Chalmers University of Technology

Implementing GPU acceleration into the pyCALC-LES code using CuPy

LES and DES using an in-house Python code (FMMS002)

Author:
Johannes Hansson

Supervisor:
Prof. Lars Davidson

January 8, 2024

CHALMERS
UNIVERSITY OF TECHNOLOGY

Abstract

A proof-of-concept implementation of full-code hardware acceleration using Graphics Processing Units (GPUs) is presented for the Large Eddy Simulation software pyCALC-LES [1]. The original version of the code is a CPU-only implementation in the Python programming language. A recent study introduced GPU acceleration when solving systems of equations in the code, resulting in a factor two increase in whole-program computational performance [1]. The current study shows that going from partial GPU acceleration to the full-code equivalent can have yet another significant impact on computational performance, with a factor 15 improvement over the CPU-only case and an almost factor 3 improvement over the partial GPU case on a studied benchmark system.

Keywords: CFD, GPU, CuPy, NumPy, SciPy, Python

Acronyms

CFD Computational Fluid Dynamics.

CPU Central Processing Unit.

DNS Direct Numerical Simulation.

GPU Graphics Processing Unit.

LES Large Eddy Simulation.

RANS Reynolds-Averaged Navier Stokes.

VRAM Video Random Access Memory.

Contents

1 Introduction

There is large academic and industrial interest in understanding how fluids flow and behave. The two main approaches for understanding flow phenomena are experimental and computational studies. Experimental studies are often very accurate in the variables they measure, but suffer from high capital costs and provide limited information about the system being studied. Computational studies, on the other hand, are often cheaper to run and can provide much more information about the flow, but at the cost of practical difficulties that might affect the accuracy of the results. In the present study we focus on computational studies of fluid systems.

In order to get accurate results from computer simulations, it is desirable to resolve as much of the underlying physics as possible. This can in theory, but not in practice, be achieved using Direct Numerical Simulation (DNS), a simulation method that resolves all of the physics in the Navier-Stokes equations down to the smallest relevant spatial scale, the Kolmogorov length scale. This high accuracy gives very good results, but due to the often immensely high computational cost it is only feasible for very small and simple systems. Using this method for industrial-scale systems is out of the question with currently available computational power. To combat this issue, several other methods have been developed that model, instead of resolve, some of the underlying physics. Large Eddy Simulation (LES) resolves all but the smallest length scales in the flow, and can in many cases be used as a more practical alternative to DNS. It has slightly lower accuracy, but significantly lower computational requirements. This lower cost means that it is possible to use this modeling technique for some industrial-scale systems, as long as they are not too complex. There are also other computational methods that further decrease the computational cost, with a corresponding decrease in accuracy. One such method is Reynolds-Averaged Navier Stokes (RANS) simulations, which is one of the standard computational methods used to study industrial-scale systems.

Since computational power is the main factor limiting the use of increasingly accurate simulation methods, it is important that available computational resources are used as efficiently as possible. In this report we focus on how we can increase the computational performance of the research and education Computational Fluid Dynamics (CFD) code pyCALC-LES [1]. Even though the results presented in this report are specific to this particular code and the RANS case set-up described in section 3.3, the general workflow and trends in the results should transfer well to other similar codes.

2 Background

2.1 GPU acceleration

The pyCALC-LES code is written in the Python programming language and uses a small set of external libraries, with NumPy [2] and SciPy [3] being two of the more heavily used libraries. NumPy is used for numerical calculations of a wide set of mathematical operations, with a particular focus on array operations. SciPy is a collection of algorithms, data structures and other tools often used in scientific computing. These libraries run on the Central Processing Unit (CPU), which is able to deliver decent computational performance for practically any type of operation. However, some computations can be completed faster if performed on specialized hardware, such as GPUs. GPUs are especially well-suited for handling simple mathematical operations that operate on large sets of data. Since CFD simulations spend a large fraction of the simulation run time solving systems of equations, for example the pressure Poisson equation, GPU acceleration is a compelling option for increasing computational performance.

Partial GPU acceleration has already been implemented in previous work based on pyCALC-LES [1]. In that work, most of the simulation logic is still handled by the CPU, but the task of solving systems of equations is specifically offloaded to the GPU. This means that the most computationally expensive operation is performed with hardware acceleration, while most of the source code remains untouched, which is a positive aspect in terms of code complexity. Benchmark cases indicate that offloading the solution to the system of equations gives a performance increase of a factor more than two [1].

