

Implementing FastMap on the GPU: Considerations on General-Purpose Computation on Graphics Hardware

G. Reina and T.Ertl

Visualization and Interactive Systems Institute, University of Stuttgart

Abstract

In this paper we focus on the implications of implementing generic algorithms on graphics hardware. As an example, we ported the dimensionality reduction algorithm FastMap to fragment programs and thus accelerated it by orders of magnitude, allowing for interactive tweaking and evaluating of the algorithm parameters for datasets of several hundred thousand points and tens of dimensions; even the animation of structural changes in relation to parameters is possible. This allows to complement the algorithmic heuristic used by FastMap by explorative results from human interaction. Such an approach can be considered a heuristic in itself, but has the advantage of being based on visual feedback, therefore allowing for iterative improvement of the results. Thus we demonstrate how to benefit from the high execution parallelism on commodity graphics hardware as an alternative to making use of other, more costly, multiprocessing techniques. We discuss performance and bandwidth aspects as well as accuracy problems since these results are of more general interest and can be applied to general processing on graphics hardware as a whole.

Categories and Subject Descriptors (according to ACM CCS): I.3.8 [Computer Graphics]: Applications

1. Introduction

The recent years have seen a dramatic increase in available data in all scientific areas. Automated experiments and large-scale simulations provide a research base that has to be harnessed because human perception as well as available time is limited. Such data sets often have the additional drawback of consisting of so many attributes (or dimensions) that it is particularly hard to provide any kind of representation beyond the straightforward spreadsheet. Practical examples for such datasets are reaction maps, cancer screening databases, or results from molecular dynamics simulations, to name just a few. Several techniques for high-dimensional visualization have been developed over the years, like the classical parallel coordinates [Ins85], star glyphs [SFGF72], or the more recent circle segments [AKK96] or recursive pattern technique [KAK95]. Other approaches analyze the high-dimensional structure of the data to produce a semantically similar low-dimensional representation of it that can be visualized and understood more easily. This can be accomplished by classical MDS [BG97], the FastMap [FL95] algorithm we are employing, or by making use of clustering techniques like BUBBLE [GRG*99] or CSM [LC02]. Large

datasets imply large processing times when working with them in any way, and we believe that one very important step is to have the shortest time possible between first obtaining a dataset and getting an impression of the overall structure in such a dataset, concerning the distribution and similarities in it. We show that it is possible to speed up this process greatly by making use of current graphics hardware as co-processor. We take advantage of this scenario to focus on the implications when using the GPU in such circumstances as well as providing a performance analysis to differentiate the currently available GPUs and, above all, the bottlenecks that come into play when ‘outsourcing’ CPU work to the graphics card.

The rest of this paper is organized as follows: section 2 discusses the related work on multi-purpose GPU usage, section 3 details our GPU-based implementation of the FastMap algorithm. Section 4 shows the application we have implemented, section 5 and 6 discuss the performance and accuracy results in relation to the different graphics cards we tested and section 7 concludes this document.

2. Related Work

Since the advent of flexibly programmable graphics cards there has been great striving to make maximal use of the always increasing capabilities of modern GPUs. Some examples are new approaches in volume rendering using radial basis functions [JWH*04], the raytracing of scenes [PBMH02] or the calculation of radiosity [CHL04]. Several authors investigated geometry generation on the GPU, making use of superbuffers to evaluate higher-order surfaces, create vertices and render them directly on the graphics card [MP03], [LH04]. Other publications suggest to generate implicit geometry using fragment programs and then use a raycasting approach to render them. This minimizes stress on the system bus, transmitting only single billboards or point primitives to render geometrically complex objects like ellipsoids [Gum03], [KE04] or application-oriented glyphs [RE05]. Since the utilization of GPUs as co-processors for algorithms that do not directly generate images is becoming ever more wide-spread as well, a specific SIGGRAPH workshop has been created to deal with the challenges that arise [gpg].

