UPPSALA
UNIVERSITET

Multi-GPU cluster wave propagation and OpenGL visualization

Jacob Lundgren, Nils Olofsson

Project in Computational Science: Report

February 3, 2014

PROJECT REPORT

Institutionen för informationsteknologi

Abstract

The inherent issues of properly deploying finite difference calculations onto GPUs are described and solutions are suggested. A speedup of 60x is achieved over the CPU version.

Four visualization methods were implemented using OpenGL and compared in terms of the clarity of their visual result. A combination of hedgehogs and slices was deemed to give the best result.

1 Introduction

1.1 Problem outline and numerics

Numerical solution of the wave equation is as ever a topic of scientific interest. In order to satisfy a demand for ever more fine-grained analysis, while still delivering results in a reasonable time frame, constant improvements of the solution algorithms and their implementations is imperative.

In this report, the issues of accelerating the solution of the wave equation using a cluster of *graphics processing units (GPUs)*, and their potential solutions are outlined. The numerical model is a sixth-order SBP-SAT scheme, applied to a three-dimensional domain of equidistant cartesian mesh points. For simplicity, periodic boundary conditions are used.

1.2 Visualization

Visualization of three dimensional data is not always a trivial problem, both in terms of computational feasibility and in the sense that the end result should be images that are understandable. In order to produce such imagery a lot of methods and algorithms have been developed. This report describes four different volume rendering techniques, how these may be implemented using a graphics API and how they perform when applied to three dimensional wave propagation. The methods were all implemented using C++ and OpenGL and resulted in a light application capable of displaying volumetric data sets.

2 Theory

2.1 GPU Architecture

In order to properly utilize the capacity of modern GPUs, attention must be paid to their internal structure and hardware characteristics. The description given in this report will pertain specifically to NVIDIA devices, and while they share certain characteristics with those of competing producers, no specific similarities can be assumed. References throughout will be the *CUDA C Programming Guide* [4], the *CUDA C Best Practices Guide* [5], as well as the *Parallel Thread Execution ISA* documentation [6].

2.1.1 Overview

A very fundamental issue that different computer hardware architectures need to address is that of minimizing the cost of data transfer, in particular between main memory and the computational units requesting it. Normally, CPU designers attempt to alleviate this by implementing a complex hierarchy of caches. Unfortunately, this is a very costly solution in terms of chip transistors dedicated to other tasks than computation. In contrast, GPUs are designed with a different mindset, where data transfer penalties are instead hidden by near-instant switching of tasks and high degrees of parallellism.

GPUs consist of a large amount of comparatively slow computational cores, and this fact delineates the manner in which they are to be programmed. In

order to attain the potential of these devices, the burden is put on the programmer to expose the parallelism of the tasks to be performed. Once sufficiently many independent operations are available to the device, the dependencies on data currently in transfer can be handled by scheduling other threads. Due to the relative simplicity of the resources available to each thread, the hardware amount required to facilitate near-instant thread switching is significantly lower than that of the CPUs cache hierarchy.

The cores on a GPU are organized into *Streaming Multiprocessors (SMs)*, hardware units also containing the components to arbitrate the aforementioned switching and scheduling of threads. Also attached is a small cache which can alternatively be used as *shared memory* for the threads currently resident on the SM. This allows for a limited amount of cooperation in what is otherwise a strictly enforced independent execution structure.

2.1.2 GPU Program Structure

When writing programs utilizing CUDA C for deployment onto NVIDIA GPU devices, a particular structure is required. The most apt view of the GPU is that of a separate unit onto which computationally heavy tasks can be offloaded. The overall control including initialization, memory management and launch of specific tasks is left to the CPU. This leads to a separation of the program code into CPU or *host* code, and GPU or *device* code.

In CUDA C the device code is written in the form of a *kernel* function which is executed on the GPU by each thread launched. The threads launched in each kernel call are structured in a *grid* consisting of all threads from the specific launch, as well as into the smaller unit of a *block*. These blocks consist of subsets of the launched threads which are scheduled onto the SMs as a group. This means that threads in each block can use shared memory to communicate with each other, as well as perform rudimentary synchronization operations.

2.1.3 Optimization Philosophy

The main priority in the design of GPU applications is ascertaining the availability of sufficiently many threads to hide memory latency. An algorithm lacking the potential for this degree of parallelism needs restructuring in order for deployment on GPUs to be a viable option. Apart from the obvious requirement that the task be divisible into independent parts, this also means that the resources allocated to each thread can not surpass a certain amount.

2.1.4 Parallel Thread Execution

The compilation process for the code intended for deployment on an NVIDIA device written in CUDA consists of several intermediary steps between source code and finished object or executable file. These steps include the pseudo-assembly language *Parallel Thread Execution (PTX)*. By having the compiler halt the process at this stage, it is possible to observe whether the more abstract operations at the CUDA C level produce superfluous instructions.

As a particularly pertinent example, the produced PTX code exposes the memory access behavior of the application, allowing for identification of potential issues. Since the memory spaces on GPUs are noticeably disparate and

their misuse can lead to significant performance losses, this capacity is not to be understated.