An obvious follow-up question to the partial-GPU case is if we can further increase performance by running everything on the GPU. The underlying idea here is that continuous copying of data back and forth between main memory and Video Random Access Memory (VRAM) takes time, which decreases overall computational performance. If all operations are performed on the GPU, then no copying needs to take place. A downside with GPUs is that they are not ideal for handling mathematical operations that act on small data sets, such as a single, or a small set of numbers. We might then need to accept that some operations are a bit slower on GPUs in exchange for getting faster solutions to systems of equations, and not having to move around large amounts of data in each solver iteration.

There are several libraries that can be used when porting software for running on a GPU, but many of them would require a complete, or almost complete, re-write of pyCALC-LES, something we want to avoid. Fortunately, one of these libraries, CuPy [4], is specialized in accelerating NumPy code so that it runs on GPUs with almost no change in the application source code. We will therefore use the CuPy library to develop a proof-of-concept implementation of the pyCALC-LES solver to evaluate possible performance gains.

2.2 The CuPy library

The CuPy library is designed to accelerate NumPy and SciPy operations by running them on GPUs. CuPy function calls are almost identical to the NumPy and SciPy equivalents, with the exception that they use a different module. For example, consider the NumPy code

```
>>> import numpy as np
>>> x_cpu = np.arange(5)
>>> x_cpu
array([0, 1, 2, 3, 4])
```

Listing 1: NumPy code for creating a five-element array.

This code loads the NumPy module and then uses it to create a five-element array `x_cpu` that is located in main memory, and is used by the CPU. If we compare this to the corresponding CuPy implementation we get:

```
>>> import cupy as cp
>>> x_gpu = cp.arange(5)
>>> x_gpu
array([0, 1, 2, 3, 4])
```

Listing 2: CuPy code for creating a five-element array.

Note that we are importing `cupy` with the module alias `cp`. This code generates the same five-element array as in the NumPy case, but this time it is stored in GPU memory. The CuPy library also includes convenience functions for easily translating between CPU and GPU arrays, but doing this implies copying data between main and GPU memory, something that takes time, especially for larger datasets.

As we can see from the two code listings above, the codes are almost identical, except for the `import` statement and the module alias, `np` or `cp`. Since CuPy is designed to be as similar to NumPy and SciPy as possible, it is relatively straightforward to use CuPy as an almost drop-in replacement for these libraries, with GPU acceleration built-in. Do note, however, that it is only an *almost* drop-in replacement. Some library features are not implemented in CuPy at the time of writing. Fortunately, this limitation generally only entails a small re-write of a single function call. Further details on this are presented in section 3.2.

2.3 PyAMG and PyAMGX

The original code uses the PyAMG library [5] to solve the systems of equations as efficiently as possible. This Python library is a collection of Algebraic Multigrid (AMG) solvers that use, as the name implies, the multigrid method to solve the systems of equations. In essence, the AMG method rewrites the original problem into a similar, but coarser, problem that is easier to solve. The solution to the coarse problem is then used to solve the original problem, a procedure that gives very computationally efficient solutions.

In the partially GPU-accelerated implementation of pyCALC-LES, the code uses the similar library PyAMGX [6], which is a set of Python bindings to Nvidia's AMGX [7] library for solving systems of equations using the multigrid method on GPUs. The rest of the code in that version of pyCALC-LES is still based on the original CPU-only NumPy code. In the fully GPU-accelerated version of the code presented in this report, the PyAMGX solver is kept as the main solver for the systems of equations. However, the CPU-only NumPy code has been replaced with GPU-accelerated CuPy code for increased computational performance.

3 Method

The main part of this work is divided into two categories, performance profiling and CuPy implementation. Performance profiling is needed for all three versions of the code: CPU only, partial GPU and full GPU. This aspect is presented in section 3.1 below. The other main part is the actual CuPy implementation of the pyCALC-LES software, presented in section 3.2.

