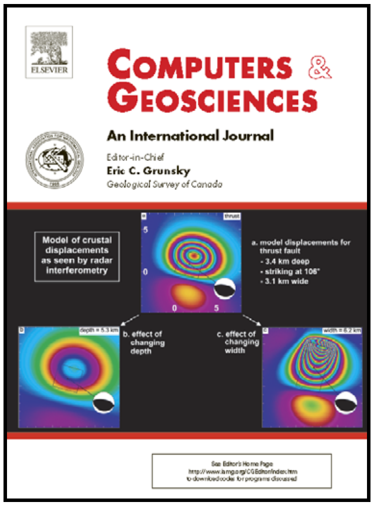


www.elsevier.com/locate/cageo

Massively parallel forward modeling of scalar and tensor gravimetry data

M. Moorkamp, M. Jegen, A. Roberts, R. Hobbs

PII: S0098-3004(10)00057-9
DOI: doi:10.1016/j.cageo.2009.09.018
Reference: CAGEO 2305

To appear in: *Computers & Geosciences*

Received date: 16 June 2009
Revised date: 2 September 2009
Accepted date: 4 September 2009

Cite this article as: M. Moorkamp, M. Jegen, A. Roberts and R. Hobbs, Massively parallel forward modeling of scalar and tensor gravimetry data, *Computers & Geosciences*, doi:10.1016/j.cageo.2009.09.018

Massively parallel forward modeling of scalar and tensor gravimetry data[☆]

M. Moorkamp[*,a], M. Jegen[a], A. Roberts[b], R. Hobbs[b]

[a] *IFM-GEOMAR, Wischhofstrasse 1-3, 24148 Kiel, Germany*
[b] *University of Durham, Department of Earth Sciences, South Road, Durham DH1 3LE, UK*

Abstract

We present an approach to calculate scalar and tensor gravity utilizing the massively parallel architecture of consumer graphics cards. Our parametrization is based on rectilinear blocks with constant density within each blocks. This type of parametrization is well suited for inversion of gravity data or joint inversion with other datasets, but requires the calculation of a large number of model blocks for complex geometries. For models exceeding 10,000 cells we achieve an acceleration of a factor of 40 for scalar data and 30 for tensor data compared to a single thread on the CPU. This significant acceleration allows fast computation of large models exceeding 10^6 model parameters and thousands of measurement sites.

Key words: gravity modeling, cuda, parallel computing

[☆] Code available from server at http://www.iamg.org/CGEditor/index.htm

[*] Corresponding author

Email address: `mmoorkamp@ifm-geomar.de` (M. Moorkamp)

1. Introduction

Driven by the computer games industry graphics cards (GPUs) have evolved into powerful computing devices that are geared towards a large number of simultaneous calculations and high memory bandwidth (e.g. Ryoo et al., 2008). In an attempt to broaden the scope of their products, the two main consumer graphics cards manufacturers, Nvidia and AMD, have released programming interfaces for general purpose calculations to their cards. So far massively parallel architectures were limited to specialized and costly hardware. With these developments such an architecture becomes available at low prices and makes the development of massively parallel algorithms attractive.

The success of solving a numerical problem on a massively parallel architecture depends heavily on the anatomy of the algorithm. If the problem can be split into independent parts that can be solved without having to transfer information, parallelization is easy and we can expect good performance. If conversely results have to be distributed globally during the calculation, parallelization becomes difficult and special care has to be taken to reduce the amount of synchronization between the parallel threads of the program. The challenge for GPU based computations is that the number of threads has to be on the order of 10,000 or more to utilize the full computing power of the architecture (Nickolls et al., 2008; Ryoo et al., 2008; Jeong and Whitaker, 2008; Komatitsch et al., 2009).

Modeling gravitational acceleration and its spatial derivatives is a common tool in geophysics to test models of the density distribution within the subsurface. Often tectonic information or seismic models are used to define broad geological structures with a common density and these are then parametrized as polygonal bodies within the numerical modeling scheme (e.g. Götze and Lahmeyer, 1988). This type of approach has the advantage that the number of bodies is kept low even for complex models which makes it easy for the user to construct such a model and reduces the number of function evaluations.