2.2 Volumetric Visualization

Visualization of volumetric data has been an active field of research for the last couple of decades and several approaches, algorithms and methods have been developed. There are two major families of methods: image order and object order. Image order methods iterate over the pixels (or equivalent) on the rendering target and assign colors to the individual pixel depending on what it covers. Image order methods are usually some kind of ray casting methods in which a ray is cast for each pixel into the active viewing volume and the color of the pixel is set to the color of the object hit by the ray. Object order rendering methods iterate over graphic primitives of a particular type and assigns colors to pixels by projecting them onto the rendering target. Both families are covered by the methods mentioned in this report. In the image order family we have maximum intensity projection and in the object order family we have the isosurface and hedgehog methods that both create renderable objects from the dataset. Both families have their own motivations, strengths and possible weaknesses. The obvious drawback of the object order methods is that additional data often has to be created and as in isosurface methods this may actually increase the amount of data that has to be rendered. Object order rendering is however the most common in the games and movies industries and so hardware, and to some extent methodology, has been focused around these methods for the last two to three decades. For example GPUs and their APIs have been developed mainly with this kind of rendering in mind. Image order rendering does not necessarily need any additional preprocessing but traversing a ray through a dataset for all of the millions of pixels on a modern computer screen is often not a cheap operation. It is however very parallel in nature so e.g. with general purpose GPU (GPGPU) computing, the practice of using GPUs for non-graphics applications, or shader implementations it is possible to perform in realtime even on low end consumer grade hardware.

2.3 3D Graphics and OpenGL

As mentioned in Section 2.2 both the underlying rendering hardware and OpenGL have been developed with object order rendering in mind so all methods covered by this report have been implemented in this way with the necessary workarounds for the image order method. 3D graphics are often seen as a pipeline, Figure 1, where an application feeds the pipeline from the left with vertex data and additional information such as transformation matrices, texture data, lights and more. The data is then fed through the different pipeline stages until the final image has been assembled. The image is then typically written to a frame buffer of some kind and then displayed on the computer screen. All stages of the pipeline used to be fixed function and the functionality of the pipeline was changed by calling specific OpenGL functions and setting different environment variables. Since OpenGL versions 1.4 and 2.0 parts of the pipeline have become programmable by the use of shaders. Shaders are relatively small programs compiled at runtime that are executed on the graphics hardware. In Figure 1 the vertex shader and fragment shader stages can be seen. In later

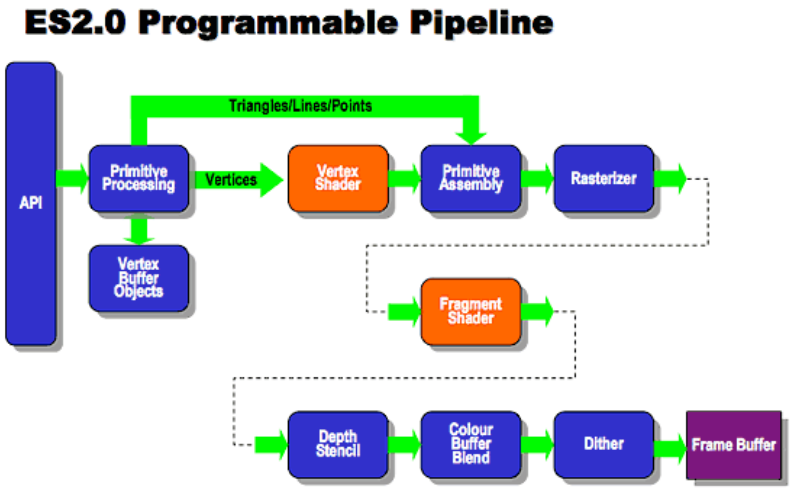


Figure 1: The graphics pipeline for OpenGL 2.0 ES, an OpenGL standard for embedded and mobile systems. Image from khronos.org

desktop versions of OpenGL more shader stages have been added, all of them optional, such as the geometry shader which would lie between the vertex shader and the primitive assembly in Figure 1, and the tesselation shader. In Figure 2 a simple example of a complete shader program with a vertex and fragment shader can be seen. This shader program will just transform the given vertex using the transformation matrix in the vertex shader and then set the color of the fragment to the color of the vertex in the fragment shader.

OpenGL is an open source graphics API first released in 1992 and has been in constant development ever since. It has a lot of different implementations and ports for various operating system and hardware vendors. The purpose of the API is to give the programmer an abstraction layer to the graphics processing unit (GPU). An example of a similar but less portable API that could be used is Microsofts DirectX which is exclusive for Windows and some other Microsoft platforms. The APIs share functionality and so in theory anything written using OpenGL could also be written using DirectX and vice versa.

3 Method

3.1 Hardware Configuration

The testing in this project was performed on a Beowulf cluster consisting of one coordinating main node and two dedicated computational nodes. The main node was equipped with two NVIDIA GeForce GTX 580 GPUs, while the computational nodes each housed four NVIDIA GeForce GTX 670 GPUs. For comparison with CPU performance, the hexacore Intel Xeon E5645 CPU on the main node was utilized.

