GPU acceleration of average gradient method for solving partial differential equations

Touko Puro [*] **Aarne Pohjonen** [**]

[*] *Aalto University, Department of Computer Science, Konemiehentie 2, 02150 Espoo (e-mail: touko.puro@aalto.fi).*
[**] *University of Oulu, Materials and mechanical engineering department, Pentti Kaiteran Katu 1 (e-mail: aarne.pohjonen@oulu.fi)*

Abstract: Previously presented method of calculating local average gradients for solving partial differential equations (PDEs) is enhanced by accelerating it with graphics processing units (GPUs) and combining a previous technique of interpolating between grid points in the calculation of the gradients instead of using interpolation to create a denser grid.
For accelerating the calculation with GPUs, we have ported the original naive Matlab implementation to C++ and CUDA, and after optimizing the code we observe a speedup factors more than two thousand, which is largely due to the original code not being optimized.

Keywords: GPU acceleration, scientific computing, numerical methods, partial differential equations, deformable grids

1. INTRODUCTION

To intelligently control the formation of material microstructure in sub-micrometer level, modern computational tools are needed. The full field models, such as the level set Hallberg (2013), phase field Steinbach and Salama (2023) and cellular automata Seppälä et al. (2023), offer capabilities to explicitly simulate the microstructure formation Pohjonen (2023). Inclusion of the relevant physical phenomena and their numerical modelling requires the solution of partial differential equations (PDEs).

There are several approaches to obtaining the numerical solution, such as the finite elements, finite differences, finite volume etc. The finite element method is perhaps the most advanced, but it's implementation is not straightforward. Finite differences in the standard implementation is limited to structured grids, which grids would be capable of solving the equations in Eulerian framework. There are approaches to simulate solid mechanics in the Eulerian meshes and they could provide certain advantages such as capability of simulating material distortions without the need of re-meshing, since the material flow through the node points can be simulated. However, more often solid mechanics simulations involving deformations are based in Lagrangian approach, which naturally describes the flow of the material and the material point dependent field variables Basaran (2008).

To simulate the movement of material points within the Lagrangian approach, and to solve the equations in the deformed grid, a triangular two-grid method Pohjonen (2024a) was previously proposed which achieved this purpose in a way which is easy to implement.

In the current work we present enhancements made to the previous version as well as the parallelization of the solution with multi-GPU methods. These improvements pave the way for numerically efficient models that can incorporate the most important physical phenomena affecting microstructure.

2. METHODOLOGY

2.1 Parallelization

For accelerating our PDE solver we have chosen to use MPI (Message Passing Interface) for communication between processes and CUDA for GPU acceleration. The technologies were chosen since previous codes Pekkilä et al. (2022) built on top of MPI and CUDA have been able to achieve impressive performance for multi-GPU stencil computations, with our PDE solver belonging to this family of *iterative stencil loop* (ISL) -algorithms Li and Song (2004). MPI and CUDA also work well together with MPI having CUDA-aware implementations where the user can send data directly from and to GPU memory. This is convenient for the user and it also provides optimal performance for the user by routing the GPU data through the fastest interconnect Potluri et al. (2013) and pipelining the GPU data movement with the communication.

Furthermore technologies that do not require handwritten kernels to offload computation to GPUs are not competitive in performance, especially at large data sizes Khalilov and Timoveev (2021).

Parallelizing an ISL-algorithm with both MPI and CUDA is conceptually straightforward. The whole domain is split into local subdomains such that each process will process its own subdomain that it is responsible for updating. In order to compute the required stencil at each grid point the processes communicate grid points that are part of their subdomain to each other as required by the data dependency of the stencil. These regions of points that are communicated to other processes are called *halo regions*. Locally on each processes the incoming halo regions coming from other processes are stored to regions called *ghost zones* that surround the local subdomain of the process.

The amount of communication needed for a single stencil iteration depends on the radius of the stencil. The radius of a stencil is defined as the maximum Chebyshev distance from the central grid point to all neighbouring grid points that are required for calculating the stencil at the central grid point. N iterations of a stencil of radius R requires rectangular halo regions of width $N \times R$ for the two-dimensional case.

GPU acceleration is also straightforward since each update at a grid point is independent from updates at each other point. For achieving close to optimal performance for memory-bound applications the GPU acceleration becomes considerably more complex since it becomes important to ensure good cache reuse to alleviate the bottleneck of data movement from global memory. However we observed our application being compute bound on the hardware used, a single RTX A2000 8GB Laptop GPU, to benchmark against the original Matlab implementation, so we were able to achieve relatively good performance with quite simple kernels.

