

Implementation of a Domain Specific Language
Report for HPC4WC Project

Caroline von Mering & Jannis Portmann

August 31, 2023

Supervisor: Tobias Wicky

1 Abstract

Domain specific language (DSL) is a useful tool to allow for user-friendly code and to adapt existing code to different hardware architectures. We created a simple DSL based on an internal representation (IR) made up of different classes representing parts of the code. The source-code is first parsed to create an abstract syntax tree (AST) using the Python AST parser. Then, our language parser visits the AST and stores it in our IR. The IR gets visited by a visitor which will generate code that can be executed. We adapted our DSL to be able to parse and generate a simple stencil code. Then we validated and compared the performance of the base code to our generated code, which we implemented ran with NumPy (CPU) and once with CuPy (GPU). We got an average runtime of 0.27 ms for the base code, 150 ms ms for the generated code with NumPy and 13 676.50 ms for the generated code with CuPy. We suppose that the main reason for the slow runtime with CuPy are wait times due to scheduling and poor optimization. We also found that the CuPy version did not validate correctly, as it did not apply diffusion in the horizontal (y) direction. While the primary goal of our project was to create the DSL a next step would be to optimize it and to find the reason why the CuPy version did not validate.

2 Introduction

To simplify the usage of common tasks in weather and climate models, it can be worth while to implement a domain-specific language (DSL). Using a DSL, problems occurring often in the respective field can be represented in a simplified manner, since conventions and assumptions present can be used implicitly. This reduces the amount of work when implementing a problem by the user.

Apart from making weather and climate models more user friendly, DSL can be helpful to adapt existing code to new hardware architectures. New hardware architectures emerge very frequently, which poses a challenge for weather and climate models, that evolve rather slowly. Weather and climate models often include millions of lines of code, which are developed over decades and then used for decades. DSL can facilitate the process of adapting weather and climate models to new hardware architectures. The idea is to parse the existing code and transform it to maximize its performance for the target architecture. For instance, the CLAW DSL by Clement et al. (2018) makes different changes to the code depending if it is to be run on CPUs or GPUs (Clement et al. 2018).

2.1 Abstract Syntax Tree (AST)

Our toyDSL works as follows: At first, the code is parsed to create an abstract syntax tree (AST). An AST is a tree-like representation of the structure of the code. Each node in the tree stands for an operation and the subsequent nodes stand for the arguments of the operation (Aho et al. 2006). An example for an AST is illustrated in Figure 1.

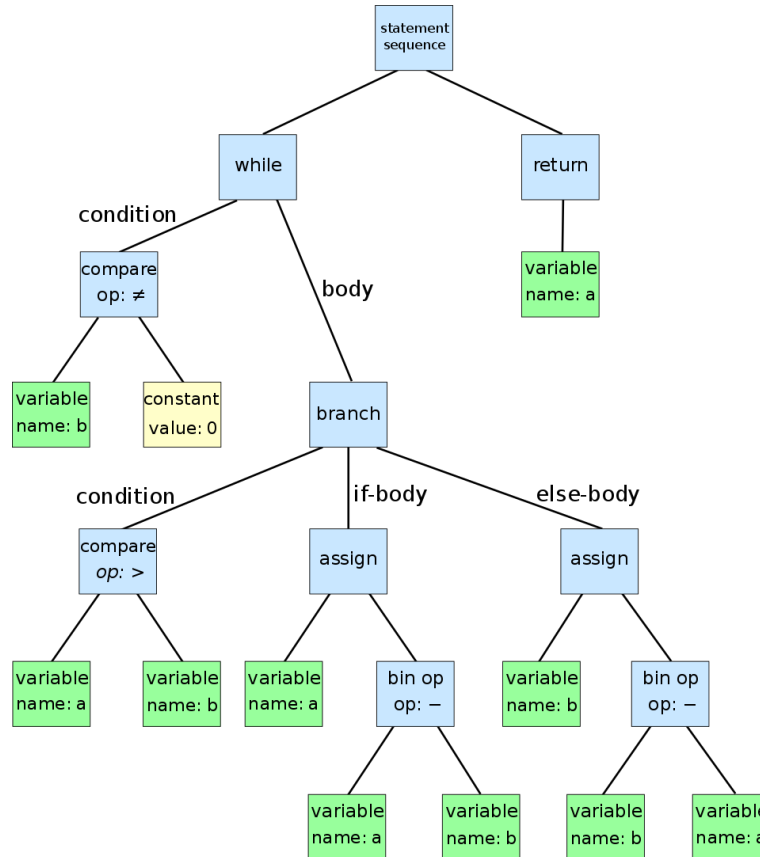


Figure 1: An example of an abstract syntax tree (AST) as it would look like for the following code:

```
1 while b != 0:
2     if a > b:
3         a = a - b
4     else:
5         b = b - a
6 return a
```

