Solving the Poisson equation with GPU acceleration

Final report for the course:

GPU-accelerated Computational Methods using Python and CUDA

Greeshma Ajayakumar, Jakub Fojt, Jian Tan, Leonard Nielsen

January 2023

Contents

1 Introduction

The use of graphical processing units (GPUs) to carry out high-performance computing has been rapidly growing in popularity over the past decade. GPU hardware architecture is especially suitable for carrying out highly parallel computations.

CUDA, a proprietary application programming interface (API) from NVidia makes it relatively easy to implement kernels for scientific computing in software, compared to API:s such as OpenGL, which were originally developed for rendering applications.

The Poisson equation occurs frequently in many areas of physics, *e.g.* electromagnetic and fluid dynamics, relating the electrostatic potential to the charge distribution in the former, and the pressure field to the velocity distribution in the latter.[1] The solution of the Poisson equation (*i.e.* determining the pressure field corresponding the a given velocity distribution) is often the costly part of fluid flow simulation. Therefore, accelerating its computation is valuable.[2] In this report, we describe our implementation of a GPU-accelerated implementation of a Poisson equation solver, and discuss the choices we made to improve the performance of the solver.

1.1 Formal definition of the Poisson solver

The Poisson equation in its compact form reads

$$\Delta\phi(\vec{r}) = f(\vec{r}),$$

where Δ is the Laplace operator. Explicitly writing out the Laplace operator in Cartesian coordinates the equation becomes

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right)\phi(\vec{r}) = f(\vec{r}). \tag{1}$$

Here, $f(\vec{r})$ is a source term that is defined by the problem, *i.e.* the charge distribution in an electrostatics simulation, or a quantity related to the velocity distribution. The source term is supplied by the user to the Poisson solver, and when the calculation is finished the potential $\phi(\vec{r})$ is obtained.

To solve the Poisson equation numerically, it is written in an approximate, discrete form

$$A\boldsymbol{x} = \boldsymbol{f}, \tag{2}$$

where $\boldsymbol{x}$ and $\boldsymbol{f}$ are vectors containing all the solution $\phi(\vec{r}_i)$ and source term $f(\vec{r}_i)$ for each point $\vec{r}_i$ on a uniform grid. The discretization of Eq. (2) is detailed in Appendix A. The matrix A is the discretization of the Laplace operator and is given in Appendix A. In particular, the matrix A is very sparse, as the derivative at one point, is approximately given by only values in a few neighboring grid points. Therefore, a numerical sparse matrix representation of the matrix A is used.

1.2 Numerical solution of the problem

In this section, we describe the algorithm used to solve the Poisson equation numerically. In principle, the iterative Gauss-Seidel algorithm is sufficient to for this purpose, but in practice it requires very many iterations to converge sufficiently close to the solution. To accelerate the convergence, a V-cycle multigrid scheme is used, where the Gauss-Seidel algorithm is a smoothing step. In this section, we point out which numerical operations are necessary to implement the algorithm. These are: sparse matrix-dense vector multiplication, lower triangular sparse matrix equation solution, and basic vector arithmetic.

1.2.1 The Gauss-Seidel algorithm

The aim of the Gauss-Seidel algorithm is to (approximately) solve Eq. (2) for $\boldsymbol{x}$. This is done by splitting the matrix A into one strictly upper triangular part U and one lower triangular part L

$$A = U + L.$$

A series of approximate solutions for $\boldsymbol{x}$ is given by the relation

$$L\boldsymbol{x}^{(k+1)} = \boldsymbol{f} - U\boldsymbol{x}^{(k)}.$$

Thus the Gauss-Seidel algorithm is defined by

1. Start with an initial guess $\boldsymbol{x}^{(0)}$
2. Calculate $\boldsymbol{y}^{(k)} = \boldsymbol{f} - U\boldsymbol{x}^{(k)}$.
   This step consists of a **sparse matrix-dense vector multiplication** and a **vector subtraction**
3. Solve for $\boldsymbol{x}^{(k+1)}$ in $L\boldsymbol{x}^{(k+1)} = \boldsymbol{y}^{(k)}$.
   This step is a **lower triangular sparse matrix equation solution**, which is described below
4. Repeat from 2.

1.2.2 Solution of the lower-triangular matrix equation

The solution of the lower-triangular matrix equation $L\boldsymbol{x} = \boldsymbol{y}$ can be done efficiently thanks to the lower triangular structure of L. Let L be given by

$$\begin{bmatrix} l_{11} & 0 & \dots & & \\ l_{21} & l_{22} & 0 & \dots & \\ l_{31} & l_{32} & l_{33} & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \end{bmatrix}$$

Then the inverse calculation gives an equation system for the elements of $\boldsymbol{x}$

$$\begin{cases} l_{11}x_1 & = y_1 \\ l_{21}x_1 + l_{22}x_2 & = y_2 \\ l_{31}x_1 + l_{32}x_2 + l_{33}x_3 & = y_3 \\ \dots & \end{cases}$$

Thus, each element of $\boldsymbol{x}$ is defined in terms of all previous x_i and y_i

$$x_i = \frac{y_i - \sum_{j=1}^{i-1} x_j l_{ij}}{l_{ii}},$$

assuming that L has all non-zero diagonal elements l_{ii}.

By dividing each row i of L and each element y_i by the diagonal element l_{ii} the same equation holds, but now the diagonals are 1

$$x_i = y_i - \sum_{j=1}^{i-1} x_j l_{ij}$$

assuming that L has all non-zero diagonal elements l_{ii}.