3.1 Performance profiling

Performance profiling is carried out in several steps in order to assess the effect of various optimizations and changes in the source code. The first step is to establish a performance baseline using the original, CPU-only, version of the code. Next, the recently developed partial-GPU implementation is benchmarked. Results from previous work indicate that the expected performance increase for the partial GPU version over the CPU version should be in the order of about a factor two speedup for the simulation as a whole [1]. Finally, identical measurements are made for the full GPU implementation.

In this study we use a particular validation case from the pyCALC-LES manual [1] as our basis for simulation run time measurements, see section 3.3 below. Simulation run time is measured as a function of the number of grid cells. The physical system remains the same in all cases, but the resolution of the computational grid, and thereby the computational requirements, are increased step-by-step in order to quantify how an increasing number of grid cells affects the performance for the three solver implementations. In order to eliminate code start-up effects from affecting the measured run times, a small dummy simulation is run before each timed simulation

case. The run time results from these dummy cases are not taken into account in the final evaluation.

3.2 Implementation of pyCALC-LES in CuPy

Given the almost drop-in nature of replacing NumPy and SciPy with CuPy, the most important changes in the code relate to module imports in Python. As is presented in section 2.2, replacing NumPy and SciPy imports with their CuPy counterparts automatically transfers all computations and memory allocations to the GPU.

In order to minimize the number of changes needed to the original source code, we can replace the import statement in Listing 1 with

```
>>> import cupy as np
>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4])
```

Listing 3: NumPy code for creating a five-element array, with GPU acceleration using CuPy.

This way, only a single line needs to be changed to get full-GPU acceleration. Note that in this case the `x` variable is stored in VRAM on the GPU. For improved readability it is, however, suggested to replace the `np` alias with something that is not as generally associated with NumPy. It could, for example, be `cp` to indicate its association with CuPy or `xp` to indicate that both NumPy and CuPy are possible, depending on which library the user decides to use. The alias `xp` is used in the final full-GPU implementation of pyCALC-LES.

Even though the CuPy library is compatible with most parts of the NumPy and SciPy interfaces, some aspects are still not entirely compatible. This means that some extra source code modifications must be made. Luckily, most of these changes belong to a small number of categories, so the same workaround can be applied in several places. The most significant changes introduced to the pyCALC-LES source code are summarized below.

- CuPy does not implement the `np.matrix.flatten(A)` function call, which collapses a multidimensional array into a single dimension. Here, `A` is a given multidimensional array (for example a matrix in two dimensions). However, CuPy does implement `A.flatten()`, so calls of the form `np.matrix.flatten(A)` are in the pyCALC-LES source code replaced with calls to `A.flatten()`. Both function calls perform the same operation, so it produces no difference in the behavior of pyCALC-LES and NumPy compatibility is not affected.
- CuPy does not implement the NumPy `insert` function. This particular function call is in the code used when computing the pressure. It can be seen in, among others, the call

  ```
  p3d_w=np.insert(p3d_w,0,np.zeros((nj,nk)),axis=0)
  ```

  The above line of code pads the array `p3d_w` with zeros on one of the edges. A workaround for this category of operations is to use the `concatenate` function, which is indeed implemented in CuPy. This way, we can achieve the same functionality using a call such as

  ```
  p3d_w=xp.concatenate((xp.zeros(1,nj,nk)),p3d_w,axis=0)
  ```

  Note that we here use the alias `xp` to indicate compatibility with both NumPy and CuPy.

- There are a few other trivial compatibility issues that can affect NumPy codes running with CuPy, but most of them are of low impact, such as CuPy implementing a slightly different behavior for the `max`, `save` and `savetxt` functions. These inconsistencies are easily fixed with small changes to the source code.

A complete list of required source code changes is presented in Appendix A. Also, a full set of case directories and pyCALC-LES implementations for the current benchmark case is attached to this report as accompanying files.