The nodes were connected via Ethernet through a common switch allowing uniform inter-node communication. The computational nodes had no dedicated

```
// --------------------------------------
// Simple vertex shader
// --------------------------------------
#version 330
layout(location=0) in   vec4 vPos;  // Vertex position in model space
layout(location=1) in   vec4 vColor;// Vertex color
out                     vec4 color; // To next shader stage

uniform mat4 pvmMatrix; // Transform matrix

main() {
    gl_Position = pvmMatrix*vPos;   // Transform to projected view coord
    color       = vColor;           // Volor of vertex to fragment shader
}

// --------------------------------------
// Simple fragment shader
// --------------------------------------
#version 330
in vec4 color;
main() {
    gl_Color = color;   // Set fragment to color from vertex shader
}
```

Figure 2: Example of vertex and fragment shaders

storage space, and instead were booted from the main node via this connection. The users' home directories were similarly shared and mounted on these machines, providing easy access to required files.

3.2 Solver code

A functioning version of the solver utilizing the GPUs in the cluster was provided at the start of the project. This code was written in the C programming language, using the *Message Passing Interface (MPI)* system to distribute tasks among specific GPUs. The deployment of computations onto GPUs was done using CUDA C.

3.3 Profiling

In order to properly dedicate optimization efforts to where they are most likely to be effective, various profiling tools can prove helpful. Due to the complexity of MPI-distributed CUDA tasks, however, the set of tools available is limited. The University of Oregon *Tuning and Analysis Utilities (TAU)* suite proved to have the required capacity, and thus it was set up on the cluster.

The data provided by the suite includes kernel durations, MPI function call durations, CUDA runtime call durations as well as data on several factors important to GPU performance such as occupancy and register spills.

3.4 Visualization application layout and organization

In Figure 3 the structure of the application can be seen. Note that the aggregation dependence often denotes a pointer or a collection of pointers. The main application class contains a GraphicsManager that among other things compiles and contains all shader programs and also contains information about

the graphics window. It also contains an InputManager that can be quired for key presses and mouse movement. Apart from these system components it also has a main container class called World which in turn contains a Volume. The Volume component stores the volumetric data and its metadata such as size, grid, gradient and more. From the volume several renderable components can be created, each corresponding to one of the methods mentioned in Sections 3.5 to 3.8. The application itself is run in a simple rendering loop where the application components are updated and rendered once every iteration until some exit flag is given by the user.

```
//Rendering loop
while(running) {
    readInput();
    update();
    render();
    processEvents();
}
```

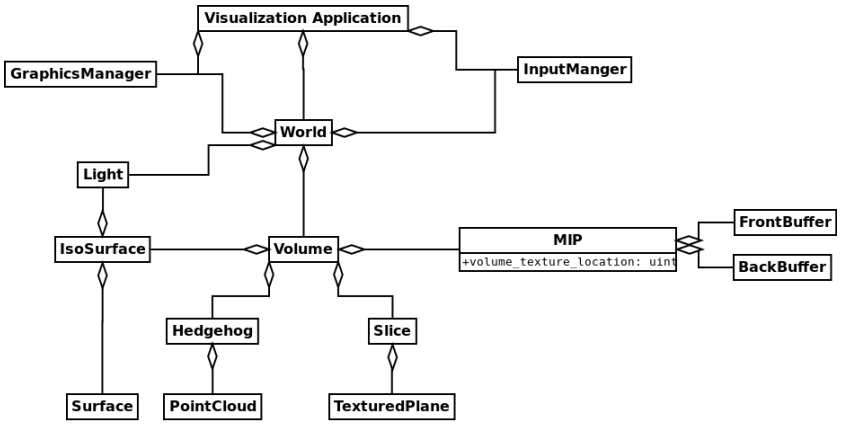


Figure 3: A schematic layout of the application components.

3.5 Isosurface extraction using marching cubes

Marching cubes [1] is an isosurface extraction method based around the premise that given a cube with a data point at each corner a surface can only intersect that cube in a finite number of ways. Given a threshold value, the value of the scalar isosurface, each of the eight data points is assigned either a logical 1 or 0 depending on whether or not they are equal or greater then the threshold. This gives $2^8 = 256$ different configurations, which after considering that the inverted and non-inverted cases produce the same kind of intersection and that there are rotationally symmetrical versions the total amount of cases is reduced to the 15 seen in Figure 4. Each data point contains its position and also a normalized gradient. Before the triangle data is entered into a vertex array these values are linearly interpolated along the edge that connects the vertices on the cube.

Among the methods mentioned in this report this is the most elaborate, considering both the implementation and the computational effort. It is the only method that requires an illumination model. The created surface is lit by a single non attenuating point light far from the scene using a specular Phong illumination model [2] with per vertex normals.