2.2 Numerical method

Here we present the implemented numerical method, which was in its initial form first presented in Pohjonen (2024b) as well some optimizations to the equations that now yield the results with less total compute. The optimized algorithms were discovered during implementation work required to accelerate the code.

Each grid point p_i and its surrounding neighbours can be grouped as triplets, which can be visualized as triangles surrounding the areas A_i shown in in Fig. 1.

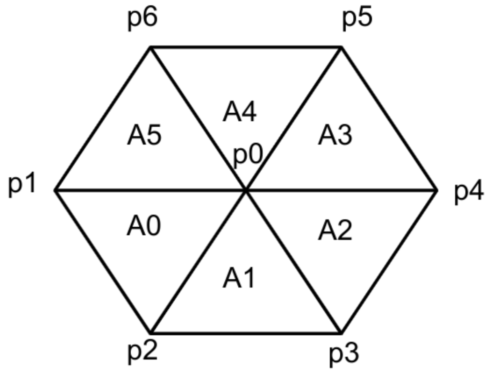

Fig. 1. Each grid point p_i and its surrounding gridpoints are grouped as triplets, which form triangular regions A_i. Each triplet defines a plane whose coefficients yield the average gradients of these regions.

Each triplet defines a plane, where the plane coefficients are calculated based on the field values and the positions of the grid points. The coefficients yield the average gradient in each of the areas A_i. The average gradient in the whole hexagonal region, composed from the surrounding triangles, is then obtained as the weighted average of the gradients by using the areas of the regions as weights.

The equations for the coefficients of the planes are:

$$a = \frac{(u_1 - u_0)y_2 + (u_0 - u_2)y_1 + (u_2 - u_1)y_0}{(x_1 - x_0)y_2 + (x_2 - x_0)y_1 + (x_2 - x_1)y_0} \tag{1}$$

and

$$b = \frac{(u_1 - u_0)x_2 + (u_0 - u_2)x_1 + (u_2 - u_1)x_0}{(x_1 - x_0)y_2 + (x_2 - x_0)y_1 + (x_2 - x_1)y_0} \tag{2}$$

The equations for the areas of the triangles are:

$$A_i = \frac{|(p_i - p_0) \times (p_{i+1} - p_0)|}{2} \tag{3}$$

And finally the equations for the partial derivatives are:

$$\partial_x u|p_0 = \frac{\sum_{i=0}^{i=5} A_i a_i}{\sum_{i=0}^{i=5} A_i} \tag{4}$$

$$\partial_y u|p_0 = \frac{\sum_{i=0}^{i=5} A_i b_i}{\sum_{i=0}^{i=5} A_i} \tag{5}$$

.

The first simplification is that since the weighting coefficients are the ratios of the areas of the triangles to the the sum of all of the areas the divisor of 2 cancels out in and for the weights we can use the equation:

$$W_i = |(p_i - p_0) \times (p_{i+1} - p_0)|, \tag{6}$$

where the divisor is now the sum of the weights.
Another simplification is that if one calculates the coefficients of the planes with the center grid point being the origin we have $x_0 = y_0 = u_0 = 0$, which allows us to simplify some terms in the formula for the coefficients:

$$a = \frac{u_1 y_2 - u_2 y_1}{x_1 y_2 - x_2 y_1} \tag{7}$$

$$b = -\frac{u_1 x_2 - u_2 x_1}{x_1 y_2 - x_2 y_1}, \tag{8}$$

where x_1, x_2, y_1, y_2 are coordinates in the coordinate frame where the center grid point is the origin.

Now importantly, if one chooses the grid points in a way that the cross product responding to the weights is positive, the denominator terms for the equations for a and b are equal to the weights. This means that one can skip division and multiplication by the cross products since they cancel out. With this insight the new formula for the partial derivatives becomes:

$$\partial_x u|p_0 = \frac{\sum_{i=0}^{i=5} a_i}{\sum_{i=0}^{i=5} W_i} \tag{9}$$

$$\partial_y u|p_0 = \frac{\sum_{i=0}^{i=5} b_i}{\sum_{i=0}^{i=5} W_i}, \tag{10}$$

where a_i and b_i are computed without the division with the cross product. Having to do less division gives a noticeable performance improvement, since division is a more costly operation than multiplication and addition.