2.2 Internal representation (IR)

In a next step, the language parser visits the AST, and stores it in an internal representation (IR). An internal representation is intermediate code used by compilers which is later translated into the target program (Aho et al. 2006). In the case of our toy DSL, the internal representation consisted of a number of classes with distinct attributes.

For instance, the line `field_1 = 10` in the source code will be stored in the IR in the class assignment statement. This class has two attributes: left and right. The name of the field will be stored under `AssignmentStmt.left` and the value 10 under `AssignmentStmt.right`. There are also many other classes. For instance, if the source code is `Field_1 = lap(Field_2)` then `AssignmentStmt.right` will be `lap(Field_2)`, as expected. However, in this case `lap(Field_2)` will furthermore be stored in the class Function with `lap` as function name and `Field_2` as function argument. It is important to know that the values are only stored temporarily before they are used for code generation (see next step). If another Assignment Statement in the source code is visited, the previous values will be overwritten. All of the different classes used in the IR can be seen in the code in Appendix A.

2.3 Code Generation

In a final step, the IR gets visited by a visitor. The visitor goes through each class of the IR and checks if it has attributes (which were assigned to it in the previous step, depending on the source code). If it has attributes, code will be generated depending on the class. So, for the class assignment statement a "=" will be generated with a left- and right-hand side to be determined by further visits to the according classes. For the class function, code will be generated according to the name of the function and its arguments. This continues for all classes, until a pure string representing the node of the IR is returned. This then is added to the generated code in memory, which in turn is saved as file in the end.

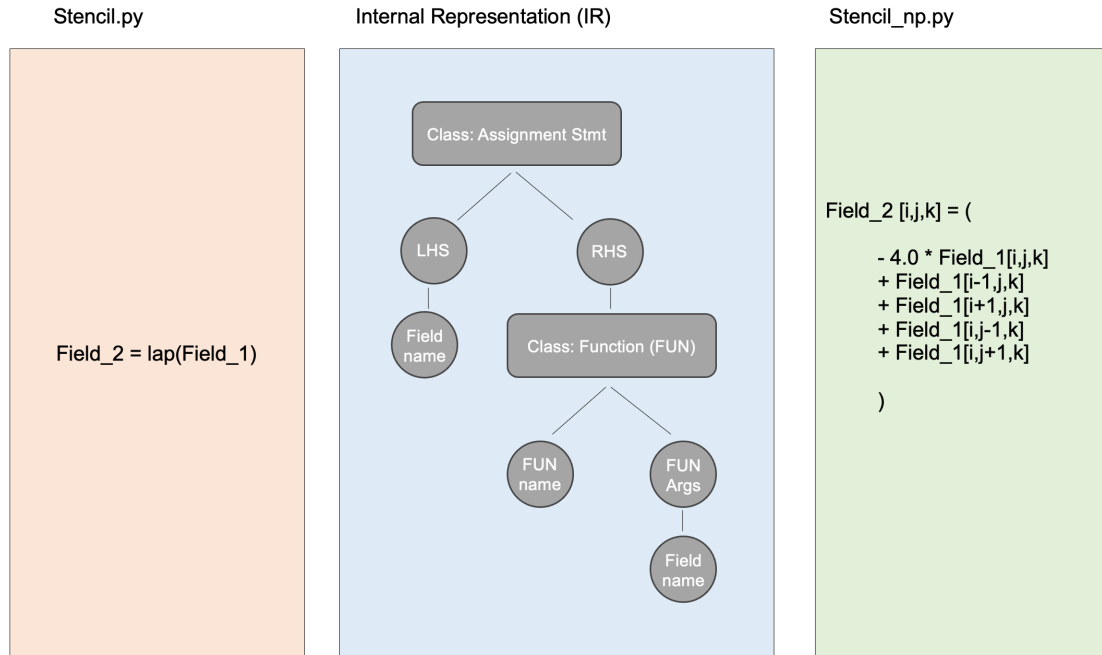


Figure 2: An illustration of the path from source code (Stencil.py) to the generated code (Stencil_np.py).