The architecture of graphics cards is not yet as flexible as the general-purpose CPU, so some limitations have to be worked around before employing the graphics hardware for general calculations. Different methods have been devised for storing complex data structures in textures, and workarounds have been implemented for emulating program flow control, for example using the z-test [KW03]. Recently support for program flow control has been introduced with the so-called *Shader Model 3.0*. Most implementations share the property of making use of the *fragment units* even though also the *vertex units* can be programmed. This is motivated by the fact that the fragment processor has more parallel-working VPU (vector processing units) than the vertex processor, usually about twice as many. Furthermore the instruction set for fragment programs is more powerful and the access to data (in form of textures) is much faster, even if not exclusive anymore since Shader Model 3.0.

We have been confronted with large, high-dimensional datasets for some time now, and have employed the FastMap algorithm on different occasions, for example to allow for fast previews of large datasets [RE04]. There are several more sophisticated algorithms for dimensionality reduction, like MDS and the many improvements thereof (for example [MRC02], [MC04]). Even though the performance has been improved significantly, there is still the drawback of higher processing times, so for prompt overviews of large datasets we use FastMap nonetheless. Since FastMap is heuristics-based, we wanted to find out how the three-dimensional structure of a dataset is affected when changing the parameters for the projection. To be able to comfortably, interactively tweak those parameters, we first had to further reduce the processing times enough to make experimenting less tedious.

3. The FastMap algorithm

The general concept of FastMap is that it projects high-dimensional data into a lower-dimensional similarity space, i.e. the resulting points are distributed on few dimensions (for practical uses 2 or 3) such that their distance reflects their similarity in the high-dimensional space. This algorithm establishes that the target dimensions are axes which connect so-called pivot points, which ideally should represent the most distant points in the high-dimensional space. However, this is also the main flaw of the algorithm, since no primary component analysis or the like is performed to take into account the spatial distribution of the points, but for the sake of execution time the points are heuristically chosen based just on their distance. The error in inter-point distances that results from the projection on one of these pivot-pivot axes is compensated when the next coordinate is calculated by projecting it onto the next axis, until there are as many coordinates as are needed. An optimal choice of these points is difficult and costly since a complete distance matrix of all the processed points is $O(n^2)$ both in time and space. So the heuristic works by randomly choosing a point and searching for the farthest point in the dataset as opposite end of the axis. This step is repeated a preset number of times alternating between the two end points of the axis and executed for each of the desired target dimensions to obtain all of the required pivots.

The projection used by FastMap is quite straightforward. After the pivot points have been chosen, equation 1 gives the position of a point on an axis, where $d_{i,j}$ represents the high-dimensional distance between two points, a and b are the pivot points for the particular axis and i is the point we want to project.

$$x_i = \frac{d_{a,i}^2 + d_{a,b}^2 - d_{b,i}^2}{2d_{a,b}} \quad (1)$$

The high-dimensional distance can be of any metric desired, for example the common euclidean distance, or any fractional distance because of the better response to noisy datasets [AHK01]. In our example we use the euclidean distance. For the target dimensions following the first one, the distance is modified by the error of the previous projection:

$$d'_{i,j} = d_{i,j}^2 - (x_i - x_j)^2 \quad (2)$$

We want to obtain these values as quickly as possible after determining the pivots, but calculating a distance matrix for all the combinations of pivots and points does not yield a speedup since each of these distances is costly, but only needed once. However we have the possibility to parallelize this part of the algorithm, since the points do not influence each other and we can store the pivots globally. We could have used a multiprocessor machine or a cluster to speed things up, but such hardware is not available widely and still quite expensive. Therefore we made use of the massively parallel fragment processing units available on commodity graphics cards. These processors offer floating point preci-

sion in data storage and computation and are flexibly programmable. The restrictions that apply in this area (no geometry can be generated in the vertex units, fragments cannot be relocated) and the costly program flow control are not an issue since we only want to use the GPU as a relatively cheap floating-point coprocessor.