2.3 Interpolation with plane equations

Previously a two-grid approach where a denser grid was created out of the original grid by interpolating between each grid point, was presented Pohjonen (2024a). During each timestep the interpolated denser grid is used to calculate the partial derivatives and the results are copied back to the coarse grid, from which a new dense grid is created for the next timestep. The two-grid interpolation method was found to help with instability that was caused near maxima and minima of the grid values. However the two-grid interpolation method comes at a cost namely that one has four as many grid points to compute the partial derivatives compared to the original coarse grid.

This motivated investigation to whether the added accuracy of the two-grid interpolation grid could be achieved without the need for a coarser grid. Since the interpolation points always lie between points in the original grid one can calculate the partial derivative values that would be at the interpolation points using the coarser mesh. One creates the interpolated points and triangles locally but they are only used locally to calculate the partial derivatives and are not stored anywhere. A visualization of the formed triangles can be seen in Fig. 2.

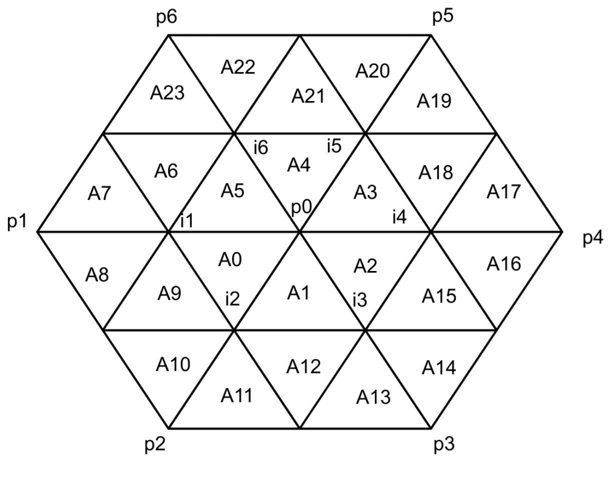

Fig. 2. The interpolated triangles drawn out. Triangles from $A_0 - A_5$ follow the same indexing scheme as in the non-interpolated scheme. The rest of the triangles are indexed in reverse clockwise order. $p_0 - p_6$ are the original grid points with $i_1 - i_6$ being the interpolated points where the first partial derivatives are calculated.

Now as an example $\partial_x u|i_1$ would be the sum of the coefficients coming from planes: A_0, A_5, A_6, A_7, A_8 and A_9. One can similarly also calculate the first order partial derivatives at the interpolated points i_2, i_3, i_4, i_5, i_6. This approach works, but it adds significantly more compute since now there are four times more planes to calculate.

Thus an important observation is that all of the added planes are copies of the innermost six planes, which are copies of the original planes between the non-interpolated points. This is because a linearly interpolated point between two points on a plane stays on the same plane as the original points.

Similarly the area of the interpolated triangles is exactly one fourth of the original triangles so one can use the original areas to calculate the weights since the common factor anyway cancels out. Taking all of this into account the equation for partial derivatives at the interpolated points simplifies, after cancelling another common factor of three out, to:

$$\partial_x u|i_{j+1} = \frac{a_j + a_{j+1}}{W_j + W_{j+1}} \tag{11}$$

$$\partial_y u|i_{j+1} = \frac{b_j + b_{j+1}}{W_j + W_{i+j}}, \tag{12}$$

where the indexing $j \in \{0, 1, 2, 3, 4, 5\}$ forms a periodic sequence such that $a_6 = a_0, b_6 = b_0$ and $W_6 = W_0$.

Thus one can calculate the first order partial derivatives at the interpolated points with a modest number of added compute. This is a worthwhile trade-off for being able to use a four times smaller grid and eight times smaller grid in two- and three- dimensions respectively. Also, since second order derivatives are only needed at the original coarse grid points one can now calculate second order partial derivatives during the same iteration in which the first order partial derivatives are calculated. This effectively halves the amount of needed memory traffic since the coordinates and field derivatives do not have to be refetched from global memory but can be directly acccessed from local memory, either being stored in registers or cache. One also saves compute since the second order partial derivatives are not unnecessarily computed at the interpolation points.

For computing the second order partial derivatives one would in general need the interpolated coordinates because the innermost planes are not anymore the same since they depend on the values at the interpolated points. However, one can show that using the original coordinates gives exactly half of the correct results so instead of calculating the interpolated points one can simply scale up the result computed with the original grid points by a factor of two.