Our forward modeling approach is geared towards usage within a joint inversion algorithm that combines gravity, seismic and magnetotelluric data (Heincke et al., 2006) and therefore we parametrize our model in terms of rectilinear blocks (Hobbs and Trinks, 2005). This type of setup is also often used for inversion of gravity data alone (e.g. Li and Oldenburg, 1998; Portniaguine and Zhdanov, 1999; Nagihara and Hall, 2001; Chasseriau and Chouteau, 2003) and has the advantage that the equations for scalar and tensor gravimetry are particularly simple, but requires the calculation of the effect of a large number of blocks, as complex geometries have to be constructed from many small blocks. On a platform with no or only a low degree of parallelism this leads to increased computational times compared to the polygonal parametrization. However, the calculation of the effect of many rectilinear block can be performed effectively on a massively parallel architecture to compensate for the higher computational cost. This cost becomes particularly relevant when we have to calculate several large models for which we cannot store the sensitivities in main memory or even on disk, for example within a non-linear inversion.

Although gravity forward modeling is generally fast compared to other methods and we restrict ourselves here to Nvidia's CUDA interface the conclusions and strategies for this relatively simple problem can be applied to other problems and other massively parallel architectures. Before we describe the details of our implementation we will discuss the basic equations of the gravimetry problem for rectilinear blocks. We will then show the performance of our approach for a number of scenarios and discuss the implications for forward modeling and inversion of gravimetric data.

2. Basic equations

The two quantities that are mainly used in gravimetry surveys, are the vertical gravitational acceleration U_z, i.e. the vertical derivative of the gravitational potential U and the gravitational tensor Γ, i.e. the tensor of second spatial derivatives,

$$\Gamma = \begin{pmatrix} U_{xx} & U_{xy} & U_{xz} \\ U_{yx} & U_{yy} & U_{yz} \\ U_{zx} & U_{zy} & U_{zz} \end{pmatrix}. \tag{1}$$

With the nomenclature shown in Figure 1 the equation for the effect of a single prism of density ρ on the vertical gravitational acceleration U_z is (Li and Chouteau, 1998)

$$U_z = -\gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \left(x_i \ln(y_j + r_{ijk}) + y_j \ln(x_i + r_{ijk}) + z_k \arctan\frac{z_k r_{ijk}}{x_i y_j} \right), \tag{2}$$

and for two elements of the gravimetry tensor it is

$$U_{xx} = \gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \arctan\frac{y_j z_k}{x_i r_{ijk}}, \tag{3}$$

$$U_{xy} = -\gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \ln(z_k + r_{ijk}), \tag{4}$$

where

$$\begin{aligned} x_i &= x - \xi_i \qquad y_j = y - \eta_j \qquad z_k = z - \zeta_k \\ r_{ijk} &= \sqrt{x_i^2 + y_j^2 + z_k^2} \\ \mu_{ijk} &= (-1)^i(-1)^j(-1)^k. \end{aligned}$$

We can calculate all other elements of the gravimetry tensor by permutation of the coordinate axes (e.g. Li and Chouteau, 1998; Nagy et al., 2000), in addition the tensor is symmetric so that we only have to calculate 6 instead of all 9 tensor elements. Theoretically, we even only have to calculate 5 elements, as the diagonal terms of the tensor are related by Poisson's equation

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \frac{\partial^2 U}{\partial z^2} = -4\pi\gamma\rho. \tag{5}$$

However, we calculate all three diagonal elements independently as this gives us an indication of the numerical precision of the results.

Scalar and tensor gravity calculation are well known linear problems and therefore in both cases a term that is purely determined by the geometry of the cell is multiplied by the density of the cell (e.g. Nagy et al., 2000). Also, the effect of several prisms is simply the sum of the contributions of a single cell. We can therefore write the forward calculation as a vector-matrix multiplication between the model vector of density values $\mathbf{m}$ and the geometric sensitivities G

$$\mathbf{d} = G\mathbf{m}. \tag{6}$$

Here each row of G corresponds to one observed quantity, i.e. a measurement of the vertical acceleration or an element of the gravimetric tensor. The resulting data vector $\mathbf{d}$ contains the data resulting from the model. We therefore have two parts in the calculation of the forward problem, 1) the calculation of the elements of G and 2) the evaluation of the matrix vector product.