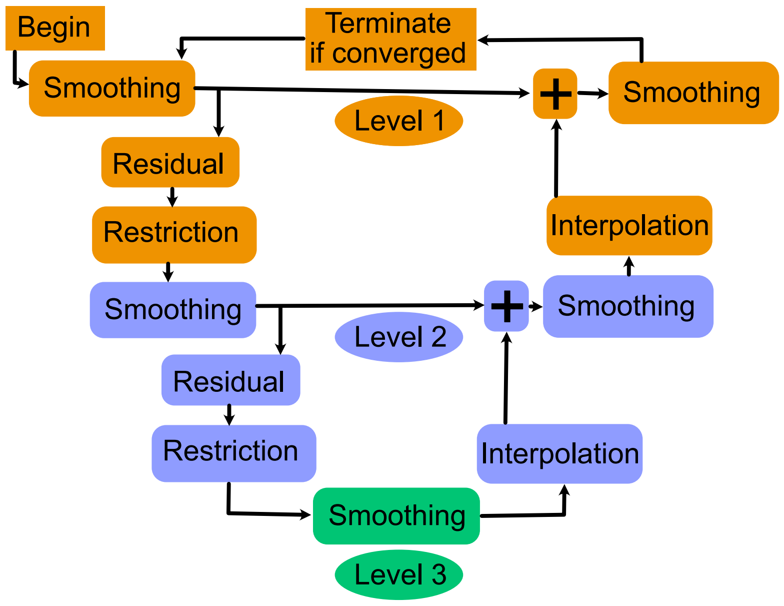


Figure 1: Schematic illustration of a V-cycle multigrid solver with three levels. Rectangles with rounded corners show the computation steps of the V-cycle. Rectangles with squared corners show the beginning and termination of the process. Ovals show which level each color belongs to.

1.2.3 Multigrid method

We use a multigrid method to solve the Poisson equation.[3] Multigrid methods are suitable for parabolic systems of equations, which tend to suffer from poor convergence of their low-frequency components. They consist of three key components - a *smoother*, which is an approximate solver using e.g. a few Gauss-Seidel iterations; a *restriction*, which is a downsampling; and an *interpolation*, which is an upsampling step. These three component are combined with an iterative refinement scheme to obtain a solution for the whole system. We employ the V-cycle multigrid approach, which is illustrated in Fig. 1. One begins by applying the smoother to obtain an approximate solution to the system. Then, one computes the residual of this solution and restricts, i.e., downsamples it. The smoother is then again applied to obtain a solution for the restricted residual. A residual for the restricted system is computed and further restricted, and so on, until the highest level of restriction is reached. At this point, the lowest-resolution solution is interpolated, or upsampled, to the preceding level, and added to that level's solution. The smoother is again applied, and the solution is interpolated, etc., until one reaches the original resolution of the system. A single iteration of the V-cycle has then been completed.

In this algorithm, the restriction and interpolation are implemented as **sparse matrix-dense vector multiplication** using restriction and interpolation matrices R and I, and the residual calculation consists of a **sparse matrix-dense vector multiplication** and a **vector subtraction**.

2 Implementation

The entire procedure of the Poisson solver, i.e., the V-cycle scheme for solving Eq. (2), can be written as basic procedural logic and control flow statements (`if`, `for`, etc.), matrix-vector arithmetic operations and a lower triangular matrix solver (Sect. 1.2.2). In particular, the following steps are sparse matrix-dense vector multiplications:

- Restriction of the solution guess or residual, $R\boldsymbol{x}$
- Interpolation of the solution guess or residual, $I\boldsymbol{x}$
- The forward step in each Gauss-Seidel iteration, $U\boldsymbol{x}$

In addition, pre-computation of two sparse matrix-matrix multiplications for each level of the V-cycle are required to compute the approximate restricted Poisson operator, namely RAI.

In this section, we describe the implementation of a CPU version of the Poisson solver, employing the Numpy and SciPy packages for the matrix-vector operations and triangular matrix solver, and of three different GPU versions, using CuPy.

2.1 CPU implementation

The Numpy and SciPy packages are standard in scientific Python programming. Numpy provides an interface to create arrays (vectors, matrices, and higher-dimensional arrays) and allows the programmer to efficiently perform linear algebra via BLAS and LAPACK libraries.[4,5] Sparse matrices and sparse matrix operations are provided by SciPy. With these packages, the implementation of the Poisson solver in CPU code is straightforward.

Dense vectors, such as the source vector $\boldsymbol{f}$ and initial guess for the solution $\boldsymbol{\phi}$ are created as Numpy arrays `np.ndarray`. The system matrix A and restriction and interpolation matrices R and I are created as sparse matrices in the compressed sparse row format `scipy.sparse.csr_matrix`.

Sparse matrix-dense vector multiplications are performed with `np.matmul`, which implements BLAS routines, and the triangular solver with the single-threaded `scipy.sparse.linalg.spsolve_triangular` function.

2.2 CuPy

The primary scope of CuPy is the provision of a CUDA interface that operates as similarly to Numpy and SciPy as possible. This is achieved by providing Python wrappers to CUDA libraries such as cuBLAS and cuSparse, performing operations comparable to those of libraries widely used by CPU-based scientific computing applications.[6] The high degree of similarity in syntax between CuPy and Numpy/SciPy allows users to translate scientific computing code to make use of GPU resources as easily as possible, without making it necessary to learn the details of CUDA or GPU usage. The high degree of similarity between the interface of CuPy and Numpy/Scipy makes it a suitable starting point for the comparison between GPU and CPU performance.

For the CuPy implementation directly adapted from CPU-based code we create dense vectors of type `cupy.ndarray` and sparse matrices of type `cupyx.scipy.sparse.csr_matrix`. The `cupy` and `cupyx` interfaces allocate memory directly on the GPU, allowing the arrays to be used for GPU operations. For such arrays, the `cp.matmul` function is used for matrix multiplication, and the triangular solver is called via `scipy.sparse.linalg.spsolve_triangular`.

2.3 JIT kernels

In addition to providing precompiled libraries, CuPy allows for the usage of just-in-time (JIT) kernels. These kernels consist of a Python function equipped with a decorator which tells the interpreter to parse and compile it as CUDA code at the time of declaration. In order to examine whether JIT kernels could potentially be used to optimize the code, they were used to perform in-place matrix subtraction in one implementation. In addition, in order to gain a better understanding of their performance, additional JIT kernels were compared to standard CuPy functions in isolation.