2.4 Rectangular grid

The numerical method was originally implemented using a regular rectangular grid Pohjonen (2024b). The regions and grid points for a rectangular grid are visualized in Fig. 3. The equations are exactly the same except now we calculate and add up plane coefficients for four planes instead of six.

The motivation for using a hexagonal grid came from the fact that with a hexagonal grid each interpolation point can be uniquely identified with linear interpolation between two grid points. This is not possible for a rectangular

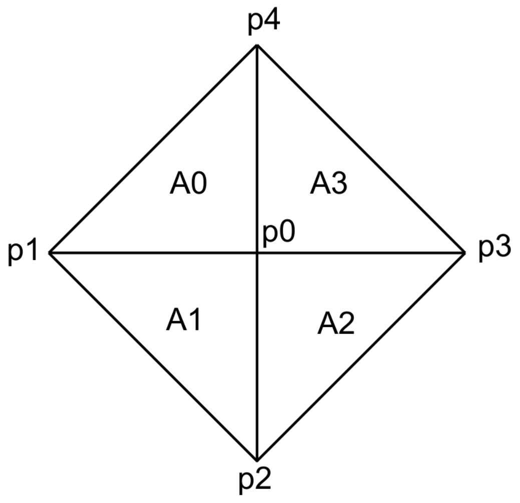

Fig. 3. Visualization of the local regions and grid points for a rectangular grid.

grid and requires for example bilinear interpolation of the surrounding four points for some interpolation points.

The rectangular grid has the advantages that it requires less computation and results in simpler memory access patterns. The rectangular grid requires less compute simply because it requires calculating plane coefficients for four planes instead of the six required for the hexagonal one. The memory access pattern of the hexagonal grid is more complex since technically it requires the use of two different stencils or the union of these stencils and the stencil used depends on the central grid point, whereas the rectangular grid requires only a single stencil that is used for all grid points. A union of two stencils contains all neighbouring grid points that are included in either stencil.

The hexagonal grid requires two stencils since as an example the x-index offset to get the upper left grid point p_6 of the local hexagon depends on the y-index of the center point. Of course our treatment of x- and y -indexes are arbitrary and one can interchange how they are used. The rectangular grid requires only a single stencil since the offsets from index of the central grid point are always the same. Visualizations of the different used stencils can be seen in Fig. 4.

For a code specifically designed for our numerical approach the added complexity of which stencil to use does not really matter but it makes it harder to implement the numerical method optimally in GPU-computing libraries like Pekkilä (2019), where it would otherwise be simple.

Importantly, the communication for both meshes is the same since the union of the two stencils for the hexagonal mesh and the single stencil for rectangular mesh both have a radius of one.

The new improved algorithm motivated us to test could it be used with the original rectangular grid to achieve the improved accuracy of the two-grid approach.

The new algorithm works exactly the same as for the hexagonal grid. Due to cancellation of common terms the

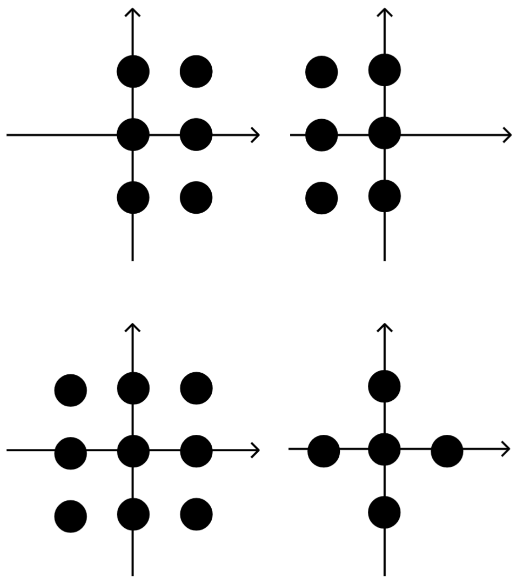

Fig. 4. Visualizations of the different stencils. The updated grid point is located at the origin. The upper stencils are the two stencils required for the hexagonal grid, the lower left stencil their union and the lower right stencil the one required for a rectangular mesh.

equations (11) and (12) can be used to compute the partial derivatives at the interpolated points $j \in 0, 1, 2, 3$, where the periodic sequence naturally is now that $a_4 = a_3, b_4 = b_3$ and $W_4 = W_3$. The interpolated points and regions can be see in Fig. 5