The GPU implementation

To execute FastMap on the GPU, we first prepare the pivots on the CPU by using the heuristic described above. The source data is then split into several floating point textures as follows: all integer and floating-point attributes of the source data are stored in groups of four in the red, green, blue and alpha channel of textures, all on the same coordinate for one single data point. Strings are mapped to unique IDs per attribute (so if we have 5 string-type attributes in a dataset, we get 5 distinct string lookup tables). This makes us save memory since strings are usually categorical attributes and thus repeated several times in one dataset. On the other hand this makes string comparison much faster since we can define the difference between two strings as simple inequality, and for categorical values an editing distance would not make sense anyway. The resulting IDs are also stored in texture color channels. If one looked at the resulting stack of textures from above, all the attributes of one point would be on the same line of sight (see figure 1), so we can access all of the attributes by using a single texture coordinate.

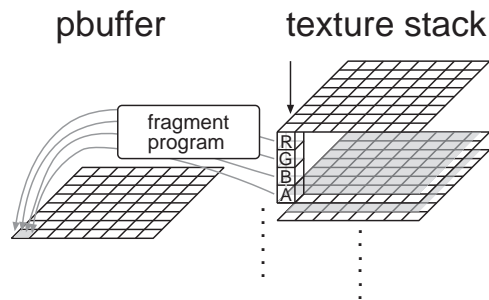


Figure 1: Data as textures on the graphics card, one ‘column’ contains all attributes of one data point

Depending on the number of attributes and points in the source data, the texture and stack size is varied, with differing impact on rendering performance (see section 5). Only one texture stack is processed per rendering pass, yielding one projected coordinate for $texture_size^2$ points. This rendering process is repeated until all input data has been reduced to lower dimensionality (resulting in $\lceil \frac{n}{texture_size^2} \rceil$ iterations). The stack of textures is complemented by another floating point texture, a *pbuffer* [pbu], as the rendering target, with the same resolution as the texture stack. The x_i calculated from the input stack is stored inside this pbuffer. If we

bind the pbuffer as texture, we can easily access the results of a previous projection and can thus calculate the modified distances d' . We calculate all result dimensions for a stack consecutively, in order to keep the source data textures untouched for two additional passes (in our case of a 3D similarity space).

To trigger the calculations, we render a single quad covering the whole viewport to obtain the texture coordinates for processing every data point/pixel in the input texture stack. The viewport has the same dimensions as the pbuffer. We utilize an *ARB_fragment_program* that stores an x_i as above in the pbuffer, depending on the distance calculated from the source data and the attribute values of the pivot points, which are passed as program parameters. This is justified by the fact that the pivot point could be in another texture stack and thus would not be accessible. We could store the pivots at a constant position in the input stack and lose just two data points per iteration, but we can save $2 * stack_height$ texture lookups per result fragment if we store them as parameters; the pivot points are constant for the whole stack, and thus one rendering step, in any case. $d_{a,b}$ and $d_{a,b}^2$ from equation 1 are also constant and therefore passed as parameters.

The fragment program basically consists of three blocks of code: the floating point attributes are retrieved, subtracted from the pivot attributes, squared and added up, yielding a quadratic euclidean distance. String IDs are checked for equality against the pivot values and added accordingly in a subsequent block. The last block calculates x_i . We can completely avoid using the square root operation since all the distances we calculated before are squared again in this block (see equation 1). The necessary code is dynamically generated based on the dimensionality and dimension types of the input file. We only need to generate the respective block per texture; only the source texture unit and the program parameter containing the respective attributes of the pivots must be set accordingly.

The fragment program for subsequent dimensions is generated in the same way and just needs two more parameters (the latest projections of the pivots) and one more texture fetch for the latest projection of the point itself (from the pbuffer of the preceding pass), along with another code block for the calculation of d' from d using these latest projections. After enough dimensions have been calculated (up to four per *pbuffer*), we can read back the results to the main memory and use them as a vertex array to display the resulting low-dimensional representation of the data set. We could also have used the result as a vertex buffer as in the superbuffer specification, but only ATI cards support it and practical tests by colleagues confirm that it does not work very reliably.

4. Results

We tested our implementation with different excerpts from a 10D dataset of 2.3M points retrieved from a cancer screening database which we projected into 3D similarity space.