2.4 cuSparse

CuPy's limited scope sometimes leads to restrictions on potential optimizations, as intermediate calculations which could potentially be reused are discarded. At the CUDA library level, the lower triangular matrix solvers called by `cupyx.scipy.sparse.spsolve_triangular` have an analysis step and a solution step. In the case of using Gauss-Seidel iterations to solve the Poisson equation, the analysis step is responsible for most of the computation time, but its results can be re-used, as long as the matrix remains the same. Caching the analysis results is not natively supported by CuPy's frontend. However, it is possible to directly access the Python calls to the analysis and solution steps through CuPy's submodules. Similarly, it is possible to access calls to cuBLAS functions for matrix arithmetic.

In one implementation, which we refer to as the "cuSparse version", the lower-level CUDA libraries accessible through CuPy submodules were utilized directly in order to optimize the computation.

3 Profiling results

In this section, we measure the performance of our one CPU and three different GPU implementations. We compare the performance for different settings, which are grid size and max level in the V-cycle. All calculations were done on the same hardware; an Intel i5-9600K @ 3.70GHz CPU with 6 cores (without hyper-threading) and 2x8 GB of 2400 MHz RAM, and a NVIDIA GeForce RTX 3080 Ti GPU with 12GiB VRAM and CUDA version 11.8. Note that the most computationally expensive part of the CPU implementation, the sparse triangular matrix solver, runs on one core only.

For the biggest grid size of the considered systems (128x128x128), the best performance was achieved with a max level of 5 in the V-cycle. Fig. 2 shows the total runtime of the Poisson solver, including time for setup, for our different implementations. The CPU only implementation is the slowest implementation for all considered grid sizes (runtime of 819 s at 128x128x128 grid size), followed by the JIT kernel implementation (171 s), and the plain CuPy implementation (10 s). The cuSparse implementation is the fastest (2.6 s). Thus, for the 128x128x128 grid size with respect to the CPU-implementation, we obtain a 4.8 times speed up for the JIT kernel implementation, 82 times speed up for the plain CuPy implementation and 314 times speed up for the cuSparse implementation.

Increasing the max level of the multigrid solver improves the performance of the algorithm, up to a certain threshold which is dependent on the grid size (Fig. 3). For the 128x128x128 grid, the total runtime of the calculation is between 3 and 6 times shorter for a max level of 5 than a max level of 3. A max level of 6 is, however, slower, as the grid in the 6:th level is too small to efficiently distribute the work on the GPU cores (each level works on a grid of half the number of points in each dimension as the previous level). For smaller grids, a smaller max level is optimal. For example, for the 48x48x48 grid, max level 3, 4 and 5 are all equally fast. Similar trends with different grid sizes and maximum V-cycle levels can be observed across different solvers as well.

In the remainder of the section, we present the timings for the different parts of the program executions.

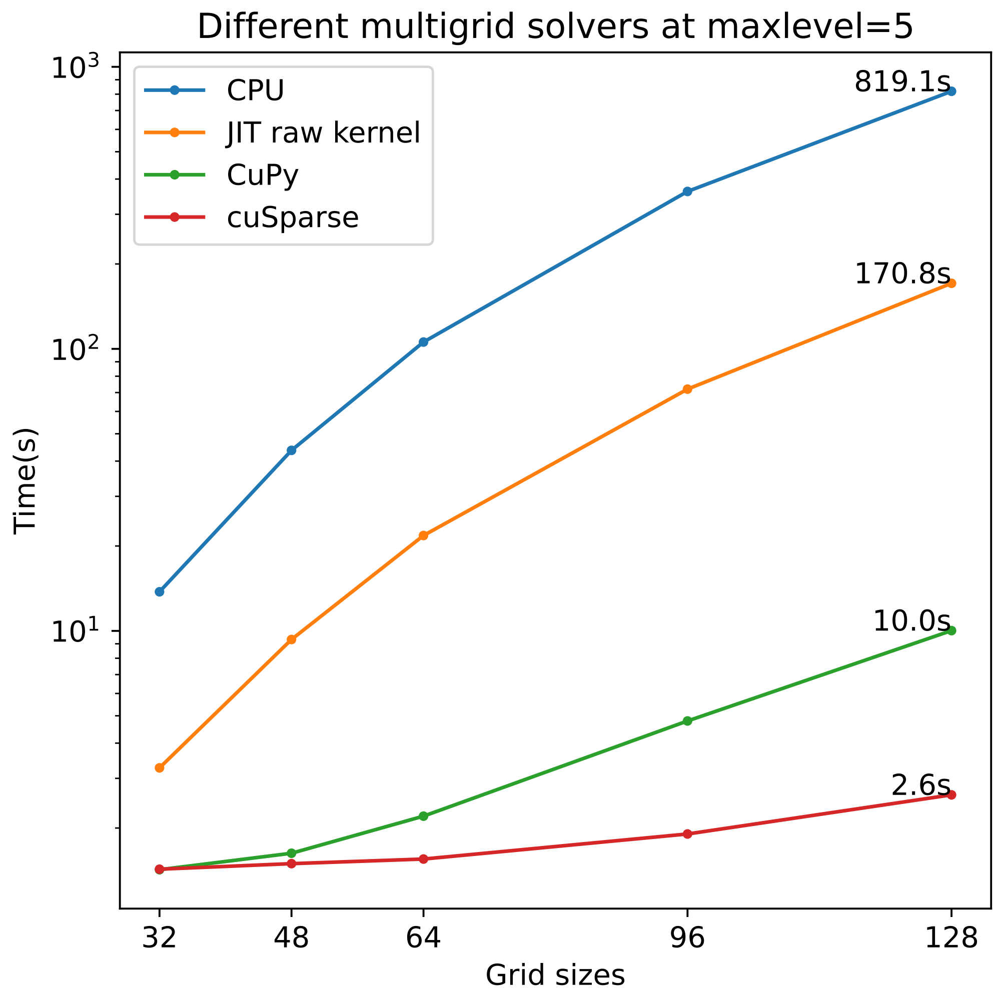


Figure 2: Run time from different cases using different multigrid methods. maxlevel=5