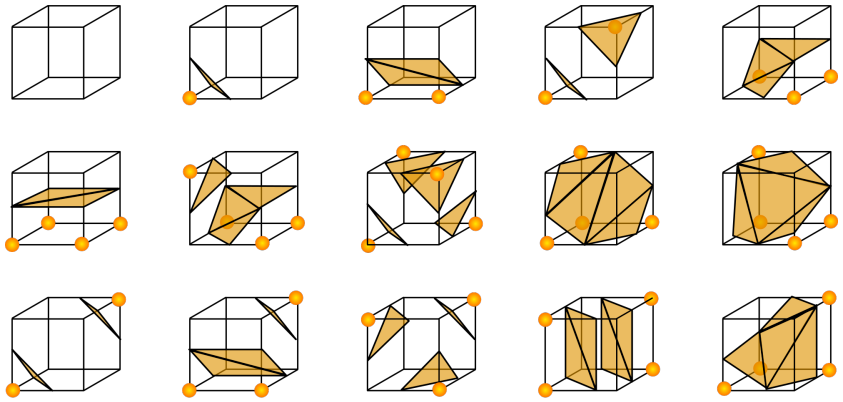


Figure 4: The 15 marching cube cases after symmetric duplicates have been removed. Image from wikipedia.

```
// --------------------------------------
// Main entry for gemotry hedgehog shader
// Creates a colormapped line segment
// --------------------------------------
main() {
    // Create vertex at original grid location
    gl_Position = vdata[0].pv*vdata[0].position;
    gdata.color = vec4(0.1,0.1,0.1,1.0);        // set it to some color
    EmitVertex();
    // Create vertex by adding an offset distance to the original location
    gl_Position = vdata[0].pv*(vdata[0].position + size*vec4(vdata[0].↩
        normal,0.0));
    gdata.color = colorMapJet(vdata[0].value); // set to colormapped value
    EmitVertex();

    EndPrimitive(); // End the creation of the line primitive
}
```

Figure 5: Hedgehog shader main function

3.6 Gradient oriented line segments (Hedgehogs) and the geometry shader

This method uses the optional geometry shading stage to add additional vertices at some user defined small distance from the original data point so that a line segment is created. The direction in which the line should be oriented is given by the gradient of the scalar field at the data point. In order not to clutter the whole region with lines only data within some user specified range is used. This culling of data is done before sending it to the graphics card, a culling in the geometry shader was tested and discarded because of performance reasons. The line segments are colored so that the vertex at the data origin has some static color (in this case dark gray) and the added vertex is given a color according to the color map in use. This gives a sense of depth that is lost for example if the data should be rendered as points only. In Figure 5 the main function for the geometry shader can be seen. As input from the vertex shader it takes a struct, vdata, that contains all vertex information for a single vertex.

3.7 Slices using texture mapping

This method visualizes the data by mapping single layers to a texture. Since the data is all equidistant on a cuboid domain it is rather trivial. A quadrilateral with its four corners at the edges of the domain and the depth position along one of the three axes of the coordinate system is created. The vertices at the corners are assigned an associated texture coordinate. A slice of data is then extracted from the dataset and mapped to a texture that is sent to the shaders.

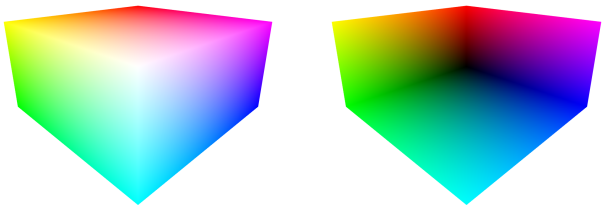
3.8 Maximum Intensity Projection (MIP)

This method is based on the premise that we want to set the value of a pixel on the screen to the value of the highest valued voxel that this pixel covers. It is performed by having three separate rendering passes. First two passes are made in which the bounding box of the data is rendered as a color coded cuboid in which each corner is given the color of its normalized three dimensional coordinate in object space. For example, the vertex at position (0,0,0) is colored black and the vertex at position (1,0,0) is colored red. The first pass is then done by simply rendering this bounding box to a buffer object, Figure 6a. The second is done by first culling all front facing faces so that only the inside of the cuboid is visible and then render it to another buffer object, Figure 6b. The final pass is done by binding these two buffers as two textures together with the entire data set which is bound as a volumetric texture. All these can then be accessed in the fragment shader where for each fragment an entry position on the bounding box is found as the color of the position of that fragment in the front buffer texture and the exit position is found as the color in the back buffer. From these two a direction from entry to exit can be calculated and the data texture can be sampled, beginning at the entry location and then moving incrementally through it in the calculated direction. For each increment a value from the data texture is read and if it is higher then the current highest value it replaces that value. The incrementation is made until we reach the exit location. The number of steps taken when incrementing through the volume is of importance since too few will lead to undersampling of the data which may create rendering artifacts, however these operations are not free and too many increments will lead to poor rendering performance. An advantage of using this method is that it needs no preprocessing of the data, it just has to be sent to the GPU packed as a 3D texture. A disadvantage is that it is highly dependent on the domain being cuboid and the data points being equidistant. Should this not be the case the data would need to be resampled and interpolated in order for this method to be valid.