2.4 Comparison with NumPy and CuPy Implementation

The aim of this report is to compare the performance of a simple stencil code generated by our DSL with implementation of the same code using NumPy and CuPy. NumPy is an abundantly used python library for array handling, which operates on the central processing unit (CPU) (Harris

et al. 2020). CuPy is has the same syntax as NumPy but it operates on the graphic processing unit (GPU), which has become more and more popular in recent years (Nishino and Loomis 2017). GPUs were originally created for graphics applications, where each pixel on a screen can be processed independently. This means that GPUs can handle a large degree of data parallelism and are well suited for parallel computing. This can lead to significant speedup. However, the decision between using CPU or GPU is not always a simple one, as the optimal choice depends on the objective, and many optimizations exist for both (Lee et al. 2010). The stencil code generated by our DSL can be implemented with both NumPy and CuPy and it will be interesting to see how the two methods will compare.

3 Implementation

In this chapter, we will first explain in greater detail how some aspects of our DSL work and then elaborate how we set up the performance analysis to obtain our results.

3.1 User defined loop bounds using With Statements

We decided that it is useful to have user defined loop bounds, as they will not always be the same in out stencil program. For instance, the first laplacian requires slightly different loop bounds than the second. Since our DSL has to be based on legal Python Code, we used with statements to read the loop bounds. With statements offered us some flexibility in how to do this.

Our source code for a simple nested loop looks like this:

```
1 with Vertical[1:nz]:
2     with Horizontal[num_halo: ny + num_halo + 1, num_halo: nx + num_halo + 1]:
3         tmp_field[i, j, k] = lap(in_field)
```

"Horizontal" and "Vertical" are defined as legal language classes and as classes in the IR. In the IR, they have a body consisting of a list of statements as well as an extent, which store the start and stop values. The with statements get visited by the visit-With function. This function visits each type of with statement. Here an excerpt is shown for the case that the with statement is "with vertical". First, the start and stop values are appended to the class. Then, the Vertical node is created and the scope is set to it, which is necessary to visit the statements within the body of the Vertical node. In the end, the scope is set back to the origin.

```
1 def visit-With(self, node: ast.With) -> ir.Node:
2     expr = node.items[0].context_expr
3     if isinstance(expr, ast.Subscript):
4         if expr.value.id == "Vertical":
5             self._parent.append(self._scope)
6
7             assert len(node.items) == 1
8             extent = []
9             extent.append(self.visit(expr.slice))
10
11             # Create the Vertical node
12             self._scope.body.append(Vertical(extent))
13
14             # Set the scope to the body of the new Vertical node
15             self._scope = self._scope.body[-1]
16
17             for stmt in node.body:
18                 self.visit(stmt)
19             self._scope = self._parent.pop()
```

In the code generation, a simple for loop gets generated by the with statement using a generate loop bounds function:

```
1 def visit_Vertical(self, node: ir.Vertical):
2     self.generate_loop_bounds(node, "k")
3     for stmt in node.body:
4         self.visit(stmt)
5     self.dedent()
6
7 def generate_loop_bounds(self, node, variable):
8     self.code += self.get_indent()
9     self.code += f"for {variable} in range("
```

```

10 self.visit(node.extent[0].start)
11 self.code += ","
12 self.visit(node.extent[0].stop)
13 self.code += "):"
14 self.indent()

```

3.2 Running

To test our DSL, we wrote

3.3 Validation

To validate the intended solution, the output arrays are compared directly to the original `stencil_2d.py` output. Further, the output arrays are also validated graphically, to understand what is happening – especially if the arrays are not equal to the base output. Since precisions are not handled equally on both CPU and GPU, the NumPy function `np.allclose()`, which allows for specification of tolerances when comparing field and returns `True` if they are within that tolerance.

4 Results

We compared runtimes of the base version (the original `stencil.py`), and the generated code, which was run once with NumPy and once with CuPy. Shown in Table 1 are the mean runtime values after running the code 10 times, using `nx=20, ny=20, nz=20, num_iter=10`, meaning 20 cells in each spatial direction and 10 steps in time (integration time).