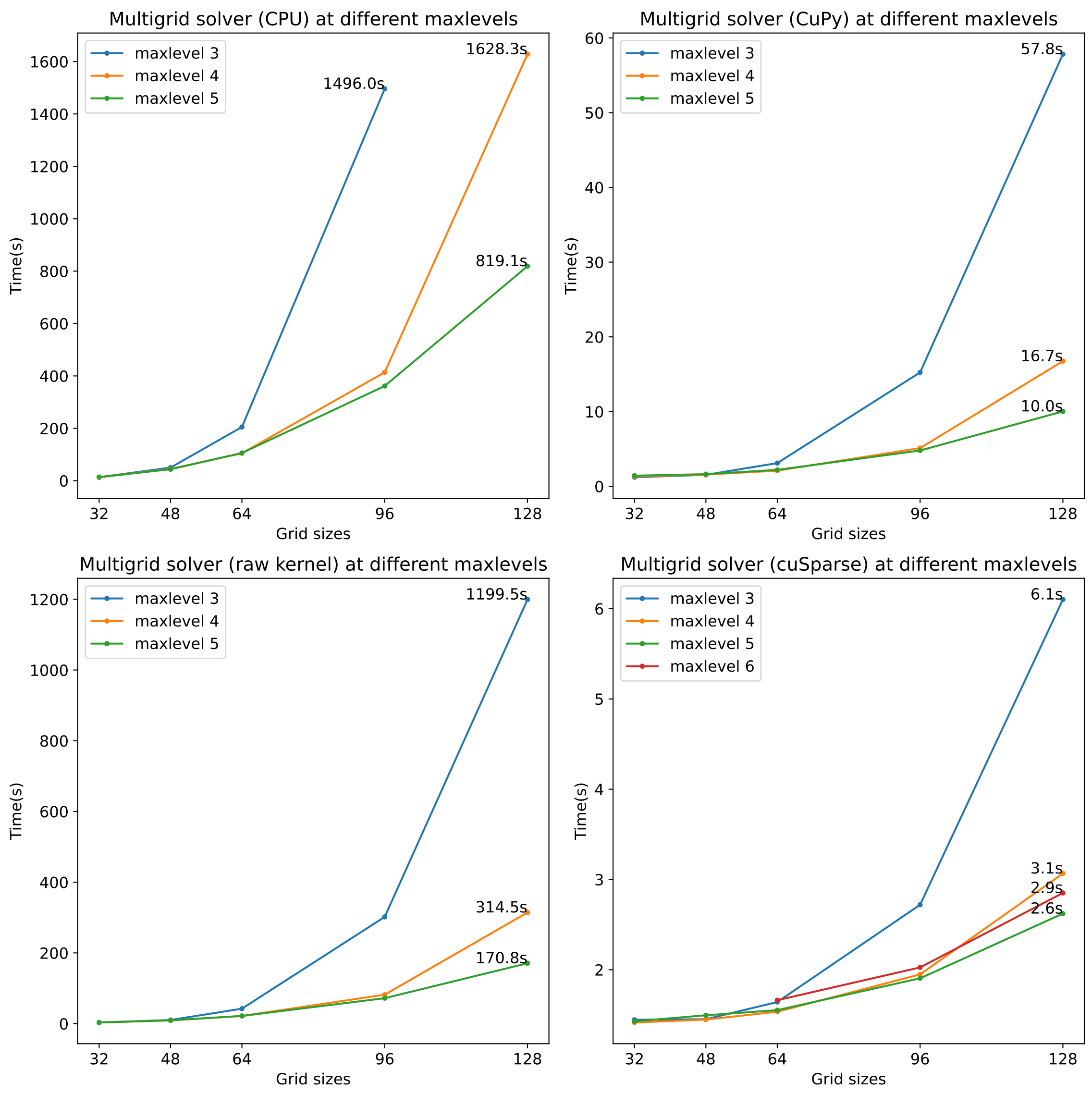


Figure 3: Run time results of different cases using differently implemented multigrid methods.
Upper left: solved by CPU. Upper right: using plain CuPy libraries.
Lower left: using JIT raw kernel. Lower right: using lower-level (cuSparse) CUDA libraries.

3.1 Dissemination of timings for a representative configuration

As explained in Sect. 1.2.3, the multigrid method contains multiple recursive V-cycles with smooth, interpolate, restrict, and compute residual steps. Each type of step takes different time lapses. Profilings of different configurations, which consist of different grid sizes, max levels and different implementations of multigrid method, are made. An example case that has a grid size of 64x64x64, with max level 5, using plain CuPy implementation (explained in Sect. 2.2), is shown in Fig. 4.