3. Implementation

Before we describe the details of our implementation we have to clarify the standard nomenclature for the CUDA interface and briefly explain the architecture. A function that can be executed on the GPU is called a kernel and is described by the extended C-syntax `kernelname<<<dimGrid,dimBlock>>>(Parameters)`. Here `dimGrid` and `dimBlock` are variables that describe the number of independent thread blocks in the computing grid and the number of threads in each block, respectively (see Figure 2). The number of threads in a single block is determined by the specifications of the GPU and is typically between 64 and 512 to optimize memory access by the hardware (nvidia, 2009). In principle different threads within a block can share information, but we will not use this feature in our implementation. The size of the grid depends on the size of the problem, in our case the number of model parameters M, and each block can be computed independently and in any order. During the parallel execution of the kernel the implementation determines the sub-problem to work on from the two variables `blockIdx` and `blockDim`. The values of these variables is set by the GPU depending on the current block index and thread index for the calculation. In principle this index can have several dimensions, we only use the first dimension `blockIdx.x` and `blockDim.x`, respectively.

As we can calculate each element of the sensitivity matrix independently and with relatively few input parameters, this part can be performed very efficiently. We parallelize over the number of grid cells M, i.e. a single row of the matrix G. In principle, it would be possible to also parallelize over the number of measurements N to obtain $N * M$ independent threads. However, for large models, for which the parallelization makes most sense, M already exceeds one million or more and therefore we can utilize the threading capabilities of all currently available GPUs. By only parallelizing over the grid cells, we avoid additional administrative overhead and also avoid having to store the full sensitivity matrix if we do not need it, instead we only have to store a single row at a time. The following listing shows the core algorithm using NVidia's CUDA API.

```
__global__ void CalcScalarMeas(const double x_meas, const double y_meas,
    const double z_meas, const double *XCoord, const double *YCoord,
    const double *ZCoord, const double *XSizes, const double *YSizes,
    const double *ZSizes, const int nx, const int ny, const int nz,
    double *Grow)

//calculate memory offset from execution parameters
const unsigned int offset = blockIdx.x * blockDim.x + threadIdx.x;
int xindex, yindex, zindex;
//if the offset is within the model size
if (offset < nx * ny * nz)
 {
   //calculate the coordinate indices for all three directions
   OffsetToIndex(offset, ny, nz, xindex, yindex, zindex);
```

```
   //calculate and assign the geometric term to the
   //row of the sensitivity matrix
   Grow[offset] = CalcGravBoxTerm(x_meas, y_meas, z_meas,
       XCoord[xindex], YCoord[yindex], ZCoord[zindex], XSizes[xindex],
       YSizes[yindex], ZSizes[zindex]);

 }
```

We generate the storage offset for the results within the current row of the sensitivity matrix from the built-in variables `blockId.x`, `blockDim.x` and `threadId.x`. As mentioned above, the values of these variables are set by the hardware for each executed thread. Therefore each offset is unique within one calculation of the sensitivities. The optimum number of blocks `blockDim.x` depends on the register use and the ability to load data from global memory to local memory in a coalesced fashion. The CUDA programming guide (nvidia, 2009) recommends a minimum number of 64 blocks or a multiple of this number. We will investigate the impact of the block size in the performance section. Depending on the block size and the model size, we might have some extra threads in the last block for which we do not need to perform any calculations. We therefore have to check whether the offset is smaller than the dimension of the model `nx*ny*nz`.

If the current thread is active, we calculate the indices of the current cell in x-direction, y-direction and z-direction, respectively, from the offset and the total size of the model in y-direction and z-direction. The function `CalcGravBoxTerm` is a straightforward implementation of the geometric term in Equation 2 and takes the three components of the measurement position, the three coordinates of the upper left front corner of the current cell and the sizes of the cell in the three coordinate directions as arguments. After the API has executed the above code we have obtained a single row of the sensitivity matrix.