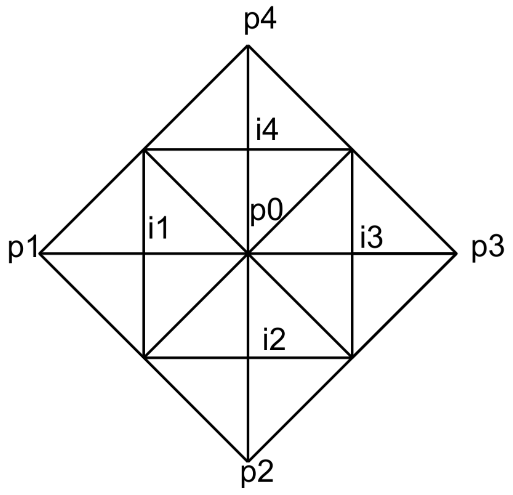

Fig. 5. Interpolated points i_{1-4} and interpolated regions drawn out.

Even though the equations are the same for using the rectangular grid the method is subtly different in the sense that now the computed gradient values at the interpolated points are different between neighbouring grid points, whereas they are the same with the hexagonal grid. We did not find this discrepancy causing problems when using the new algorithm with the rectangular grid.

2.5 Reusing partial results

Since on the tested hardware the GPU kernels were found to be compute bound we investigated approaches that can reuse partially computed results. The observation that the planes A_2, A_3 at grid point x, y are the same as the planes A_0, A_5 at $x+1, y$ motivated the idea of a single thread computing multiple contiguous results in the x-direction. Thus we implemented a kernel where each nx, ny thread block computes $nx \times BlockSize, ny$ results with thread i, j calculating the results at points $i, j, i+1, j..., i+BlockSize, j$, where $BlockSize$ is a tunable parameter. In order to avoid non-contiguous memory accesses between threads in a warp, the whole thread block first loads all needed $nx \times BlockSize+1, ny+1$ input values into shared memory in a contiguous manner from which threads can access the values in a non-contiguous pattern without a performance penalty. Similar to the planes the interpolation point i_4 at grid point x, y corresponds to i_1 at the grid point $x+1, y$ so the computed gradient values at the interpolation point can be also reused. A visualization of the algorithm for a 2×2 thread block with $BlockSize = 4$ can be seen in Fig. 6.

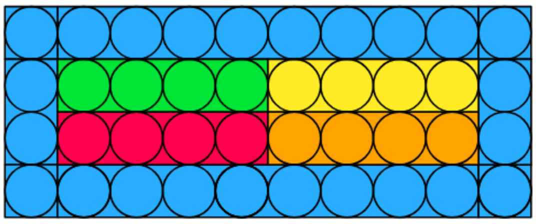

Fig. 6. Visualization of the algorithm for a 2×2 thread block with $BlockSize = 4$. Blue regions are those that are only loaded and read from shared memory. Green, yellow, red and orange points are computed by threads $(0,0)$,$(0,1)$,$(1,0)$ and $(1,1)$ respectively.

One could also reuse partial results also in the y-direction, but we leave answering whether reusing partial results in both dimensions increases performance to future work.

2.6 Communication

For optimizing the communication of the halo regions between the processes we use the important technique of overlapping computation and communication. MPI supports an asynchronous API where one can immediately continue execution after the sending of a message has started and wait later for the arrival of the message. This allows us to compute the update at those grid points that do not depend on the ghost zones at the same time as the ghost zones are being received. Finally when the ghost zones are received we can compute the update at the outer regions that depend on the ghost zones. This overlaps most of the computation with the communication, which makes the runtime of a grid update be $max(W, C)$ instead of the naive $W + C$, where W is the time needed for the computation and C is the time needed for the communication. In Fig. 7 one can see a visualization of the different subdomain regions of a stencil of radius one.