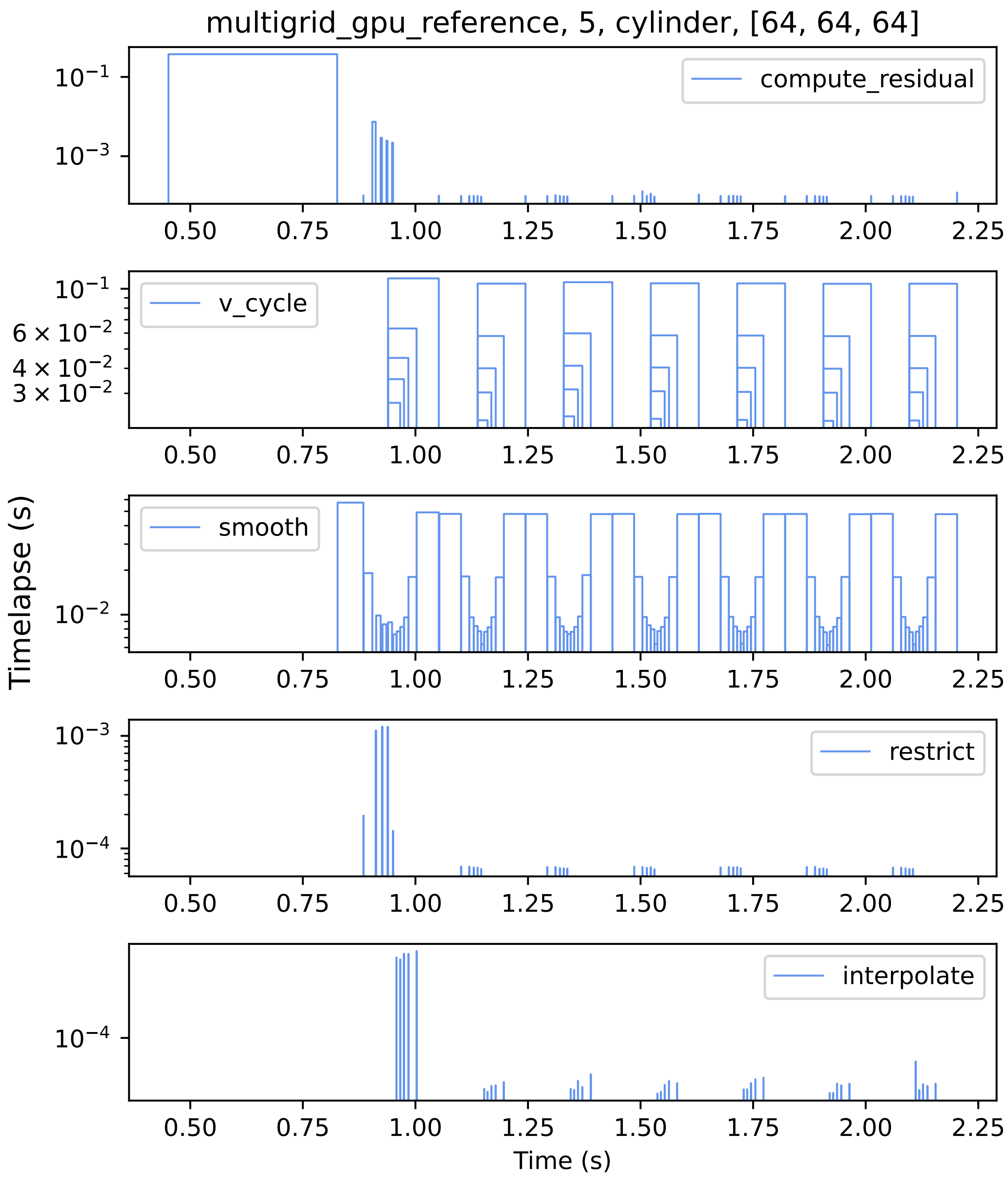
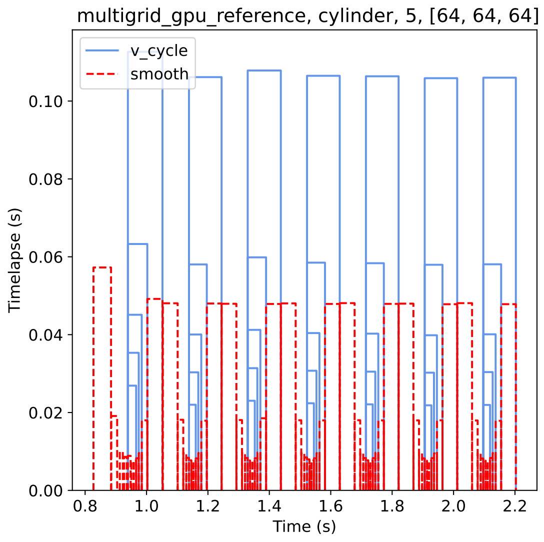
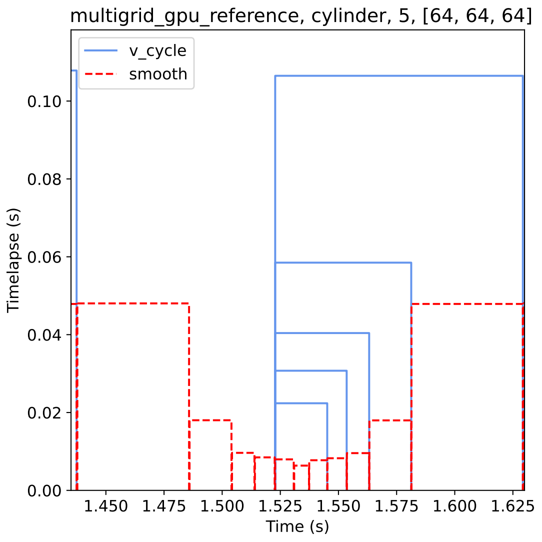


Figure 4: Execution profile for the CuPy implementation of the multigrid method, with grid size 64x64x64 and maxlevel 5. The computation consists of several nested V-cycle iterations. Restriction and interpolation takes a relatively short time, while the most time is spend smoothing. The smoothing in the other v-cycle levels (low levels in the notation of Fig. 1) takes longer time than in the inner levels, as it acts on a larger grid. The disproportionally large initial time spent in the residual computation is attributed to initial GPU synchronization.

This representative case is shown as a clear picture of the solving process and it shows that *smooth* steps are more time consuming as bottle necks compared to other steps. Fig. 5a presents only V-cycles with smooth steps and Fig. 5b is shown as a zoom-in of Fig. 5a during the 4:th V-cycle. It is clearly demonstrated that 5 V-cycles are nested when a maxlevel of 5 is implemented. And different smooth steps are undertaken within each recursive V-cycle.



(a) All V-cycles with smooth steps.



(b) Zoom-in of the 4:th V-cycle with smooth steps.