Implementation	Runtime (standard deviation) [ms]
Base (original stencil)	0.27 (0.02)
NumPy (generated)	150 (0.97)
CuPy (generated)	13676.50 (110.45)

Table 1: Runtimes

Figure 3 shows the results of the validation. Figure 3a shows the base version, Figure 3b the generated version using Numpy and Figure 3c the generated version using CuPy. It can be seen that the NumPy version has the same result as the base version, while the CuPy version did not. This is also supported by the numerical validation, where the output fields of base and NumPy match, but base and CuPy do not within the specified tolerance of `rtol=1e-5`, `atol=1e-8` (relative and absolute tolerance). Especially interesting is the fact, that in Figure 3c, it looks like the diffusion is only applied in the horizontal (y) direction, but not in the vertical (z) direction.

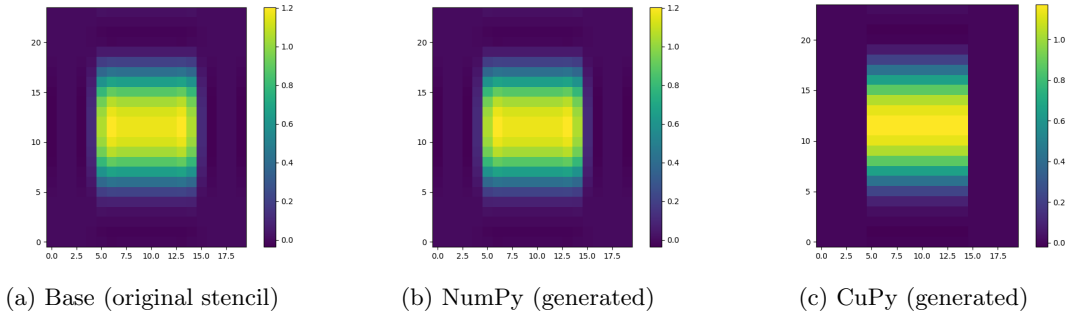


Figure 3: Visual validation using a yz cut through the 3D field at timestep `n=num_iter`

5 Discussion

Our aim was to compare the base version of a stencil code with our generated version using NumPy and CuPy. We found that the three versions performed very differently. When looking at the runtimes in Table 1, we were surprised to see that in both cases, our generated code performed

significantly worse than the base version. This was especially true for the version run with CuPy, which took over 13 seconds. This also includes the time for transferring the NumPy array to the GPU using `cp.asarray()` and back again from the GPU to the CPU using `out_field.get()`. This adds some overhead, but it should not be too big for the three $20 \times 20 \times 20$ 3D fields, which are only transferred before and after the stencil computation. We suppose that the main reason for the long runtimes are wait times due to scheduling and poor optimization. Especially the explicit loops over `i, j, k` could be improved by replacing them with element-wise notation where possible. This should significantly improve the performance on both GPU and CPU. Further thoughts should be given to how we can leverage parallelization efficiently on the GPU, since much of the performance benefits stem from running computations in parallel. Since this is not the core aspect of this project, showing that running code also on a GPU is possible will suffice.

During the validation, we noticed that the generated CuPy version also did not create the correct output field. The output field generated by the CuPy version did not seem to represent diffusion correctly in the y-direction (see Figure 3c). This surprised us, as the generated stencil code used was the same for both versions (see 'main' in Appendix). The difference only lies in the arguments of the generated function, being NumPy or CuPy arrays (which are copied from the NumPy arrays). We first thought, that the difference might come from rounding errors, but looking at Figure 3c, it seems like there is a more systematic error. However, we were not able to find what caused it.

6 References

- Aho, Alfred V. et al. (Aug. 2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley. ISBN: 0321486811. URL: <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20%5C&path=ASIN/0321486811>.
- Clement, Valentin et al. (2018). “The CLAW DSL: Abstractions for Performance Portable Weather and Climate Models”. In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC ’18. Basel, Switzerland: Association for Computing Machinery. ISBN: 9781450358910. DOI: 10.1145/3218176.3218226. URL: <https://doi.org/10.1145/3218176.3218226>.
- Harris, Charles R. et al. (2020). “Array Programming with NumPy”. In: *CoRR* abs/2006.10256. arXiv: 2006.10256. URL: <https://arxiv.org/abs/2006.10256>.
- Lee, Victor W. et al. (June 2010). “Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU”. In: *SIGARCH Comput. Archit. News* 38.3, pp. 451–460. ISSN: 0163-5964. DOI: 10.1145/1816038.1816021. URL: <https://doi.org/10.1145/1816038.1816021>.
- Nishino, ROYUD and Shohei Hido Crissman Loomis (2017). “Cupy: A numpy-compatible library for nvidia gpu calculations”. In: *31st confernce on neural information processing systems* 151.7.