The further computational strategy depends on the context in which the calculation is performed. For pure forward modeling the most efficient approach is to perform a scalar multiplication between the current row of the sensitivity matrix and the vector of densities on the GPU to obtain the current datum and then discard the sensitivity information. In this case we minimize both the storage requirements and the number of transfers between the memory of the GPU and the main memory. In an inversion context however it is beneficial to store the sensitivity matrix, if possible, for two reasons. First, as long as the geometry does not change we can calculate the data for models with varying density distributions by a matrix-vector product as shown in Equation 6. We will show the acceleration we can achieve with this below. Second, we can use the sensitivity matrix to perform Gauss-Newton type inversion. We therefore always transfer the current row of the sensitivity matrix from the GPU to main memory, then perform the scalar product on the CPU and let the main application decide whether this row should be stored for later use or discarded. In the performance section we will assess the cost of the additional transfers.

The implementation for the gravimetric tensor is similar to the scalar implementation. We only have to replace the calculation of the geometric term with the appropriate mathematical expressions and preserve the 6 independent rows of the sensitivity matrix when copying from the GPU to main memory.

4. Performance

In this section we demonstrate the performance gain we can achieve with our GPU based implementation. All tests were run on a Intel Q6600 with 2.4GHz, 4GB of main memory and a NVidia GTX260 graphics card which has 192 processor cores and 896 MB of onboard memory with a bandwidth of 111.9 GB/s. This is the cheapest graphics card that can handle double precision computations that we use throughout the comparison and is readily available in standard consumer PCs.

We compiled the main code with the GNU compiler collection version 4.3.3 under Ubuntu 09/04 using the "-O3" optimization flag and the GNU openmp implementation. For the CUDA code we used NVidia's `nvcc` in Version 2.1 with the driver version 180.44. In all cases we average over 5 independent runs to obtain the calculation time. In each run we use a different density model where each cell of the model is randomly assigned a density between 0.1 – 3.0 g/cm^3 and the cell sizes randomly vary between 1 and 11 km.

First, we examine the impact of the execution block size on the performance. For three different model sizes we vary the number of threads per execution block between 64 and 256. In Figure 3 we plot the time relative to the fastest run for each model size in order to make the results for the three model sizes comparable. For the chosen model and block sizes we observe that the performance varies by only 17% between the fastest and the slowest configuration. Depending on the model size 64, 128 or 256 threads per block result in the highest performance. Between these three configurations the maximum difference in performance is only 6%. We therefore choose a block size of 128 for all subsequent experiments and do not attempt to optimize this value.

Figure 4 shows the computation time for varying model sizes between 8 and 1 million model cells and 30 stations for computation of scalar gravity data on one CPU core, 4 CPU cores and the graphics card, respectively. To illustrate the benefits of storing the sensitivity matrix for later computations we also show the time it takes to evaluate the matrix vector product using the ATLAS linear algebra library (Whaley et al., 2001).

As expected, for a single core of the CPU the time increases linearly with model size. There is very little overhead to the computation and profiling shows that most time is spent evaluating the trigonometric and natural logarithm functions in Equation 2. When using all 4 cores of the CPU we observe that for models with less than 1,000 model cells there is some administrative overhead associated with the parallelization. For larger models, however, we achieve the same linear increase with model size. For these large models the acceleration compared to a single core is close to the theoretical maximum of a factor of 4. This demonstrates that the problem can be efficiently parallelized for multi-core architectures.

The curve for the GPU based computations shows some interesting behavior. For models with less than 100 cells the computation time is higher than for both CPU based calculations. This demonstrates the overhead associated with initializing the GPU and transferring data between main memory and the memory of the graphics card. Furthermore, for less than 3,000 simultaneous threads the calculation time is independent of model size illustrating the massively parallel architecture. For fewer than a few thousand model parameters we do not utilize all available computing units on the card. For more than 10,000 parameters we again achieve a linear dependency of computation time on the model size. Within the linear domain the acceleration compared to a single core of the CPU is approximately a factor of 40. This is a significant increase in performance that allows to calculate the response of large models within a a few seconds. In our case the number of measurement sites is relatively low and therefore even the calculation time of 70 s at 10^6 model parameters for the single CPU core is not problematic, for large surveys with hundreds of sites however the acceleration provided by the GPU marks an important step.

Our performance comparison also shows the time for calculations with pre-computed sensitivities as it could be done within a non-linear inversion, e.g. when combining gravity with other data (Heincke et al., 2006). Given enough RAM we only have to perform the full computation in the first iteration and can then benefit from the accelerated evaluation with the atlas library. In this case the acceleration factor is 1,000 for large models. This makes the calculation of the model response essentially instantaneous, but requires large amounts of memory. The storage of the sensitivity matrix in double precision requires $8 \times N \times M$ bytes which corresponds to about 240 MB for our largest test case, but exceeds the memory of current computers for larger models or more measurement sites.