Figure 5: V-cycle and smooth steps for one run case using multigrid method with plain CuPy implementation. Grid size: 64x64x64, maxlevel=5

Efforts are therefore focused on the *smooth* step, where triangular solver for sparse matrix is implemented, to reduce its time.

3.2 Detailed comparison between implementations

3.2.1 CPU

As explained in Sect. 2.1, by solving sparse linear system directly using `spsolve` from `scipy` in the multigrid method, Fig. 3 (upper left) shows the time costs of different cases with different configurations. The case with 128x128x128 grid size and maxlevel 3 is not presented here as it is clear that it will be the most time consuming case.

3.2.2 GPU: CuPy

As explained in Sect. 2.2, by implementing pre-compiled CuPy libraries in the multigrid method, Fig. 3 (upper right) shows the time costs of different cases with different configurations. Time is greatly reduced at higher maxlevels with larger grid sizes, as a higher maximum level of V-cycle takes less iterations to converge.

3.2.3 GPU: JIT raw kernels

As explained in Sect. 2.3, by implementing matrix subtraction and multiplication with JIT raw kernels, Fig. 3 (lower left) shows the time costs of different cases with different configurations. However, time is greatly increased compared with plain CuPy library, shown in Fig. 3 (higher right).

In figure 6, we see a comparison between JIT rawkernels and CuPy functions for three operations - vector subtraction, sparse matrix-dense vector multiplication, and a sparse triangular solver. The GPU time was measured using `cupyx.profiler.benchmark`. The very first configurations of the subtraction and matrix-vector multiplication appear to be slower; this is likely due to CUDA initialization and/or compilation of the JIT kernel. This difference is not visible in the case of the sparse triangular solver, but this is likely because the solver takes an order of magnitude longer or more for the equivalent vector size. In the case of the vector subtraction, the built-in CuPy subtraction is slightly faster in almost all cases. For the cases of matrix-vector multiplication and the sparse triangular solver, the CuPy functions are faster once the size of the vectors exceed a few hundred elements, which is small in this context, since even a three-dimensional cubic grid with size $16 \times 16 \times 16$ will yield a vector with 4096 elements. This performance suggests that our JIT functions are either not as well coded as the built-in CuPy libraries, that they are compiled differently, or that the CuPy rawkernel parser does not translate the code into CUDA well. In either case, this poor performance of the individual functions for larger element numbers helps to explain why the JIT rawkernel implementation performs poorly compared to the other GPU-based implementations we wrote.

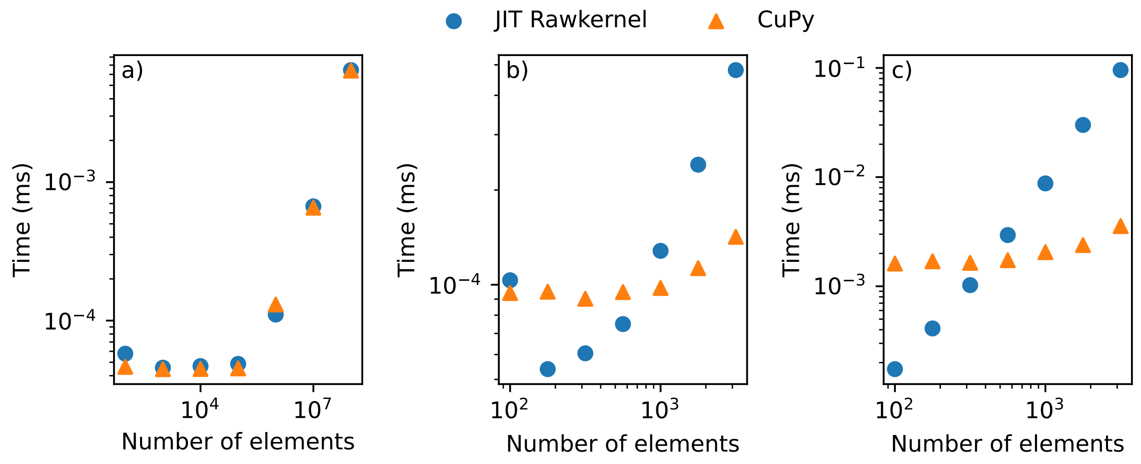
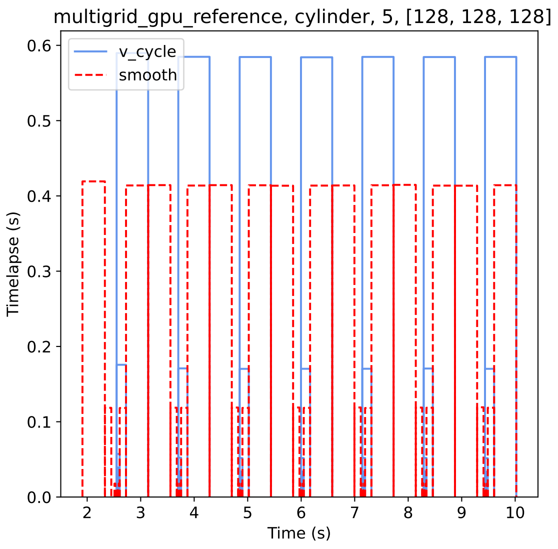


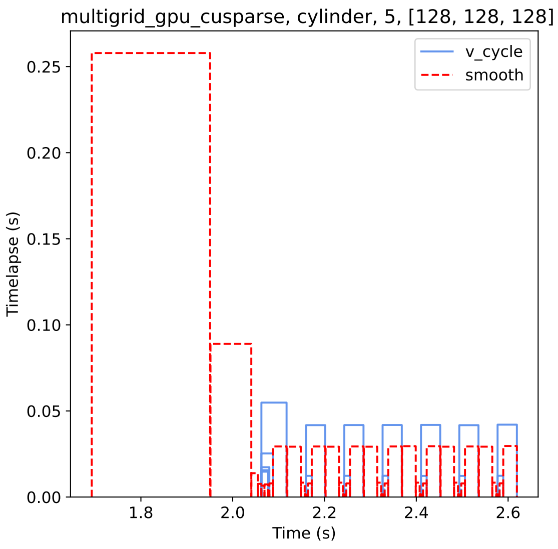
Figure 6: Comparison between GPU time for JIT rawkernels and built-in CuPy functions for a) dense vector subtraction, b) sparse matrix-dense vector multiplication, and c) sparse triangular solver. The functions are all tested using randomized matrices, with the sparse matrices having 10 % non-zero elements. The x-axis shows the number of elements in the vector of each operation.