Our approach for overlapping communication and computation is equivalent to the one in Pekkilä et al. (2022),

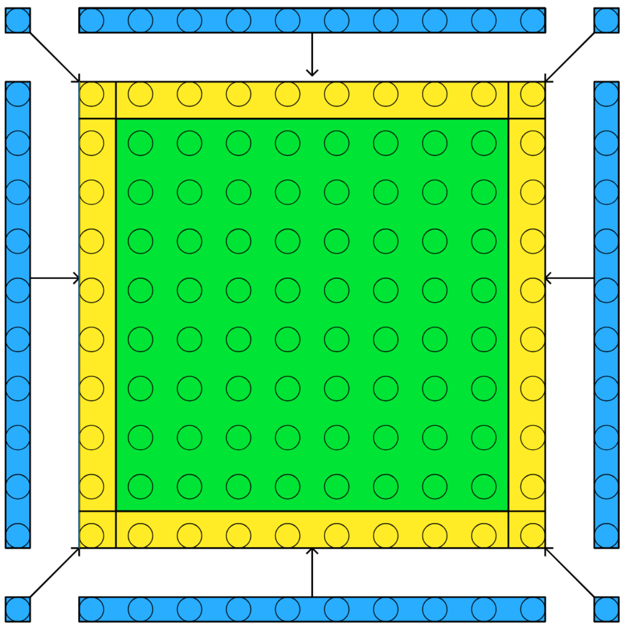

Fig. 7. Regions of a single stencil update for a stencil of radius one. Grid points in green do not depend on the blue ghost zones coming from other processes and their computation can be overlapped with the communication of the blue ghost zones. The yellow outer regions depend on the blue ghost zones and their update has to wait for the arrival of the ghost zones.

except we have a two dimensional simulation instead of a three dimensional one. For performing the original algorithm that requires two iterations of the grid to compute second-order derivatives is also suboptimal from the viewpoint of communication, because it needs halo regions of width two since the grid is iterated twice for a single update. Being able to calculate the update in a single iteration means we can use halo regions of width one, which halves the amount of communication needed.

3. RESULTS

3.1 Comparison to analytical solution

To make sure the accelerated version and the new algorithm have the same accuracy as the original code we tested the numerical solution similarly to Pohjonen (2024b) and Pohjonen (2024a).

More specifically we solved the diffusion equation (13) from an initial point concentration and compared it to the analytical solution (14) MIT (2024).

$$\partial_t u = D(\partial_{xx} u + \partial_{yy} u) \tag{13}$$

$$u(x,y,t) = \frac{M}{4\pi t D} exp(-\frac{(x-x_c)^2 + (y-y_c)^2}{4Dt}), \tag{14}$$

with x_c, y_c being the origin of the initial point concentration. Values of $D = 1$ and $M = 0.1$ were used.

To make sure the code works on a deformed grid we randomly perturbed each grid points coordinates by $0.16(r -$

$0.5)\cdot a, 0.16(r-0.5)\cdot h$, where r is a random number between 0 and 1 and a is the x-distance between neighbouring grid points and h is the y-distance between neighbouring grid points. Comparison of the analytical solution to the result of the simulation can be seen in Fig. 8.

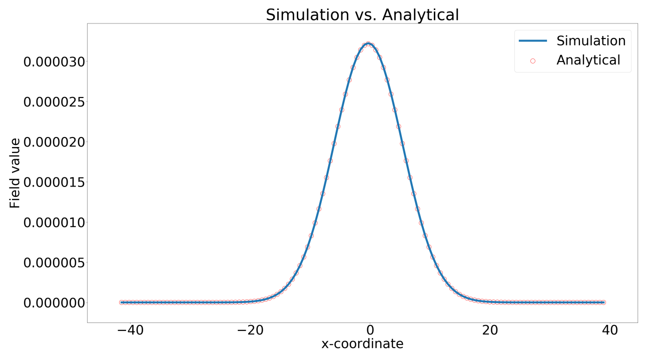

Fig. 8. Simulation result compared to the analytical result. The plotted values are a one-dimensional slice along the x-dimension in the middle of the grid.

We also tested the new algorithm using the rectangular grid and found it to give as precise answers as the hexagonal grid for this simple test case.

3.2 Benchmarks

We present timings of the different GPU kernels in Fig. 9. The benchmarks were performed on a single RTX A2000

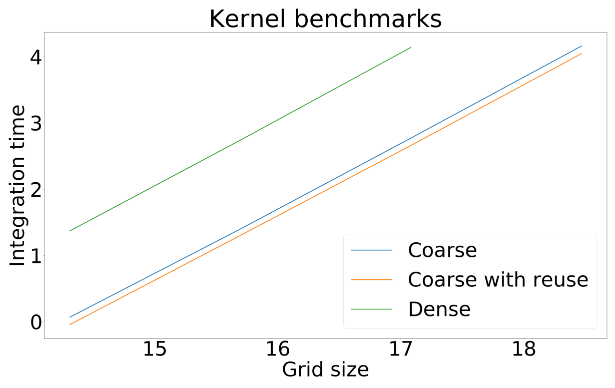

Fig. 9. Timings of different kernels. Integration time is measured as the log seconds taken to integrate 200 timesteps and grid size is measured as the log amount of grid points.