The graph for the full tensor calculation in Figure 5 shows the same general behavior as for the scalar data. Although we now calculate 6 elements of the tensor, the calculation time only increases by a factor of two compared to the scalar data. The reason for this is the simpler structure of the equations for the elements of the tensor. This result also shows that the calculations on the CPUs are essentially dominated by the evaluation of the mathematical functions and not by memory transfers. As before we observe a nearly linear increase of the calculation time with model size for the calculations with one processor and, apart from some overhead for small models, also for four processors.

The transition from constant calculation time to linear increase for the GPU calculation again occurs at a model size of 3,000 parameters. This is because we calculate the 6 elements of the tensor in strictly serial order. The structure of the calculation in terms of parallelization is therefore the same as for the scalar case. The acceleration through the GPU for the tensor case is a factor of 30 compared to 1 processor of the CPU. Due to the simpler structure of the equations and the larger amount of data we have to transfer, the acceleration is not quite as high in this case as for the scalar case, but still significant.

As the FTG calculations require the most transfers of sensitivity information between the GPU and general memory, we use these calculations to assess the cost of the memory transfers. For each independent element of the gravimetric tensor, we transfer a row of the sensitivity matrix from the GPU to the CPU. Profiling shows that for models with 10^6 parameters the code only spends 1% of its time for these memory transfers and this behavior is therefore not critical for the performance.

Finally, we examine the numerical precision of the results. Figure 6 shows a histogram of the relative difference between the results from the CPU and the GPU for FTG calculations with 5 random models with 30 sites each. The histogram shows a clear peak around zero with most values concentrated between $-5\cdot10^{-13}$ and $5\cdot10^{-13}$, the minimum and maximum relative difference are $-5\cdot10^{-10}$ and $1\cdot10^{-10}$, respectively. This shows that for practical purposes the results are identical. Also the trace of the tensor agrees with the theoretical value within numerical precision.

We also examine the possibility of performing the calculations in single precision on the GPU. Until recently GPUs were only capable of single precision calculations and their performance is significantly higher for this type of calculations. Compared to the double precision calculations we observe an acceleration factor of roughly 4, more than 100 times faster than calculations on the CPU. However, the numerical precision is problematic. When comparing the results to the double precision calculations in most cases the relative difference stays below $1\cdot10^{-3}$, a satisfactory value for practical purposes. However, more than 10% of the results show a relative difference of 0.1 or more, most likely due to accumulated rounding errors (Li and Chouteau, 1998). Such a difference impacts on the result of an inversion or the interpretation of a forward model and thus is not acceptable for reliable forward modeling.

5. Conclusions

The calculation of the scalar and tensorial forward response of large density models can be efficiently parallelized and accelerated by performing the calculation on a standard consumer GPU. Our tests show that it is important perform the calculations with double precision to obtain reliable results. In this case we achieve accelerations of a factor of 40 for scalar data and a factor of 30 for tensorial data with more than 3,000 model parameters, respectively. For the tested cases the number of threads per execution block has only a minor impact on the performance.

This is a significant improvement, particularly when considering the relatively low cost of these graphics cards. Our approach allows to quickly calculate the response for different density distributions as required, for example, in a joint inversion without storing sensitivity information. Although utilizing the sensitivity information accelerates the calculation further, even modern computers cannot store the sensitivity matrix for large models. Furthermore, even then we have to calculate the sensitivities once which can be performed using the GPU based algorithm.

6. Acknowledgments

We thank Colin G. Farquharson and an anonymous reviewer for constructive comments on the manuscript. This work was funded by Chevron, ExxonMobil, Nexen, RWE Dea, Shell, Statoil and Wintershall within the JIBA consortium.

References

Chasseriau, P., Chouteau, M., 2003. 3D gravity inversion using a model of parameter covariance. Journal of Applied Geophysics 52, 59–74.

Götze, H.-J., Lahmeyer, B., 1988. Application of three-dimensional interactive modeling in gravity and magnetics. Geophysics 53, 1096–1108.