4 Results

4.1 Specific Code Improvements

The areas investigated for improvement, as well as the rationale behind their selection, are described in detail in the following. Most of the concepts are generally applicable to CUDA C programming, while their importance and expected results might vary between applications.



(a) Front buffer (b) Back buffer

Figure 6: The buffer objects used in the Maximum Intensity Projection method

	4 GPUs	4 + 4 GPUs	4 + 4 + 2 GPUs
Single	0.40s	0.38s	0.36s
Double	0.46s	0.68s	0.65s

Table 1: Single versus double precision execution times. The configurations correspond to utilizing one dedicated computational node, both, or both as well as the main node.

4.1.1 Precision

When optimizing applications for deployment on GPUs, one of the first areas worth examination is whether the calculations can be performed in single precision. In comparison to traditional CPU architectures, the focus on simple cores means that GPUs generally suffer a significantly higher performance penalty when double precision floating point numbers are involved. Thus, if the accuracy provided by single precision floating point numbers is sufficient for an application, substantial speedups can be realized.

In order to facilitate comparisons, the code was adapted to allow for the flexibility of switching between precisions through changing a global definition. Experiments using single precision yielded relative errors in the L^2 norm of the order 10^{-7}, which was deemed sufficiently low that modeling errors would dominate. Thus, the use of single precision was considered appropriate for subsequent experiments.

The execution times for seven iterations of the finite difference solver are summarized in Table 1, using three different launch configurations.

4.1.2 Conversion

Despite the previous result showing some promise for performance improvements using single precision, the difference was still slight compared to what can be expected. Inquiry into the generated PTX code yielded additional insight as to why this was the case. By simply adapting the existing code to work on single precision numbers a significant amount of calculations executed faster, but some areas of the code were generating seemingly unnecessary conversion instructions between single and double precision numbers. The source of this inefficiency was the fact that many calculations were utilizing hard-coded literal constants, which were given as double precision numbers.

In addition to the unnecessary conversion, the code was also duplicated in

	4 GPUs	4 + 4 GPUs	4 + 4 + 2 GPUs
Single	0.21s	0.29s	0.34s
Double	0.40s	0.53s	0.66s

Table 2: Execution times without conversion instructions, using restricted pointers and constant precalculation. The configurations correspond to utilizing one dedicated computational node, both, or both as well as the main node.

a way that complicated maintenance. Thus, along with performing explicit type conversions of the literal constants at compile time to adapt them to the currently used precision, an effort was made to expose the pattern behind the calculations. This resulted in function definitions using C++ templates, reducing the code base by over 300 lines, as well as making it decidedly more maintainable. Together with the adoption of restricted pointers as well as the precalculation of a constant otherwise separately calculated in each thread, a new version of the code was created. The execution times are listed in table 2, using similar launch configurations as previously.

4.1.3 Kernel launch configurations

With the above modifications, the most obvious performance improvements were exhausted, and interprocess communication costs were becoming increasingly noticeable. An area where general code improvements were still possible, however, was that of kernel launch configurations, i.e. the grid and block sizes given in each GPU kernel launch.

An initially seductive method of selecting these parameters is to adapt them to the computational domain, and have, for instance, blocks representing entire slices in the xy-plane. This ensures that there is no requirement for bound checks in the kernels, and that the mapping from thread to point in the computational domain is trivial. Thus, performance benefits can be attained, as otherwise costly computations can be neglected.

The problem this creates, however, is that it puts restrictions on the computational domain due to hardware restrictions in block size. Attempting to use the program on a larger domain thus lead to incorrect behavior. Simply adding the necessary bound checks and mapping computations, however, led to severe reductions in performance.

The solution to this problem was found by retaining the connection between thread blocks and xy-plane slices, but allowing several blocks to represent the same slice. Thus, the program was modified to execute correctly for general problem domain sizes without performance reductions. The final execution times are listed in Table 3.

Figure 7 shows the attained speedups of the final version. The reference CPU implementation was provided by supervisors, and is the version on which the GPU program was based.

	4 GPUs	4 + 4 GPUs	4 + 4 + 2 GPUs
Single	0.20s	0.29s	0.35s
Double	0.41s	0.56s	0.68s

Table 3: Final execution times. The configurations correspond to utilizing one dedicated computational node, both, or both as well as the main node.