3.2.4 GPU: cuSparse

As explained in Sect. 2.4, by implementing lower-level CUDA (cuSparse) libraries in the multigrid method, Fig. 3 (lower right) shows the time costs of different cases with different configurations. Time is greatly reduced compared to previously mentioned implementations. Fig. 2 shows the comparison of time costs between different implemented multigrid methods. It is clear that the lower-level CUDA (cuSparse) libraries greatly reduce the time. Detailed profilings showing V-cycle and smooth for both CuPy and cuSparse libraries implemented multigrid method are presented in Fig. 7a and Fig. 7b. Improvement with `cuSparse` libraries of *smooth* steps can be observed.



(a) Profiling from CuPy implementation.



(b) Profiling from cuSparse implementation.

Figure 7: Detailed profiling from both CuPy and cuSparse implementations. Grid size: 128x128x128. maxlevel=5

4 Discussion

The usage of CuPy resulted in a substantial overall speedup - 5 times for the slowest GPU implementation (JIT rawkernel), 82 times for the standard CuPy implementation, and 300 times for the cuSparse implementation. However, a potential weakness in this comparison is that only one CPU-based implementation was made, which employs a single-threaded triangular solver. The only CUDA-based implementation which uses a single-threaded triangular solver is the JIT rawkernel implementation. Multithreaded triangular solvers are in general not straightforward to design, and it is not clear if any existing libraries have triangular solvers would result in a more competitive CPU implementation. However, for specific problems with well-defined tolerances for the size of the residual, the most likely solution would be to employ a least-squares solver for the smoothing step. While such a solver might not be directly comparable to a direct solver due to its iterative nature, as as converging more slowly, it could likely be tuned to perform adequately. Thus, while such comparisons would require more work and need more assumptions than is within the scope of this project, it is plausible that the speedup in a real-world scenario would fall closer to the 5 times seen for the slowest GPU implementation than the 80-300 times seen with comparison to the faster implementations.

With this in mind, it is nevertheless remarkable that a translation that consisted of essentially direct substitution of CuPy operations for Numpy operations gave a speedup as large as 80 times. There are likely to be many situations where work is carried out with, for example, legacy code that is no longer maintained. If such code causes bottlenecks in computations, then the possibility of obtaining large speedups by simply substituting in CuPy could be a large benefit. In addition, it is common in scientific computing to write quick code drafts, not necessarily intended for long-term implementation, in order to test out an idea, examine a toy model, or probe the behaviour of a system. Again, in these cases CuPy offers the opportunity to examine systems at e.g. larger scales, greater resolution, or over greater timeframes with minimal additional work, compared to using Numpy.

However, when examining its limits, some aspects of the memory handling of CuPy appear inconvenient. While its universal functions do contain an `out` keyword, this does not in fact perform the operations in place. Instead, the functions allocate the memory as usual, and then replace the reference attached to the value given through they keyword. When CuPy encounters memory limits, it will simply throw an error, rather than attempting garbage collection or using workarounds. In some cases, such as sparse matrix-matrix multiplication, the amount of buffer space requested by the underlying CUDA libraries can be much larger than the actual matrix. Given how common memory limitations are when performing GPU computations, the usability of CuPy would be far improved if it had built-in features (through e.g., contexts, configuration parameters or keywords) which attempt to handle such cases by, for example, splitting up the matrix or even evaluating the operation on the CPU and copying it to the GPU. Instead, each case needs to be handled individually by the developer.

The creation of custom kernels gives more flexibility in the computations that can be carried out. However, tests suggested that these were significantly slower than using built-in functions. It is unclear whether this is because of differences in how the kernels are compiled, or because of the way in which they were written. The documentation on writing efficient kernels is limited, and while low-level function calls to CUDA are possible, there are no higher-level abstractions of common usage patterns available. In terms of abstraction and documentation, CuPy kernels compare somewhat unfavourably to Numba CUDA kernels. However, at the time of this writing, Numba does not natively implement bindings for CUDA linear algebra libraries, which in some cases makes it less convenient to use.

The restricted scope of CuPy in terms of providing an easy-to-use API is a barrier to optimization. Exposure and caching of the analysis results of the sparse solver, for example, requires the developer to write a custom function employing BLAS semantics through the bindings used by CuPy's submodules. It would have been preferable to be able to expose such features through e.g. functions and class objects which employ cuPy semantics, similar to `scipy.linalg.blas` and `scipy.linalg.lapack`. While certain functions from cuBLAS and cuSPARSE are relatively convenient to access through CuPy's submodules, this option is not made clear in the CuPy documentation, leaving future maintenance of these possibilities uncertain.

The large amount of buffer space needed for sparse matrix-matrix multiplication was a limiting factor together with the difficulty of accessing certain aspects of the underlying CUDA libraries. The computation of the approximate restricted Poisson matrix required too much buffer space to go beyond a 128^3 grid. While it was possible to simply carry out this computation on the CPU in the simple cuPy implementation, CPU-GPU copying resulted in instability for the cuSPARSE implementation, for reasons that are not clear. It would have been possible to simply define the exact Poisson matrix for each of the grid sizes, however, tests carried out early in the project had suggested that this resulted in poorer convergence for the system. For this reason, we opted to restrict the size of the system rather than working around this limitation.