8GB Laptop GPU for easy comparison of our accelerated implementation with the original one.

We observe that the difference between computing with the coarse and dense grid is close to the factor of four one would assume from the grid sizes. A benefit of using the smaller grid and not having to store the calculated first order derivatives means that one needs less memory for the simulation and one is able to simulate larger grids. As an example in the benchmark results one can simulate $64 \cdot 160 \times 64 \cdot 160$ grid with the coarse grid but with the dense grid one runs out of memory. Also there is quite a constant performance increase of ten percent between the coarse grid kernel and the coarse grid kernel reusing partial results. *BlockSize* of 11 was found to give optimal performance on the tested hardware.
We do not present detailed timings for the original Matlab implementation but see it adequate to mention that the best benchmark results for the Matlab implementation were when it was more than 2000 times slower. Thus it is clear the the performance increase made possible with the accelerated version makes significantly larger simulations possible. We also do not compare the performance of the GPU implementation against the CPU implementation because we have spent considerably more effort on optimizing the GPU implementation, which would make the comparison unfair and misleading.

3.3 Scaling

Not only is the single GPU performance important, but it is important that we get good scaling when increasing the amount of GPUs we use. The theoretical optimal scaling is that when we increase the number of devices by a factor of N we get a speedup factor of N also.

The reported scaling benchmarks were performed on the CSC supercomputer Puhti, which has four Xeon Gold 6230 Nvidia V100 -GPUs per node with peak bandwidth of 200 Gpbs between nodes.

For ISL-algorithms the scaling starts to deviate from the theoretical optimum when the network bandwidth becomes the performance limiter Pekkilä et al. (2022). Since the amount of needed compute scales as $O(N^2)$ and the amount of needed communication scales as $O(N)$ for a subdomain of size $N \times N$, we can get good scaling by keeping the subdomain sizes large enough.

Strong scaling, meaning how well does the code scale when we add more GPUs to a simulation of fixed size, results of the simulation can be seen in Fig. 10. From the figure one

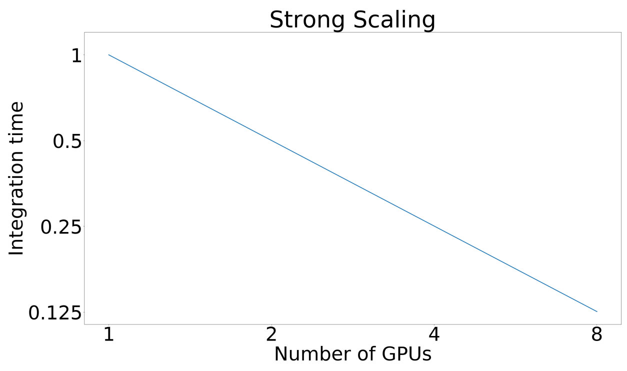

Fig. 10. Strong scaling of $60 \cdot 160 \times 60 \cdot 160$ grid. Integration time is measured as the time taken to integrate 200 timesteps relative to the time taken on a single GPU.

can see that we have good strong scaling up to 8 GPUs, with the required time going down linearly as expected.

Weak scaling, meaning how well does the code scale when we add more GPUs with a fixed subdomain size, results of the simulation can be seen in Fig. 11. From the plot one can see that we get good weak scaling with the time required staying effectively constant.

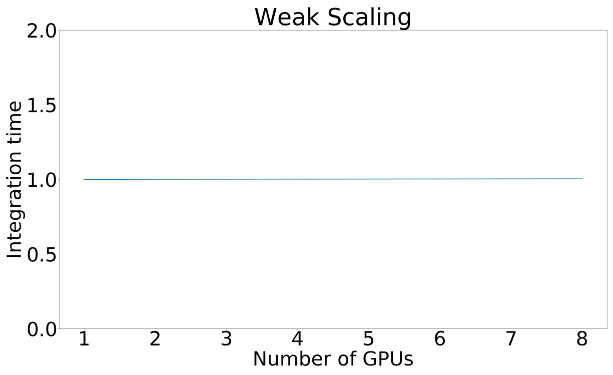

Fig. 11. Weak scaling for subdomains of size $60 \cdot 160 \times 60 \cdot 160$. Integration time is measured as the time taken to integrate 200 timesteps relative to the time taken on a single GPU.