A Code

A.1 main.py

```
1 import numpy as np
2
3 def generated_function(in_field, out_field, num_halo, nx, ny, nz, num_iter,
4 tmp_field, alpha):
5     for n in range(1,num_iter-1):
6         in_field[:,num_halo,num_halo:-num_halo] = in_field[:,-2*num_halo:-num_halo,
7 num_halo:-num_halo]
8         in_field[:,-num_halo:,num_halo:-num_halo] = in_field[:,num_halo:2*num_halo,
9 num_halo:-num_halo]
10        in_field[:,::num_halo] = in_field[:,:-2*num_halo:-num_halo]
11        in_field[:,:-num_halo:] = in_field[:,num_halo:2*num_halo]
12        for k in range(1,nz):
13            for j in range(num_halo,ny+num_halo+1):
14                for i in range(num_halo,ny+num_halo+1):
15                    tmp_field[i,j,k] = -4.0 *in_field[i, j, k] + in_field[i-1, j, k
16 ] + in_field[i+1, j, k] + in_field[i, j-1, k] + in_field[i, j+1, k]
17
18        for k in range(1,nz):
19            for j in range(1+num_halo,ny+num_halo):
20                for i in range(1+num_halo,ny+num_halo):
21                    out_field[i,j,k] = -4.0 *tmp_field[i, j, k] + tmp_field[i-1, j,
22 k] + tmp_field[i+1, j, k] + tmp_field[i, j-1, k] + tmp_field[i, j+1, k]
23
24        out_field[:,num_halo:-num_halo,num_halo:-num_halo] = in_field[:,num_halo:-
25 num_halo,num_halo:-num_halo]-alpha*out_field[:,num_halo:-num_halo,num_halo:-
26 num_halo]
27        tmp_field[:,:::] = in_field[:,:::]
28        in_field[:,:::] = out_field[:,:::]
29        out_field[:,:::] = tmp_field[:,:::]
30
31        in_field[:,num_halo,num_halo:-num_halo] = in_field[:,-2*num_halo:-num_halo,
32 num_halo:-num_halo]
33        in_field[:,-num_halo:,num_halo:-num_halo] = in_field[:,num_halo:2*num_halo,
34 num_halo:-num_halo]
35        in_field[:,::num_halo] = in_field[:,:-2*num_halo:-num_halo]
36        in_field[:,:-num_halo:] = in_field[:,num_halo:2*num_halo]
37        for k in range(1,nz):
38            for j in range(num_halo,ny+num_halo+1):
39                for i in range(num_halo,ny+num_halo+1):
40                    tmp_field[i,j,k] = -4.0 *in_field[i, j, k] + in_field[i-1, j, k] +
41 in_field[i+1, j, k] + in_field[i, j-1, k] + in_field[i, j+1, k]
42
43        for k in range(1,nz):
44            for j in range(1+num_halo,ny+num_halo):
45                for i in range(1+num_halo,ny+num_halo):
46                    out_field[i,j,k] = -4.0 *tmp_field[i, j, k] + tmp_field[i-1, j, k]
47 + tmp_field[i+1, j, k] + tmp_field[i, j-1, k] + tmp_field[i, j+1, k]
48
49        out_field[:,num_halo:-num_halo,num_halo:-num_halo] = in_field[:,num_halo:-
50 num_halo,num_halo:-num_halo]-alpha*out_field[:,num_halo:-num_halo,num_halo:-
51 num_halo]
52        in_field[:,num_halo,num_halo:-num_halo] = in_field[:,-2*num_halo:-num_halo,
53 num_halo:-num_halo]
54        in_field[:,-num_halo:,num_halo:-num_halo] = in_field[:,num_halo:2*num_halo,
55 num_halo:-num_halo]
56        in_field[:,::num_halo] = in_field[:,:-2*num_halo:-num_halo]
57        in_field[:,:-num_halo:] = in_field[:,num_halo:2*num_halo]
58        return out_field
```

A.2 Repository

The full code of this project is available at:

 HPC4WC  thisfro