3.3 Benchmark case

The validation case used in this study is based on the first workshop tutorial case presented in [1]. It is a RANS channel flow simulation with the $k-\omega$ turbulence model and a Reynolds number of $Re = 1200$. The only difference between the benchmark case and the tutorial case is that the computational mesh is slightly different. In this report, a much higher total number of grid cells is used to properly capture the performance differences between CPUs and GPUs. A grid with 2048 cells in the spanwise direction across the channel (the N_j direction, see [1]) is then used instead of the only 96 cells used in the tutorial case. To compensate for the increased number of cells, the thickness stretching per cell layer is decreased from 15 % to 0.3 %. The artificially increased number of grid cells is needed because GPUs need a certain smallest problem size in order to work at peak efficiency. Fine grids are also consistent with complex physical systems, so the procedure of using a large number of grid cells is representative of many real-world problems. The mesh is 2D rectangular, which means that the total number of computational cells is $N_i \times N_j$. In this work we keep a constant $N_j = 2048$, and let N_i assume values of the power of 2 between 2 and 9, i.e. $N_i \in [2^2, 2^9] = [4, 512]$. This gives total cell counts between 8192 and almost 1.05×10^6. The upper limit was chosen based on the maximum grid size that could be stored and processed by pyCALC-LES in VRAM of an Nvidia GTX1660 SUPER (6 GB).

4 Results

A comparison of simulation wall-clock run time as a function of the number of grid cells for the benchmark case is presented in Figure 1. The blue circles represent the original CPU-only version of the code, the orange triangles represent the previously added partial GPU acceleration in the solving of the system of equations, and finally the green squares represent the full-code GPU acceleration that is the main product of the current work. Note that both the x- and the y-axes are logarithmic and that horizontal, dashed, lines have been added as a guide for the eye to indicate 2 min, 20 min and 200 min.

As can be seen from the picture, the original CPU-only version of the code requires the most amount of simulation run time across all sampled grid sizes. The partially GPU-accelerated version is noticeably faster, and the fully GPU accelerated version is the fastest. For the second-to-largest mesh tested, the partially GPU-accelerated version is more than a factor five times faster than the original CPU-only version, and the fully GPU-accelerated version is almost a factor 15 times faster than the reference CPU-only version (see Figure 2). The full-code GPU acceleration is almost three times faster than the partial GPU implementation.

Unfortunately, the CPU-only version of the code had trouble running the largest mesh. It is believed that this was caused by factors external to pyCALC-LES, but the exact cause is unknown at the time of writing. It is suggested to rerun this particular measurement to get an accurate estimation of the run time requirements for the CPU implementation on the largest mesh as well.

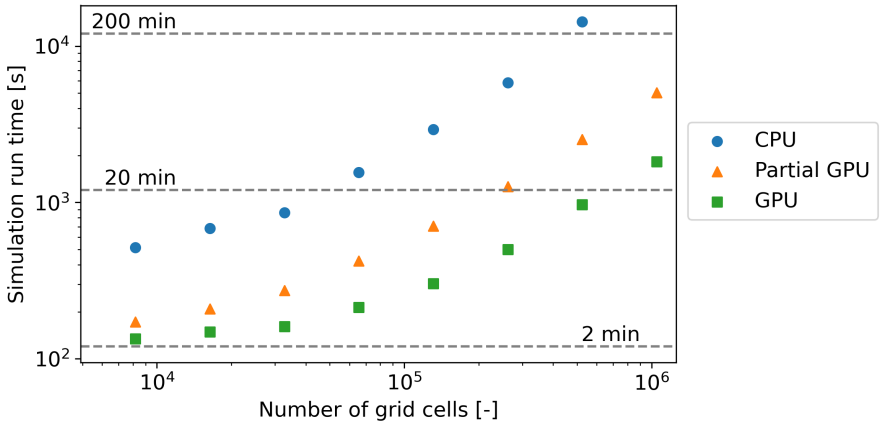

Figure 1: Simulation run time as a function of number of grid cells. Three cases are presented: CPU only, partial GPU and full GPU acceleration (this work).

A normalized plot of the achieved speedup is presented in Figure 2. The speedups are normalized against the CPU-only simulation wall-clock time. As can be seen in the figure, the existing partial-GPU implementation achieves a factor 3 to 5 speedup relative to the CPU only case. Similarly, the full-GPU case achieves speedups of a factor 4 to 15.

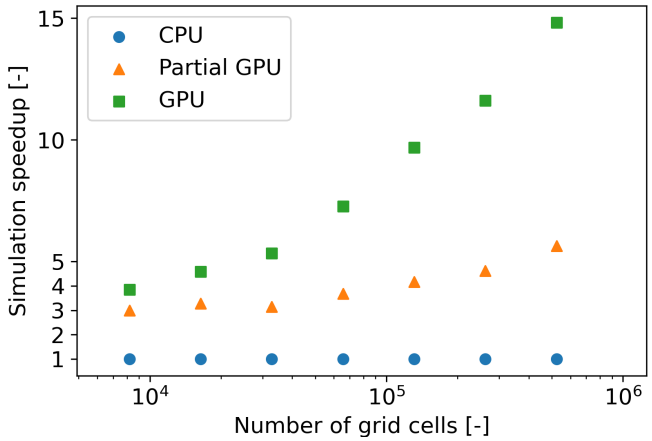

Figure 2: Speedup relative to the original CPU-only implementation.