This dataset contains 8 numerical and 2 categorical alphanumeric attributes and quantizes the reaction of various cancer cells towards different substances in terms of cancer growth inhibition. In Figure 2 the result from using the original heuristic on a 1M point subset is shown in the application we have implemented. The OpenGL window shows the dataset in 3D similarity space with the pivots highlighted and colored according to the target dimension they belong to (in RGB order). A short fragment program is used to cool/warm-color-code the scatterplot depending on the z depth of a point. Since unshaded points do not occlude each other in a way that allows for satisfactory visual depth perception, we used this approach to convey a better impression of depth to the user. Intensity attenuation would also have been an option, however the darkened points would be quite difficult to perceive because of their small size. The parameter window allows the user to scroll through the different pivot points in real time or even to animate one of the axes with adjustable step size and delay between steps. Figure 2 shows some well-defined streaks with similar data, however the result points all lie on a plane, so we use only two of the available three dimensions.

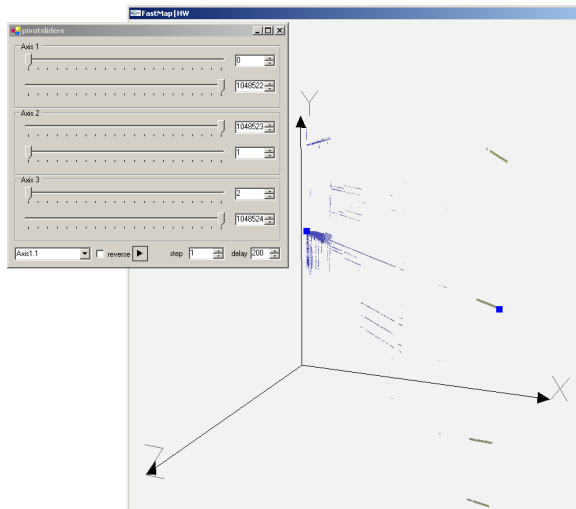


Figure 2: Application user interface with OpenGL 3D scatterplot (right) and FastMap parameter window (left). Pivots have been chosen using the original heuristic.

When tweaking the pivot points by hand in real time, other cluster-like arrangements can be spotted. However we also found that choosing pivots that are quite close on the resulting axis can yield results which are fanned out more thoroughly than when always using points that are as far from each other as possible. Figure 3 shows such a hand-tweaked result and suggests that three major clusters exist, the bottom one being clearly composed of overlapping streaks, which hints at several series of data with slowly varying attributes. This does not mean that the original projection is faulty or

useless, but that we can spot different characteristics in one dataset when making use of human interaction and experimentation as an added heuristic to complement the automated one.

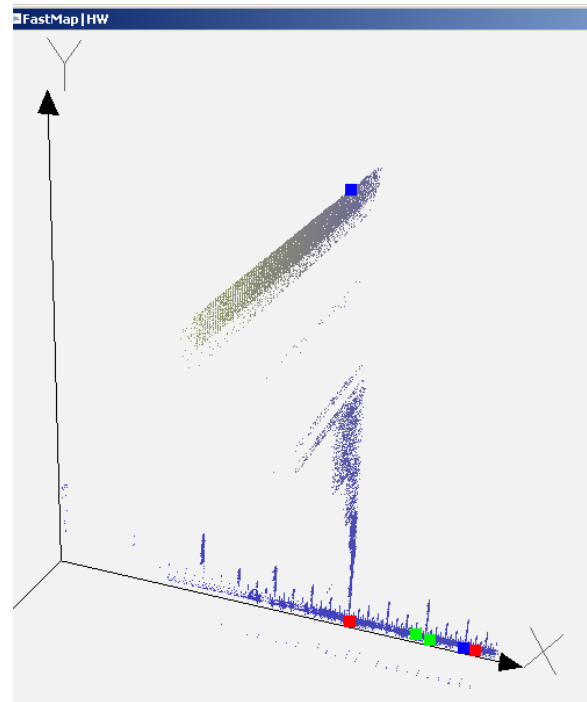


Figure 3: FastMap result with tweaked pivots. The result points are much more widely spread in 3D.

