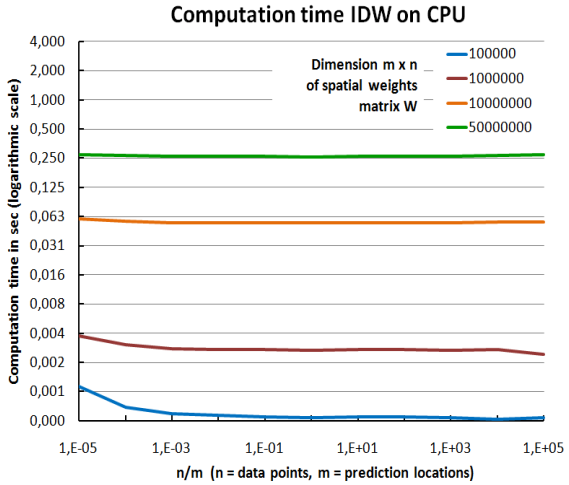

Figure 4: Performance of the single CPU implementation of IDW for different data/prediction location ratios assuming coordinates in *2d* space. Note that the computation time in sec on the y-axis and *n/m* along the x-axis are in logarithmic scale. Each curve represents the behavior for a given number of entries (i.e. *m x n*) in the spatial weights matrix.

4.3 Integration in software tools for geostatistics

In our experimental set-up, the GPU implementation of IDW scales well for prediction dominated interpolation tasks. The underlying idea of GPU accelerated matrix operations can be re-used and extended for some common geostatistical analysis tasks. Henneböhl et al. (2011) use OpenCL and the open source statistical environment R (R Development Core Team, 2010) to provide massively parallel functions not only for spatial interpolation but also for conditional and unconditional simulation of spatial random fields, using fast matrix operations and random number generation. The implementation follows a hybrid approach, taking advantage of CPU and GPU capabilities in a personal computing environment. Furthermore, we think that advances in multi-GPU computing as for instance discussed in Kindratenko et al. (2009) offer numerous opportunities not only for geostatistics but other GIS related, large-scale modeling tasks and satellite image processing.

5. CONCLUSION

This study evaluated the behavior of the well known IDW algorithm on a single GPU. The performance tests addressed in particular the influence of different ratios between data size and prediction locations while the overall problem size was kept constant. The massively parallel implementation is characterized by the fact that one thread interpolates the value for one prediction location. Although the absolute performance results are specific for the test setting and hardware equipment we conclude that spatial interpolation benefits from massively parallel computing environments such as programmable GPUs. The underlying ideas can be extended to provide a hybrid CPU/GPU framework with highly parallel functions for geostatistical analysis including unconditional and conditional simulation of spatial random fields. We consider this an important step to establish GPU computing as part of a general high performance computing framework for geostatistics.

REFERENCES

Cressie, N. (1993) Statistics for Spatial Data. Wiley Series in Probability and Statistics.

Henneböhl, K., Appel, M. (2011) Towards highly parallel geostatistics with R. Short paper accepted for "Geoinformatik 2011 – Geochange", June 15-17, 2011, Münster (Germany).

Huraj, L., Siládi, V., Siláci, J. (2010) Design and Performance Evaluation of Snow Cover Computing on GPUs. LATEST TRENDS on COMPUTERS Volume II, pp. 674-677.

Kindratenko, V., Enos, J., Shi, G., Showerman, M., Arnold, G., Stone, J. Phillips, J., Hwu, W. (2009) GPU clusters for high-performance computing. In: Proceedings on the IEEE Cluster'2009 Workshop on Parallel Programming on Accelerator Clusters (PPAC'09), New Orleans, Louisiana, USA, 2009, pp. 1–8.

Kirk, D. B., Hwu, W. W. (2010) Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

Munshi, A. (Ed.) (2010) The OpenCL Specification. The Khronos Group. – URL http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf (last access: Mar 5, 2011).

NVIDIA (2008) NVIDIA CUDA Programming Guide 2.0. – URL: http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf (last access: Mar 5, 2011).

NVIDIA (2009) NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Whitepaper V1.1 – URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (last access: Mar 5, 2011).

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., Phillips, J. C. (2008) GPU Computing. In: Proceedings of the IEEE 96 (2008), pp. 879-899.

R Development Core Team (2010) R: A language and environment for statistical computing. R foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/ (last access: Mar 5, 2011).

Rasmussen, C., Williams C. (2005) Gaussian Processes for Machine Learning. The MIT Press.

Srinivasan, B. V., Duraiswami, R., Murtugudde, R. (2010a) Efficient Kriging for Real-Time Spatio-Temporal Interpolation. 20th Conference on Probability and Statistics in the Atmospheric Sciences, American Meteorological Society, January 2010.

Srinivasan, B.V., Qi, H., Duraiswami, R. (2010b) GPUML. Graphical processors for speeding up kernel machines. Workshop on High Performance Analytics – Algorithms, Implementations, and Applications. Siam Conference on Data Mining, April 2010.

Yang, C., Raskin, R., Goodchild, Gahegan, M. (2010) Geospatial Cyberinfrastructure: Past, present and future. Computers, Environment and Urban Systems 34 (2010), pp. 264-277.

Zhang, J. (2010) Towards Personal High-Performance Geospatial Computing (HPC-G): Perspectives and a Case Study. In: Proceedings ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems, 2010, pp. 3-10.