Heincke, B., Jegen, M., Hobbs, R., 2006. Joint inversion of mt, gravity and seismic data applied to sub-basalt imaging. SEG Technical Program Expanded Abstracts 25 (1), 784–789.

Hobbs, R., Trinks, I., 2005. Gravity modelling based on small cells. In: Proceedings of 67th EAGE (European Association of Geoscientists and Engineers) Conference and Exhibition. Madrid, Spain, p. 347.

Jeong, W.-K., Whitaker, R. T., 2008. A fast iterative method for eikonal equations. SIAM Journal on Scientific Computing 30 (5), 2512–2534.

Komatitsch, D., Michéa, D., Erlebacher, G., 2009. Porting a high-order finite-element earthquake modeling application to nvidia graphics cards using cuda. Journal of Parallel and Distributed Computing 69 (5), 451 – 460.

Li, X., Chouteau, M., 1998. Three-dimensional gravity modeling in all space. Surveys in Geophysics 19, 339–368.

Li, Y., Oldenburg, D. W., 1998. 3-D inversion of gravity data. Geophysics 63, 109–119.

Nagihara, S., Hall, S. A., 2001. Three-dimensional gravity inversion using simulated annealing: Constraints on the diapiric roots of allochthonous salt structures. Geophysics 66, 1438–1449.

Nagy, D., Papp, G., Benedek, J., 2000. The gravitational potential and its derivatives for the prism. Journal of Geodesy 74, 552–560.

Nickolls, J., Buck, I., Garland, M., Skadron, K., 2008. Scalable parallel programming with cuda. Queue 6 (2), 40–53.

nvidia, 2009. Nvidia CUDA programming guide, Version 2.1. URL `http://www.nvidia.com/cuda, [accessed 1 December 2009]`

Portniaguine, O., Zhdanov, M. S., 1999. Focusing geophysical inversion images. Geophysics 64, 874–887.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., Hwu, W., 2008. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Salt Lake City, UT, pp. 73–82.

Whaley, C. R., Petitet, A., Dongarra, J. J., 2001. Automated empirical optimization of software and the ATLAS project. Parallel Computing 27 (1–2), 3–35.

Figure 1: Nomenclature and parameterization for gravity forward problem. Position of the measurement is described by the coordinate triple (x, y, z). Model is divided into rectilinear blocks of constant density ρ, for clarity we only show a single block. Coordinates of corners of the block can be completely described by two coordinate triples (ξ_1, η_1, ζ_1) and (ξ_2, η_2, ζ_2) for opposing corners of the block.

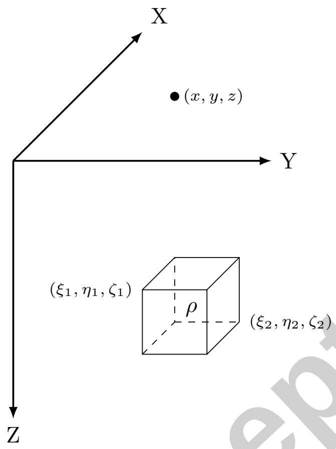
Figure 2: Overview of CUDA execution model and mapping of sensitivities. Execution grid consists of independent blocks that can be executed in any order. In turn each block consists of a number of threads. Each element of the sensitivity vector for the current measurement is mapped onto a different thread.

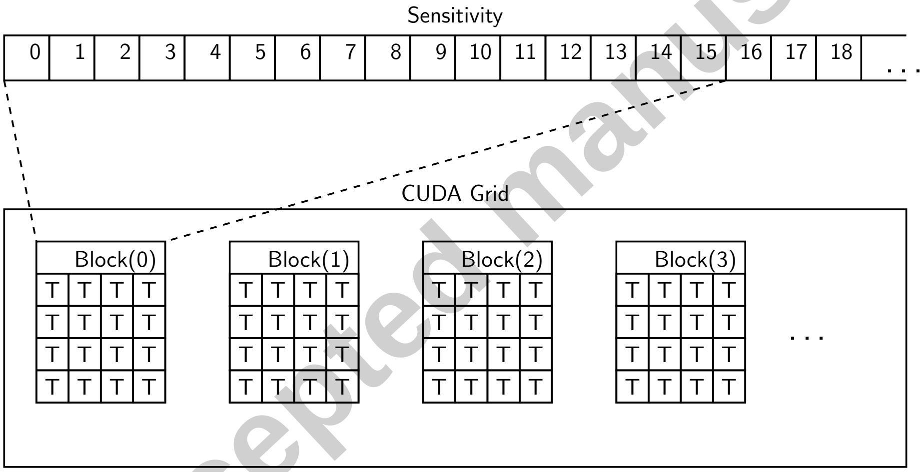
Figure 3: Dependency of execution time on number of threads per block. For each block size we measure execution time of models with $40 \times 40 \times 40$, $60 \times 60 \times 60$ and $80 \times 80 \times 80$ model cells, respectively. To make results comparable we divide by the time for the fastest execution for each model size. Execution time is relatively similar for all block sizes but shows minima at 64, 128, 192 and 256, respectively.