5 Discussion

In Figure 1 we see a qualitative difference in the scaling behavior of especially the CPU-only version and the full-GPU version. The run time requirements for the CPU version start to increase almost immediately as the number of grid cells goes up. The GPU version, on the other hand, exhibits a somewhat flat run time response for the cases

with a small number of grid cells (less than approximately 50 000 cells). This behavior is expected for both computing devices, as CPU run time tends to scale directly with increasing computational load, whereas a GPU, due to its inherent design, tends to have a minimum amount of work that is needed before the device is fully utilized. This is indicated by the relatively flat time response in the beginning, followed by an increase in run time as the number of grid cells reaches a certain threshold, in this case at around 50 000 cells.

From Figure 1 it is also clear that the CPU-only version exhibits a sharper increase in simulation run time as the number of grid cells is increased beyond about 50 000 cells. A possible reason for this behavior could be related to exceeding the size of the CPU caches, but this is speculation. It is unknown if the changes in scaling behavior at 50 000 cells of the CPU and full-GPU cases are related. Worth noting is that the partial GPU implementation does not exhibit a sudden change in scaling behavior.

In Figure 2 we can see that the speedups of the partial-GPU and full-GPU implementations continue to rise as the number of grid cells increase. It is expected that the speedups will level off at constant factors for large enough meshes. Unfortunately, the measurements found in the figure do not indicate what these large-size speedup factors are. It is possible that the the largest mesh included in this study would give us approximations to these numbers, but since the CPU-only results for this mesh are unavailable we cannot do more than speculate.

One of the limitations in this study is that the number of RANS iterations is fixed at exactly 1000 for all cases. This is, unfortunately, not enough to achieve flow field convergence for the large-grid cases. However, all three solvers were subjected to the same number of iterations, so it does not affect the measured performance results. It would be beneficial to also compare the run times for fully converged cases, based on some convergence criterion. This would increase simulation run time for the large-grid cases, but it would be more indicative of computational performance in typical simulation use-cases.

Another limitation is that this study focused on a single RANS validation case found in the pyCALC-LES manual [1]. Further work is needed to extend the GPU implementation to also work for arbitrarily chosen RANS cases or transient LES and DNS simulations.

6 Conclusions

This study presents a proof-of-concept full-code GPU acceleration of the pyCALC-LES CFD code. The implementation uses the CuPy library, which is a GPU-accelerated implementation of the NumPy and SciPy libraries. The GPU accelerated code provides significant simulation speedups for the benchmark system used in this study. On the second-to-largest computational grid tested, the full-GPU implementation is almost 15 times faster than the original CPU-only version, and almost three times faster than the recently developed partial-GPU implementation.

This study is to some degree specific to the pyCALC-LES software, but the general principles applied can be used with any similar CFD code. The changes are particularly simple if the code is written in Python, which means that the CuPy library can be used as an almost drop-in replacement for many of the NumPy and SciPy features.

Acknowledgment

A special thanks to professor Lars Davidson for interesting discussions and ideas about how to use pyCALC-LES, and also for access to computational resources needed for generating the results in this study.

References

[1] Lars Davidson. *pyCALC-LES: A Python Code for DNS, LES and Hybrid LES-RANS*. Chalmers University of Technology. Göteborg, Sweden, 2023-07-18. URL: https://www.tfd.chalmers.se/~lada/postscript_files/py-calc-les.pdf (visited on 2023-07-18).

[2] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020-09), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[3] 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.

[4] 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.

[5] Nathan Bell et al. "PyAMG: Algebraic Multigrid Solvers in Python". In: *Journal of Open Source Software* 8.87 (2023), p. 5495. DOI: 10.21105/joss.05495. URL: https://doi.org/10.21105/joss.05495.