5. Performance

To demonstrate that our approach is an improvement over CPU-based FastMap, we have taken the timings of the implementation for various datasets on an Intel Pentium4 running at 2,4Ghz and on a GeForce 6800GT (see figure 4). The CPU-based FastMap also uses only 32bit floats in order not to penalize its performance further by using doubles. We used different excerpts from the cancer screening dataset mentioned in section 4. The first subset is very small (26824 items) while the second consists of one million items. The higher-dimensional datasets are just repetitions of the original data to allow for simple investigation of the performance variation when increasing texture reads per result point. One can see that the GPU implementation clearly outperforms the CPU variant by orders of magnitude, allowing for interactive adjustment of the pivot points. The projection times stay below one second for source data sizes of up to two million data points and 10 dimensions (since two color channels are left in one of the source textures, we could get the same performance for up to 12 dimensions).

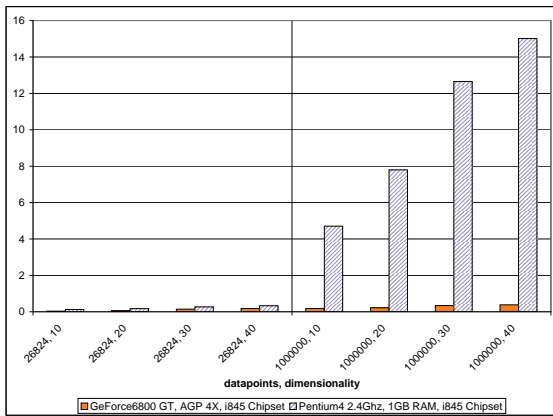


Figure 4: Total time in seconds for FastMap on CPU and GPU with optimum texture size

We wanted to further investigate how the different parameters of the implementation, like texture sizes, system bus types and, last but not least, graphics chipsets, affect these numbers. One would think that the optimal parameters for executing such an algorithm in graphics hardware would be the use of as few and as large textures as possible, to keep management overhead at a minimum. The textures would be created once and reused for every stack that has to be iterated. By and large this is true, but the measurements we have taken show where specific strengths lie for different GPUs and some unexpected flaws that must be taken into consideration when using them.

We chose OpenGL for the implementation of the GPU FastMap in order to retain the option of easily integrating it with our existing OpenGL-based solutions which could benefit from the fast dimensionality reduction. This poses a challenge when it comes to performance measurements which go beyond simple FPS for final visualization. Therefore we repeated our measurements many times and also made sure that the GL pipeline was flushed after the timed phases. The measurements were also taken with constantly varying pivots so we could also visually verify the constantly changing output to make sure the graphics drivers would honor the *glFinish()* request. We can see in figure 5 that the overhead for constantly replacing the contents of small texture stacks is much higher on nVidia chips than on the ATI chips, so if we cannot use the whole graphics card memory for FastMap, the approach becomes extremely slow. However with big textures the results are similar to the ATI X800 AGP. Another irregularity we discovered was extremely bad readback performance on the PCIe X800 card with texture sizes of 512^2 and above, despite the acceptable results from the AGP variant. Comparing the calculation of the 1M 10D dataset, we found that moving from four stacks of 512^2 textures to one stack of 1024^2 decreased the performance by

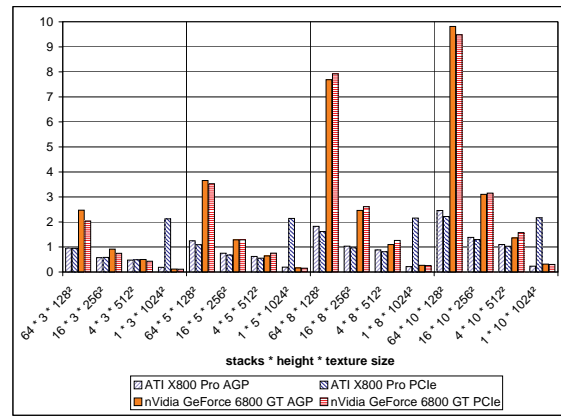


Figure 5: Total time in seconds for FastMap on different GPUs with different tiling of the 1M dataset and increasing dimensionality