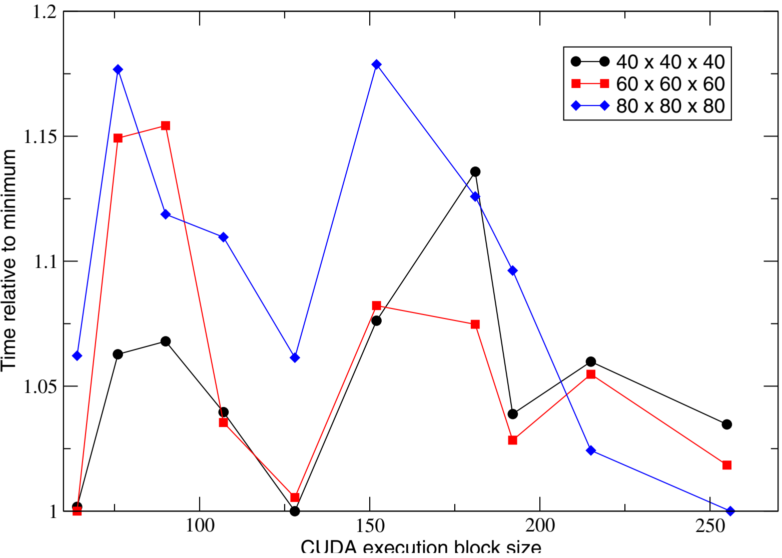
Figure 4: Calculation times for different size models for scalar gravity data for a single CPU thread (Q6600), 4 CPU threads and GPU (GTX260). For comparison we also show the time to evaluate the matrix vector product with the ATLAS library when the sensitivity matrix has been calculated.

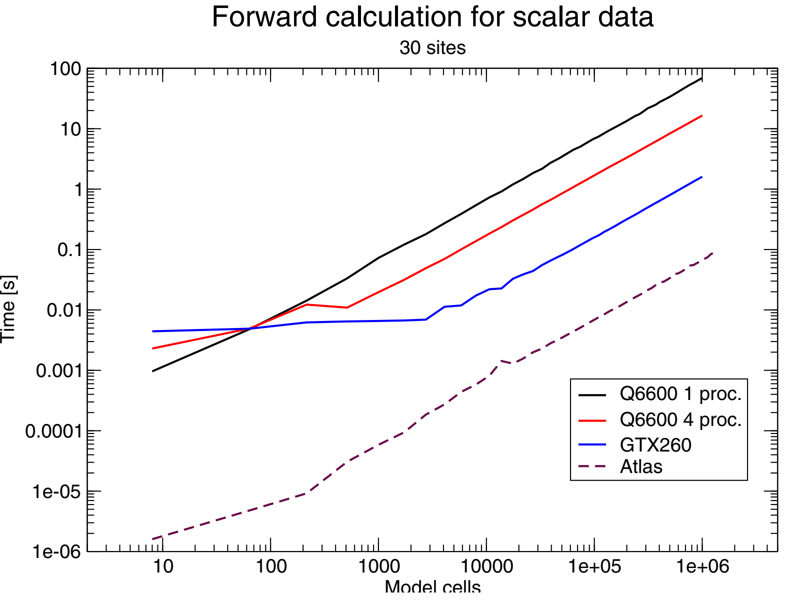
Figure 5: Calculation times for different size models for FTG data for a single CPU thread (Q6600), 4 CPU threads and GPU (GTX260).