4. CONCLUSIONS

We have presented optimizations of the original code and the used numerical method from the viewpoints of GPU acceleration, algorithmic improvements and how the method is mathematically formulated. Together these optimizations achieve an impressive speedup factor of over 2000. This enables more accurate simulations of material microstructure, for example simulations of three dimensional cases, that were previously too prohibitive due to the needed additional computation required. Additionally our implementation scales well to multiple GPUs which enables larger simulations, where more computational resources are needed.

We also investigated the use of a simpler rectangular mesh which would be easier to use and would require less compute. The accuracy of the rectangular mesh has to be studied in harder test cases.

In future studies we plan to conduct larger simulations of material microstructure. After the original implementation work we have reimplemented the algorithm and the solver in the aforementioned GPU-computing library Astaroth.

Because in the new implementation the numerical algorithm is more separated from the rest of code it is more suitable to be the reference implementation. The reference implementation, which can be accessed from this link, also includes preliminary 3d implementation of the method that will be expanded on future publications.

ACKNOWLEDGEMENTS

We acknowledge Matthias Rheinhardt for useful discussions that clarified the tradeoffs of using a Lagrangian compared to an Eulerian mesh for material simulations involving displacements. Aarne Pohjonen acknowledges financial support from the AID4GREENEST project (AI powered characterization and modelling for green steel technology) funded by Horizon Europe Research and Innovation Program, Grant Agreement No. 101091912

REFERENCES

Basaran, S. (2008). *Lagrangian and Eulerian descriptions in solid mechanics and their numerical solutions in hpk framework*. Ph.D. thesis, University of Kansas.

Hallberg, H. (2013). A modified level set approach to 2D modeling of dynamic recrystallization. *Modelling and Simulation in Materials Science and Engineering*, 21(8), 085012. doi:10.1088/0965-0393/21/8/085012. doi:10.1088/0965-0393/21/8/085012.

Khalilov, M. and Timoveev, A. (2021). Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU. In *Journal of Physics: Conference Series*, volume 1740, 012056. IOP Publishing.

Li, Z. and Song, Y. (2004). Automatic tiling of iterative stencil loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6), 975–1028.

MIT (2024). Lecture Notes on diffusion, accessed 10.6.2024, Massachusets Institute of Technology. URL http://web.mit.edu/1.061/www/dream/THREE/THREETHEORY.PDF.

Pekkilä, J. (2019). Astaroth: A library for stencil computations on graphics processing units. URL https://aaltodoc.aalto.fi/server/api/core/bitstreams/c73ad7b3-47a2-4c23-b802-7721366fb961/content.

Pekkilä, J., Väisälä, M.S., Käpylä, M.J., Rheinhardt, M., and Lappi, O. (2022). Scalable communication for high-order stencil computations using CUDA-aware MPI. *Parallel Computing*, 111, 102904.

Pohjonen, A. (2023). Full field model describing phase front propagation, transformation strains, chemical partitioning, and diffusion in solid–solid phase transformations. *Advanced Theory and Simulations*, 6(3), 2200771. doi:10.1002/adts.202200771. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/adts.202200771.

Pohjonen, A. (2024a). Application of two-grid interpolation to enhance average gradient method for solving partial differential equations. *The Journal of Physics: Conference series*, 2701(2), 97–111.

Pohjonen, A. (2024b). Solving partial differential equations in deformed grids by estimating local average gradients with planes. *The Journal of Physics: Conference series*, 2701(2), 97–111.

Potluri, S., Hamidouche, K., Venkatesh, A., Bureddy, D., and Panda, D.K. (2013). Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs. In *2013 42nd International Conference on Parallel Processing*, 80–89. IEEE.

Seppälä, O., Pohjonen, A., Mendon, V., Podor, R., Singh, H., and Larkiola, J. (2023). In-situ SEM characterization and numerical mod-elling of bainite formation and impingement of a medium- carbon, low-alloy steel. *Materials & Design*, 230, 111956. doi:10.1016/j.matdes.2023.111956. URL https://www.sciencedirect.com/science/article/pii/S0264127523003714.

Steinbach, I. and Salama, H. (2023). *Lectures on phase field*. Springer Nature.