References

[1] Wim van Rees et al. "High performance CPU/GPU multiresolution Poisson solver". In: *Advances in Parallel Computing* 25 (Jan. 2014), pp. 481–490. DOI: 10.3233/978-1-61499-381-0-481.

[2] Mitja Jančič, Jure Slak, and Gregor Kosec. "GPU accelerated RBF-FD solution of Poisson's equation". In: *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. 2020, pp. 214–218. DOI: 10.23919/MIPRO48935.2020.9245221.

[3] Wikipedia. *Multigrid method — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Multigrid%20method&oldid=1106856721. [Online; accessed 23-November-2022]. 2022.

[4] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[5] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[6] Ryosuke Okuta et al. "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations". In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. 2017. URL: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

[7] Wikipedia. *Finite difference coefficient — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Finite%20difference%20coefficient&oldid=1121699974. [Online; accessed 23-November-2022]. 2022.

A Discretization of the Poisson equation

The goal is to discretize Eq. (1) so that it can be written as a matrix problem $A\boldsymbol{x} = \boldsymbol{b}$.

Consider a grid consisting of N_x points in the x-direction, N_y points in the y-direction, and N_z points in the z-direction, with uniform spacing h between the grid points. The grid is thus defined by the points

$$\vec{r}_{ijk} = (x_0, y_0, z_0) + (i \cdot h, j \cdot h, k \cdot h),$$

where the indices i, k, and k span between 1 and N_x, N_y and N_z, respectively. (x_0, y_0, z_0) defines one of the corners of the grid.

As our grid is finite, we must make an assumption of what happens outside the grid. These are our boundary conditions. We will assume zero-boundary conditions, i.e. that

$$\phi(\vec{r}) = 0, \text{ for } \vec{r} \text{ outside our grid.}$$

We need to store the potential that is the solution to our problem, and the source term, at each grid point. Introduce the notation

$$\begin{aligned} f_{ijk} &= f(\vec{r}_{ijk}), \\ \phi_{ijk} &= \phi(\vec{r}_{ijk}). \end{aligned}$$

The Laplace operator needs to be discretized. Introduce the notation $\Delta\phi_{ijk}$ for the Laplace operator acting on the potential ϕ, evaluated at the grid point $\vec{r}_{ijk}$. Using a second-order-accuracy central difference[7] approximation to the second derivatives

$$\begin{aligned} \Delta\phi_{ijk} =& \frac{1}{h^2}\big[\,(\phi_{i-1,j,k} - 2\phi_{ijk} + \phi_{i+1,j,k}) + \\ & (\phi_{i,j-1,k} - 2\phi_{ijk} + \phi_{i,j+1,k}) + \\ & (\phi_{i,j,k-1} - 2\phi_{ijk} + \phi_{i,j,k+1})\,\big] + \mathcal{O}(h^2) \\ =& \frac{1}{h^2}\left[\phi_{i-1,j,k} + \phi_{i,j-1,k} + \phi_{i,j,k-1} - 6\phi_{ijk} + \phi_{i+1,j,k} + \phi_{i,j+1,k} + \phi_{i,j,k+1}\right] + \mathcal{O}(h^2). \end{aligned}$$

We can thus express the Laplace operator acting on the potential ϕ, evaluated at the grid point $\vec{r}_{ijk}$, in terms of the potential ϕ_{ijk} at the same grid point and at the neighboring grid points $\phi_{i\pm1,j\pm1,k\pm1}$.

Thus we can write the Poisson equation in the form

$$\sum_{i'j'k'} A_{ijki'j'k'} \cdot \phi_{i'j'k'} = f_{ijk}, \tag{3}$$

where $A_{ijki'j'k'}$ is a sparse matrix of finite difference coefficients

$$A_{ijki'j'k'} = \begin{cases} -6/h^2 & \text{, if } i = i',\ j = j' \text{ and } k = k' \\ 1/h^2 & \text{, if } |i - i'| = 1,\ j = j' \text{ and } k = k' \\ 1/h^2 & \text{, if } i = i',\ |j - j'| = 1 \text{ and } k = k' \\ 1/h^2 & \text{, if } i = i',\ j = j' \text{ and } |k - k'| = 1 \\ 0 & \text{, otherwise} \end{cases}$$

Now, what is special about Eq. (3)? It is simply a matrix equation

$$A\boldsymbol{x} = \boldsymbol{f}, \tag{4}$$

where $\boldsymbol{x}$ and $\boldsymbol{f}$ are vectors containing all the ϕ_{ijk} and f_{ijk}'s. One has to "flatten" the ijk indices into one dimension. Similarily, A is the matrix of all $A_{ijki'j'k'}$.

$$\boldsymbol{x} = \begin{bmatrix} \phi_{111} \\ \phi_{112} \\ \phi_{113} \\ \vdots \\ \phi_{11N_z} \\ \phi_{121} \\ \phi_{122} \\ \phi_{123} \\ \vdots \\ \phi_{12N_z} \\ \vdots \\ \phi_{N_xN_yN_z} \end{bmatrix}, \boldsymbol{f} = \begin{bmatrix} f_{111} \\ f_{112} \\ f_{113} \\ \vdots \\ f_{11N_z} \\ f_{121} \\ f_{122} \\ f_{123} \\ \vdots \\ f_{12N_z} \\ \vdots \\ f_{N_xN_yN_z} \end{bmatrix}, A = \begin{bmatrix} A_{111111} & A_{111112} & \cdots & A_{111N_xN_yN_z} \\ A_{112111} & A_{112112} & \cdots & A_{112N_xN_yN_z} \\ A_{113111} & A_{113112} & \cdots & A_{113N_xN_yN_z} \\ \vdots & \vdots & \ddots & \vdots \\ A_{11N_z111} & A_{11N_z112} & \cdots & A_{11N_zN_xN_yN_z} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N_xN_yN_z111} & A_{N_xN_yN_z112} & \cdots & A_{N_xN_yN_zN_xN_yN_z} \end{bmatrix}.$$