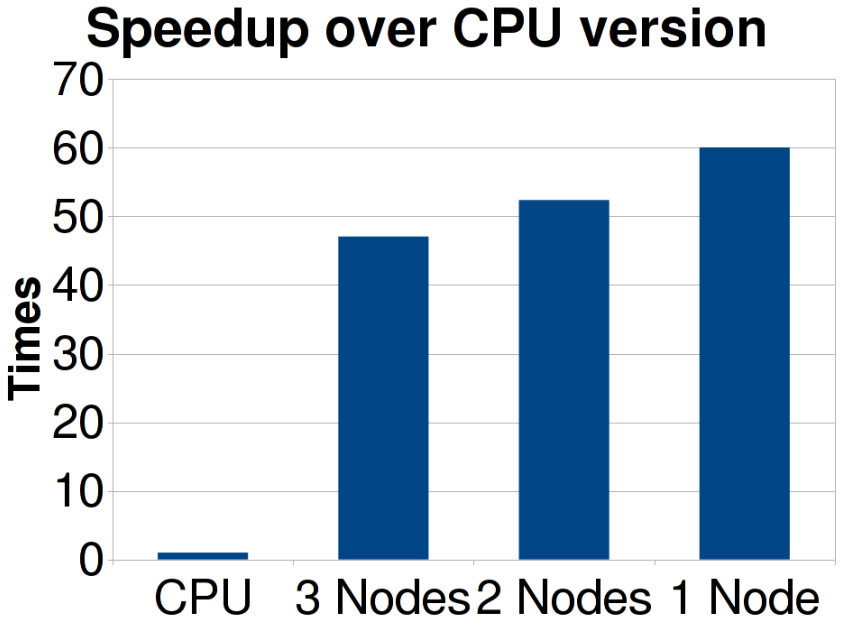


Figure 7: Final version speedups over CPU version

	4 GPUs	4 + 4 GPUs	4 + 4 + 2 GPUs
Single	0.96s	1.02s	1.19s
Double	-	2.31s	2.50s

Table 4: Execution times of final version using doubled computational domain. The configurations correspond to utilizing one dedicated computational node, both, or both as well as the main node.

4.1.4 Domain Size

In order to examine the behavior of the program as the size of the computational domain increases, the final version was run on a system where all dimensions had been doubled in size. With this size, running the program in double precision on only one node using 4 GPUs was impossible, due to the limited memory size of each GPU, and thus double precision data is only available for the launch configurations with two dedicated computational nodes, as well as two dedicated computational nodes and the main node. The results are given in Table 4.

4.1.5 Shared Memory

A typically very effective optimization available to programmers is adapting the implemented algorithm to make use of block shared memory. Having several threads in the same block cooperate, either by loading elements for common use from slower memory or by using the shared memory as temporary storage for partial results used by other threads in the block, tends to produce marked speedups.

In this application, as well, it would be reasonable to expect performance gains, as the pattern for threads to cooperate in loading elements from global memory is trivial. So far, no efficient implementation for this was found, however.

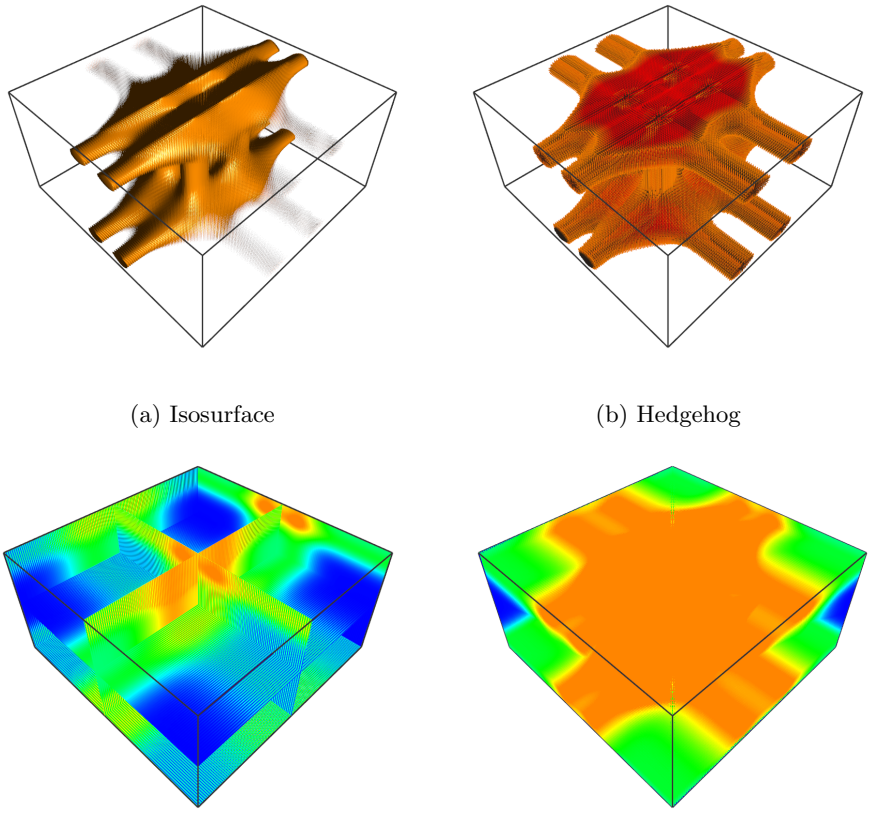
4.2 Visualization results

The isosurface extraction and rendering is valid for the wave dataset but performs very poorly due to the small scale variations present. This causes the algorithm to produce a large amount of triangles, at a scale of 10^6, which is bad both from a computational standpoint as it requires a lot of memory and increases rendering times and from a visual standpoint as the final resulting image becomes very fragmented as can be seen in Figure 8a.