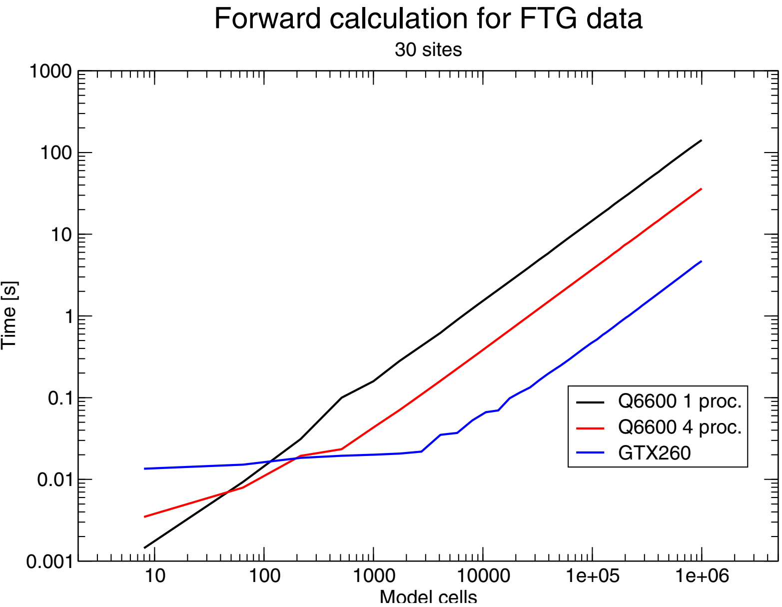
Figure 6: Relative difference between FTG calculations performed on CPU and on GPU in double precision, respectively. Maximum relative deviation between results is $-5 \cdot 10^{-10}$.

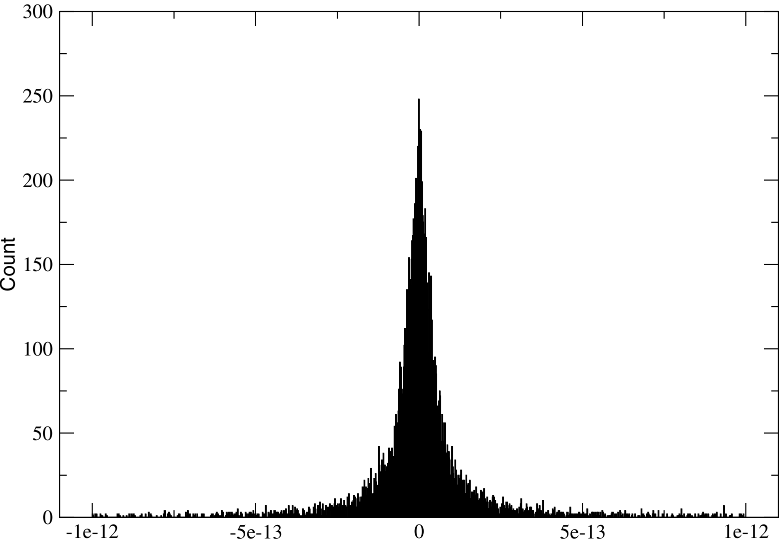

X
(x, y, z)
Y
(ξ_1, η_1, ζ_1)
ρ
(ξ_2, η_2, ζ_2)
Z


Sensitivity
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
. . .
CUDA Grid
Block(0)
Block(1)
Block(2)
Block(3)
T
. . .


1.2
1.15
1.1
1.05
1
Time relative to minimum
100
150
200
250
CUDA execution block size
40 x 40 x 40
60 x 60 x 60
80 x 80 x 80


Forward calculation for scalar data
30 sites
Time [s]
100
10
1
0.1
0.01
0.001
0.0001
1e-05
1e-06
10
100
1000
10000
1e+05
1e+06
Model cells
Q6600 1 proc.
Q6600 4 proc.
GTX260
Atlas


Forward calculation for FTG data
30 sites
Time [s]
0.001
0.01
0.1
1
10
100
1000
Model cells
10
100
1000
10000
1e+05
1e+06
Q6600 1 proc.
Q6600 4 proc.
GTX260


300
250
200
150
100
50
0
Count
-1e-12
-5e-13
0
5e-13
1e-12
Relative difference (GPU-CPU)/CPU