[6] Ashwin Srinath. *pyamgx*. GitHub repository. 2023. URL: https://github.com/shwina/pyamgx (visited on 2023-12-20).

[7] Nvida corporation. *AmgX*. GitHub repository. 2023. URL: https://github.com/NVIDIA/AMGX (visited on 2023-12-20).

Appendix A Implementation details

This section describes the needed changes for implementing full-code GPU acceleration starting from the `exec-pyCALC-LES.py` file generated in the first tutorial case of the solver manual [1].

A.1 High-level changes

The first and most impactful change is that we need to replace NumPy with CuPy. We therefore replace all the NumPy library imports with their CuPy equivalents. We therefore replace all occurrences of

```
import numpy as np
```

with

```
import cupy as xp
```

Note that we also replace the `np` alias with `xp` to indicate the double-library compatibility of the new code. Together with this we must also replace all function calls of the type `np.<...>` with `xp.<...>`. For example, a call such as `np.ones(ntstep)` should be replaced with `xp.ones(ntstep)`. This can easily be achieved by performing a search-and-replace operation on the entire code base, where we replace `np.` with `xp.` while being careful to include the dot at the end.

We must also replace the scipy imports such that all occurrences of the two lines

```
from scipy import sparse
from scipy.sparse import spdiags,linalg,eye
```

are replaced with the corresponding CuPy equivalent lines

```
from cupyx.scipy import sparse
from cupyx.scipy.sparse import spdiags,linalg,eye
```

The tutorial case uses the `lgmres` sparse linear solver for velocity and turbulence equations. To get full-GPU acceleration we instead use the `pyamgx` solver. Therefore,

```
solver_vel='lgmres'
solver_turb='lgmres'
```

becomes

```
solver_vel='pyamgx'
solver_turb='pyamgx'
```

Similarly, the tutorial case uses the `pyamg` solver when calculating the pressure. This is a CPU-only solver, so we replace it with `pyamgx`. Therefore,

```
solver_p='pyamg'
```

becomes

```
solver_p='pyamgx'
```

Finally, we need to be able to branch code behavior depending on if the code is running with the NumPy or CuPy libraries. As such, we still need to import NumPy, even if we do not really use it for calculations. We therefore add

```
import numpy
```

at the very first line of `exec-pyCALC-LES.py`.

A.2 Low-level changes

The changes outlined above represent the major steps needed to run pyCALC-LES with full-code GPU acceleration. However, there are some incompatibilities between CuPy and NumPy that have not yet been addressed by the CuPy developers. A full-GPU version of pyCALC-LES can nonetheless be achieved using the code rewrites and workarounds presented in the current section.

In the `modify_u` function, the line

```
l1=[itstep,k3d[1,5,0],k3d[1,10,0],k3d[1,20,0],k3d[1,30,0],k3d
    [1,40,0],\
    k3d[1,50,0],k3d[1,60,0]]
```

should be changed to

```
l1=xp.array([xp.array(itstep),k3d[1,5,0],k3d[1,10,0],k3d
    [1,20,0],k3d[1,30,0],k3d[1,40,0],\
    k3d[1,50,0],k3d[1,60,0]])
```

This change transforms the list into a NumPy/CuPy array instead of regular Python list. This change is needed because the CuPy `savetxt` function expects a CuPy array that it can convert to a NumPy array, and then call the NumPy version of `savetxt`. Unfortunately the CuPy `savetxt` function cannot handle the case of a regular Python list. This code change makes the data type conversion explicit instead of implicit.

A few small changes are needed in the `pyamgx` solver function `solve_pyamgx`. The code blocks

```
su=np.matrix.flatten(su3d)
phi=np.matrix.flatten(phi3d)
```

and

```
aw=np.matrix.flatten(aw3d)*acrank_conv_local
ae=np.matrix.flatten(ae3d)*acrank_conv_local
as1=np.matrix.flatten(as3d)*acrank_conv_local
an=np.matrix.flatten(an3d)*acrank_conv_local
al=np.matrix.flatten(al3d)*acrank_conv_local
ah=np.matrix.flatten(ah3d)*acrank_conv_local
ap=np.matrix.flatten(ap3d)
```

must be replaced with

```
su=su3d.flatten()
phi=phi3d.flatten()
```

and

```
aw=aw3d.flatten()*acrank_conv_local
ae=ae3d.flatten()*acrank_conv_local
as1=as3d.flatten()*acrank_conv_local
an=an3d.flatten()*acrank_conv_local
al=al3d.flatten()*acrank_conv_local
ah=ah3d.flatten()*acrank_conv_local
ap=ap3d.flatten()
```

because CuPy does not implement the `np.matrix` class. We must also implement two different versions of how to copy the solution vector back from the PyAMGX solver. The original command is

```
x_x.download(phi)
```