The hedgehogs, Figure 8b, give overall good results in that they, in spite of their simple nature, actually convey the properties of the underlying "surface" pretty well. They perform well compared to the isosurface method in the sense that they do not introduce any excess rendering primitives, however the small scale variations are not visible for this method either.

The slices, Figure 8c, work rather well though care needs to be taken as to where the planes cut the data. Because of the small scale variations it is probably the best choice for the wave data since this is the only method that is able to accurately represent this feature within the data.

Static images created by the MIP method do not represent the three dimensional properties of the underlying data very well. Interactive rendering or a



(a) Isosurface

(b) Hedgehog

(c) Slices

(d) MIP

Figure 8: Visualization results

sequence of images is preferable but even then it may be difficult to interpret, Figure 8d.

It is also possible to combine methods and it was found that a combination of slices and hedgehogs will likely give the best overall result. The slices will show the small scale variations and the hedgehog the larger overall three dimensional shape, Figure 9.

5 Conclusions

5.1 Execution Configuration Runtimes

The results might seem paradoxical at first glance, since in most cases the utilization of more hardware does not improve performance, but indeed hampers it. The explanation lies in inter-node communication and the fact that the system has non-uniform memory access. Since running the program on several nodes means that communication is performed over the comparatively slow Ethernet connection, it is hardly surprising that execution times suffer. As the

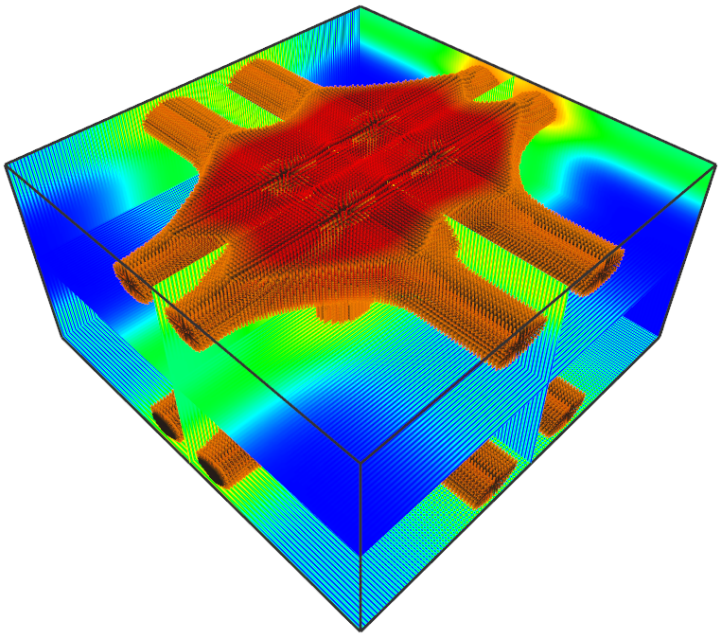


Figure 9: Visualization using a combination of one hedgehog and several slices.

computational domain grows, however, it seems that this trend might reverse, and additional hardware will yield the expected performance benefits. Further experiments on larger systems is required for this to be a valid conclusion, however.

5.2 Shared Memory

A potential topic for discussion is why the common shared memory optimization was seemingly ineffectual. The most probable explanation is due to the shared memory also being able to be utilized as an L1 cache. Since this option produces better results in this case, it seems likely that the access pattern of the threads already produces a sufficiently efficient utilization of the shared memory/L1 cache resource. This is one of the issues that would need to be carefully monitored as the problem size is increased, and as the hardware setup changes. It is not unlikely that the situation could arise where adapting the code to make use of this resource becomes highly beneficial.

5.3 Discussion Regarding Visualization

All methods have their respective pros and cons. The isosurface method has the highest requirements in both implementation and computational resources. The hedgehogs do not inquire any high computational costs and the result shows important large scale properties. However it actually misses and conceals the small scale variation property. The slices may be the best overall method in that it is easy to implement and gives results that do not miss any of the data it is supposed to represent. It also acts as a sort of region of interest method

in that the user may choose slices that only lie in some interesting regions. However care must be taken when choosing the slices so that important regions are not missed. The MIP method gives rather unclear results that may be hard to interpret. The lack of shading, occlusion or any other depth cue when presenting it as a static image limits its usability. The isosurface, hedgehog, and slice methods can be combined in order to obtain various properties. In particular, combining slices and hedgehogs seemingly produces the best overall visual result, Figure 9.

5.4 Continuation and suggested future work

5.4.1 Domain Division

One area with the potential for improvements is the division of the computational domain between nodes. The current setup has the advantage of requiring no communication in the y and z dimensions. As the number of datapoints in these directions increases, however, the limited amount of memory on each device will mean that the width of the slice allocated to each device will shrink.

Let D_k denote the number of elements of the computational grid in the k dimension. Then the current configuration yields communication on the borders between each slice equal to the depth of communication required, i.e. three elements, multiplied by the interface between the slices $D_y D_z$ multiplied by the amount of interfaces, two. Finally the amount of slices required is D_x/w where w designates the width in the x-dimension of each slice. The total amount of communicated elements is thus $\frac{6D_x D_y D_z}{w}$. As the aforementioned decline in width of each slice progresses, the amount of communicated elements has the potential to become problematic.