four times, after the already irregularly small increase from 256^2 (see also table 1). We also encountered a side-effect on an ATI X800 Pro AGP, where the 20-dimensional dataset with 1 million items distributed over 4 stacks of 512^2 -sized textures produced a delay after each calculation, so the measured 623ms produced only 1 result frame per second. This seems to be a driver bug, since for example 4 stacks of $8 \cdot 512^2$ -sized textures resulted in no such delay, and the same executable with the same parameters did not cause a delay on any other configuration. We cross-checked the readback

card	512^2	1024^2
GeForce 6800GT AGP	750.47	751.31
GeForce 6800GT PCIe	818.92	829.37
ATI X800 Pro AGP	117.21	116.36
ATI X800 Pro PCIe	235.32	7.56
ATI 9700 Pro AGP	114.03	113.31

Table 1: Readback performance in MB/s for RGBA float buffers of given size from a particular graphics card.

performance with an individual test, the results of which can be seen in table 1. It is conspicuous that the readback performance has not improved moving from ATI 9700 to X800 (for the AGP cards). The result for 512^2 textures shows that the native PCI Express interface on the X800 cards is an improvement over the AGP interface, however the cards cannot catch up with the current nVidia chips (at least when working with float formats). The lack of a significant performance improvement on the GeForce when moving from AGP to PCIe seems to be caused by the fact that the chip does not have a native PCIe interface, but uses a transponder chip instead.

An additional factor of the overall performance of the al-

gorithm is the execution speed of the code on the GPU. A detailed analysis has been conducted in this regard [DE04], however we wanted to find out the specific effects for our case. In figure 6 we can see that current generation of ATI chips has a big advantage over the nVidia cards. The cause of this is two-fold: the ATI cards have a much higher core clock and make use of only 24bit-floating-point VPUs (see also section 6). We can also see that the ATI X800 scales much better with increasing number of texture reads and constant texture size than the GeForce 6800, but has a weak spot when it comes to 512^2 texture sizes. The ATI X800 PCIe could have the performance lead (if not the precision lead) were it not for the much better readback performance of the nVidia cards regardless of the system bus employed. The CPU bus and chipset should also not be underestimated for its performance impact. We compared a i845-based system with 400Mhz FSB and AGP 4X to a i865-based system with 800Mhz FSB and AGP 8X, using a GeForce 6800GT in both cases. We measured a performance increase of over 200% instead of the optimistically anticipated 100% which demonstrates that also different chipsets and the increased FSB bandwidth have a significant performance impact.

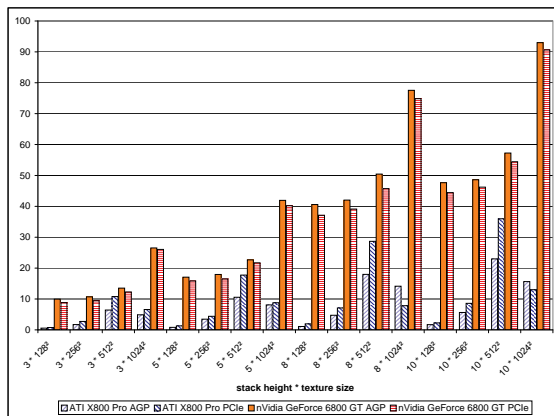


Figure 6: Fragment program execution time in milliseconds per stack with different stack sizes and texture sizes

6. Precision

If the GPU is used to replace the CPU, a major issue that arises is the computational accuracy. ATI chips currently offer only 24 bits for computational accuracy, even though source and result data is stored as 32bit IEEE floats. nVidia chips allow the programmer to select between half (16bit) and full (32bit) precision floats for computation, while storage relies on 32bit IEEE floats as well. One advantage of the CPU over the GPU is the availability of higher-accuracy number formats and the possibility of at least using these for calculation, even if the results are then stored only as floats

Operations	ATI X800/9700	nVidia GF6800
x	0.000003636	0
$x * x$	0.013201884	0.000023515
$x \frac{1}{x+1}$	0.000016217	0.000000005
$(x+x)(x+x)$	0.052807537	0.000094060

Table 2: Computational accuracy (average relative error) for selected operations on different GPUs.