In the CuPy implementation we must, on the other hand use

```
if xp is numpy:
    x_x.download(phi)
else:
    x_x.download_raw(phi.data)
```

The if statement checks if we are running with the NumPy or CuPy library. In the first case we download the solution vector to the variable `phi` in main memory. In the second case we instead copy the solution data directly from the PyAMGX GPU memory to the CuPy array via a raw pointer. The structure with an if statement is needed to maintain both NumPy and CuPy compatibility.

In the `correct_conv` function, the three lines containing calls to the NumPy `insert` function should be replaced with equivalents using the `concatenate` CuPy function. The reason for this is that CuPy does not implement the `insert` function. Therefore, each of the lines

```
p3d_w=np.insert(p3d_w,0,np.zeros((nj,nk)),axis=0)
p3d_s=np.insert(p3d_s,0,np.zeros((ni,nk)),axis=1)
p3d_l=np.insert(p3d_l,0,np.zeros((ni,nj)),axis=2)
```

should be replaced with the corresponding `concatenate` alternative

```
p3d_w=xp.concatenate((xp.zeros((1,nj,nk)),p3d_w),axis=0)
p3d_s=xp.concatenate((xp.zeros((ni,1,nk)),p3d_s),axis=1)
p3d_l=xp.concatenate((xp.zeros((ni,nj,1)),p3d_l),axis=2)
```

The above mentioned `flatten()` substitution is also needed directly after calling the `correct_conv` function in the global iteration loop, where we replace the line

```
res_1d=np.matrix.flatten(su3d)
```

with

```
res_1d=su3d.flatten()
```

Towards the end of the global iteration loop, the line

```
resmax=np.max([residual_u ,residual_v,residual_w,residual_p])
```

should be replaced with

```
resmax=xp.max(xp.array([residual_u ,residual_v,residual_w,
    residual_p]))
```

This is due to a slight difference between NumPy and CuPy, where NumPy `max` function accepts regular Python lists whereas the CuPy equivalent does not.

In the `save_time_aver_data` function, the line

```
np.save('itstep',[itstep_stats_counter,nk,dz3d[0,0,0]])
```

must be replaced by

```
np.save('itstep',xp.array([xp.array(itstep_stats_counter),xp.
    array(nk),dz3d[0,0,0]]))
```

This change is needed because CuPy uses the NumPy `save` function internally, but CuPy has trouble interpreting how to convert the Python list into a NumPy array. This change does the conversion explicitly.

A.3 Switching the CuPy implementation back to NumPy

Once the low-level CuPy details have been implemented, switching between the two implementations is as simple as replacing a few import statements. The three lines

```
import cupy as xp
from cupyx.scipy import sparse
from cupyx.scipy.sparse import spdiags,linalg,eye
```

must then be replaced with

```
import numpy as xp
from scipy import sparse
from scipy.sparse import spdiags,linalg,eye
```

This change can easily be achieved with two search-and-replace operations in any text editor by replacing all occurrences of `cupyx.` with an empty string, and then replacing all occurrences of `cupy` with `numpy`.

Note that changing the imports gives us the partially GPU-accelerated version of pyCALC-LES since the solvers are still set to `pyamgx`. Changing these solvers to something else gives us the CPU-only version of pyCALC-LES.

A possible future improvement of the code constitutes implementing a simple switch that selects the correct NumPy or CuPy libraries based on a single boolean variable set by the user. This would make it easier for the user to swap backend libraries.

A.4 Note about the GPU implementation presented in this report

This report presents a proof-of-concept GPU implementation using CuPy for a tutorial case in the pyCALC-LES manual. Only the changes needed for running this particular benchmark case are presented here. Additional changes are needed if other cases are to be simulated as well. These changes are expected be quite similar to the the ones presented here, so it should be straightforward to get other cases to run using full-GPU acceleration.