In comparison, consider splitting the computational domain into cubes with side a allocated to each device. Assuming that this side length is less than each dimension of the computational domain, this requires communication along all six sides of each cube. The communication is once again three elements deep, and each cube thus needs to receive $36\dot{a}^2$ elements. The amount of cubes in this case would be, assuming each dimension is divisible by a, $\frac{D_x D_y D_z}{a^3}$, and thus the total amount of communicated elements is $\frac{18 D_x D_y D_z}{a}$. Since the amount of elements allocated to each device has to be constant in both cases, i.e. $D_y D_z w = a^3$, as the y and z dimensions of the computational grid increase, w will decrease below $\frac{1}{3}a$. At this point, according to the above formulas for the amount of communicated elements, the cube solution will be preferable, and beyond that point the potential gains increase steadily.

This solution is not without its difficulties, as communication needs to be extended to the y and z dimensions, and thus the amount of neighbors increases from the current two to six. Despite this however, as the domain grows, this will go from worthwhile to imperative.

5.4.2 Hardware Setup

The current solution of using a common Ethernet switch for communication is inefficient, especially considering the fact that all nodes currently communicate simultaneously and in a synchronized manner. One solution for this is to investigate the possibility of staggering communications and calculations so as to

avoid congestion.

With the current communication pattern, there is another potentially interesting setup that could be investigated. Since communication is only performed between neighboring processes, letting hardware mirror this, and connecting machines in a ring formation would allow for point-to-point communication. This means that the overhead of congestion would be alleviated, and presumably should provide for lower latency. It would be particularly interesting to examine the potential of using other standards for communication such as FireWire.

5.4.3 Visualization

All implementations of the visualization methods could benefit from some redesigns or additional features. No real optimization has been performed on any of the methods though the only one that probably would truly benefit from this is the isosurface extraction and then mostly at the triangle creation stage. The Marching Cubes algorithm is parallel in its nature but this implementation does not exploit this, a parallel implementation either on CPU using e.g. OpenMP or by using GPGPU e.g. CUDA could probably give a significant speedup when creating the triangles for the isosurface. For this method to be usable in an interactive environment where the data changes over time and new isosurfaces would have to be continuously created this optimization is close to a requirement. This would however not change the poor visual results for the method.

Improvements to the hedgehog method could be adding an optional illumination model similar to the one for the isosurface rendering. This could give stronger depth cues and additional gradient orientation information to the user. It could however give a poorer representation of the data since the shading would change the color of the line segments.

The slices could probably benefit from an implementation where the whole dataset is loaded into graphics memory as a 3D texture instead of, as in the current implementation, having an extraction of slices and packing them in 2D textures in a preprocessing stage on the CPU. This would make it easier to implement more flexible slices that do not need to be aligned to one of the three main axes. This method could easily be adapted to work on other grid types than the equidistand cartesian grid. Instead of creating quadrilaterals that cut through the data, all points on the grid in a slice would be given a vertex and a texture coordinate corresponding to its ordered position in the dataset.

The MIP method implemented could possibly be extended to work for other more general convex bounding geometries than just the cuboid. The method itself could be altered slightly in order to produce images that correspond to the ones created by the isosurface method. Instead of recording the maximum value encountered when moving through the data it could move until it encountered some threshold value, corresponding to the isosurface value, and record this distance. A location of this intersection could then be calculated in world space so that light calculations could be made for the fragment in the same manner as when shading the isosurface. The surface normal could be calculated either in a preprocessing stage by calculating the gradients for the whole data set, as in the isosurface and hedgehog method, or it could be calculated in the shader given information about step length and grid layout. This extension to the method would add a much needed depth cue and could be a true substitute to the

isosurface since that method performed so poorly. Other volumetric illumination techniques could also be considered, e.g. a 2D postprocessing method like the one discussed at chapter 7.8 in [3]. Due to the manner in which this method is implemented, it is not trivial to combine it with any of the other methods. However, by performing the changes discussed and writing manually to the depthbuffer the location at which the value was found it would be possible to use it together with any of the other methods.

References

[1] H. E. C. William E. Lorensen, "Marching cubes: A high resolution 3d surface construction algorithm," *Computer Graphics*, vol. 21, pp. 163–169, July 1987.

[2] D. S. Edward Angel, *Interactive Computer Graphics: A Top Down Approach Using Shader Based OpenGL*. Pearson, 2012. Chapter 5.

[3] B. L. Will Scroeder, Ken Martin, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Kitware, 4 ed., 2006. p.234-239.

[4] NVIDIA Corporation
CUDA C Programming Guide
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
Last reviewed January 20, 2014

[5] NVIDIA Corporation
CUDA C Best Practices Guide
http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
Last reviewed January 20, 2014

[6] NVIDIA Corporation
Parallel Thread Execution ISA
http://docs.nvidia.com/cuda/parallel-thread-execution/index.html
Last reviewed January 20, 2014