(if we want to save memory, but want to reduce the accumulating error). We investigated the effects the precision has on the relative error of some example operations. The results can be seen in table 2. These values are obtained as follows: A small test program uploads a RGBA float texture to the GPU, activates a fragment program and draws a viewport-filling quad into a pBuffer. Then the pBuffer is read back and compared to CPU-based results calculated with double accuracy which represent the ideal, if not correct, result. This obviously is a worst-case scenario since also on the CPU normal floats would be less precise. The texture contains $512^2 * 4$ floats with the value $\frac{1}{512}$ ($index + 1$), that is, values in the range $[\frac{1}{512}, 2048]$. As a first test we simply passed the values from the texture to the pBuffer (x in the table). Since the data has to pass through the VPU, we already get an error on ATI chips from this. The other tests execute some simple calculations. We chose a different denominator from x in the reciprocal multiplication to avoid that the operation simply be discarded by optimization in the drivers. We can see that the 24bit VPU accumulates a large error quickly, however the implications only become clear when we consider some applications. If we calculate a position for displaying data (as is also the case with FastMap) and normalize each dimension to allow us to fill a unit cube, and we display this cube at screen size (1200 pixels in height), this would result in an average vertical positional error of 60 pixels on an ATI card when considering the 5% relative error ($(2x)^2$ in table 2). On an nVidia card this error would amount to barely $\frac{1}{10}$ of a pixel, which is more than sufficient. As these cards are originally intended for delivering good performance and visual effects for computer games, this can be considered adequate, but for general processing on graphics cards it is a factor that must be kept in mind (especially when using ATI cards). We also measured the error of the FastMap algorithm and compared it to FastMap on the CPU (using only floats as in the performance comparison). For the 1M dataset the ATI card produced an average relative error of 0.000294823, the nVidia card one of 0.000056497, so the discrepancy is well below one pixel in both cases if the same assumptions as above apply.

7. Conclusion and Future Work

We have demonstrated how graphics hardware can be used to improve the execution times of FastMap, opening up the possibility of interactively changing the parameters and observ-

ing the structural changes in the resulting low-dimensional data. We also have discussed the more generally applicable performance and accuracy constraints that come into play when using GPUs as co-processors, pointing out some pitfalls to keep in mind when considering to move an algorithm from the CPU to the GPU. From this example we can see that if we keep these limitations in mind, a modern GPU can provide a very cost-effective way to execute massively parallel algorithms. With the availability of high-level languages like GLSL or Cg, or specialized frameworks like Brook [Buc03], [BFH*04], the porting of an algorithm has become quite easy if one finds an efficient way to map the data structures onto ‘flat’ arrays (textures).

For the future we would like to give a more thorough analysis of the factors that influence the resulting accuracy of GPU calculations, like inter-operation and value-range dependent effects. We would also like to rewrite the algorithm for Brook for easier comparison of Direct3D and OpenGL performance. Additionally we can now use our implementation to investigate different heuristics for choosing pivot points as well as manually optimize the low-dimensional projections produced by FastMap by using hand-selected pivots.

Acknowledgements

We would like to thank Thomas Klein for his additional bandwidth benchmarking results and fruitful discussion.

References

- [AHK01] AGGARWAL C. C., HINNEBURG A., KEIM D. A.: On the surprising behavior of distance metrics in high dimensional space. *Lecture Notes in Computer Science 1973* (2001), 420.
- [AKK96] ANKERST M., KEIM D. A., KRIEGEL H.-P.: Circle segments: A technique for visually exploring large multidimensional data sets. In *Proceedings Visualization '96* (1996), IEEE Computer Society.
- [BFH*04] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH 2004* (2004).
- [BG97] BORG I., GROENEN P.: *Modern Multidimensional Scaling*. Springer Verlag, New York, 1997.
- [Buc03] BUCK I.: Data parallel computing on graphics hardware. In *Graphics Hardware '03* (2003).
- [CHL04] COOMBE G., HARRIS M. J., LASTRA A.: Radiosity on graphics hardware. In *Proceedings of Graphics Interface '04* (2004).
- [DE04] DIEPSTRATEN J., EISSELE M.: In-Depth Performance Analyses of DirectX9 Shading Hardware concerning Pixel Shader and Texture Performance. In *Shader X3* (2004), Wolfgang Engel, (Ed.), Charles River Media.
- [FL95] FALOUTSOS C., LIN K.-I.: FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (1995), pp. 163–174.
- [gpg] SIGGRAPH 2004 GPGPU Workshop <http://www.gpgpu.org/s2004/>.
- [GRG*99] GANTI V., RAMAKRISHNAN R., GEHRKE J., POWELL A. L., FRENCH J. C.: Clustering large datasets in arbitrary metric spaces. In *ICDE* (1999), pp. 502–511.
- [Gum03] GUMHOLD S.: Splatting illuminated ellipsoids with depth correction. In *VMV* (2003), pp. 245–252.
- [Ins85] INSELBERG A.: The plane with parallel coordinates. *The Visual Computer*, 1 (1985), 69–91.
- [JWH*04] JANG Y., WEILER M., HOPF M., HUANG J., EBERT D. S., GAITHER K. P., ERTL T.: Interactively Visualizing Procedurally Encoded Scalar Fields. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '04* (2004), Deussen O., Hansen C., Keim D., Saube D., (Eds.).
- [KAK95] KEIM D. A., ANKERST M., KRIEGEL H.-P.: Recursive pattern: A technique for visualizing very large amounts of data. In *Proceedings VIS '95* (1995), IEEE Computer Society, p. 279.
- [KE04] KLEIN T., ERTL T.: Illustrating Magnetic Field Lines using a Discrete Particle Model. In *Workshop on Vision, Modelling, and Visualization VMV '04* (2004).
- [KW03] KRUEGER J., WESTERMANN R.: Acceleration techniques for gpu-based volume rendering. In *Proceedings IEEE Visualization 2003* (2003).
- [LC02] LIN C.-R., CHEN M.-S.: A robust and efficient clustering algorithm based on cohesion self-merging. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining* (2002), ACM Press, pp. 582–587.
- [LH04] LACZ P., HART J. C.: Procedural Geometric Synthesis on the GPU. In *Proceedings of the GP² Workshop* (2004).
- [MC04] MORRISON A., CHALMERS M.: A pivot-based routine for improved parent-finding in hybrid mds. *Information Visualization* 3, 2 (2004), 109–122.
- [MP03] MĚCH R., PRUSINKIEWICZ P.: Generating subdivision curves with L-systems on a GPU. In *GRAPH '03: Proceedings of the SIGGRAPH 2003 conference on Sketches & applications* (2003), ACM Press, pp. 1–1.
- [MRC02] MORRISON A., ROSS G., CHALMERS M.: A hybrid layout algorithm for sub-quadratic multidimensional scaling. In *IEEE Information Visualization '02* (2002), pp. 152–158.
- [PBMH02] PURCELL T. J., BUCK I., MARK W. R.,

HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics* 21, 3 (July 2002), 703–712. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[pbu] OpenGL ARB pbuffer specification, http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_pbuffer.txt.

[RE04] REINA G., ERTL T.: Volume visualization and visual queries for large high-dimensional datasets. In *VisSym* (2004), pp. 255–260.

[RE05] REINA G., ERTL T.: Hardware-Accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization. In *EuroVis '05* (2005).

[SFGF72] SIEGEL J., FARRELL E., GOLDWYN R., FRIEDMAN H.: The surgical implication of physiologic patterns in myocardial infarction shock. *Surgery* 72 (1972), 126–141.