TOY DSL FOR WEATHER AND CLIMATE SIMULATIONS

Clément Thorens, Josefine Leuenberger, Pascal Sommer

High Performance Computing for Weather and Climate
ETH Zürich, Switzerland

ABSTRACT

Numerical computations in weather and climate simulations often make use of stencil calculations. While there are several well-known techniques for optimizing stencil calculations, it can be tedious to always manually apply them when writing such programs. We can exploit the constrained nature of stencil calculations by creating a domain specific language (DSL) that allows the user to generate efficient code from a simple description of their intent.

1. INTRODUCTION

When predicting weather and climate with numerical methods, we consider a cube of the atmosphere bounded below by the ground, with a fixed width and length and height. The different dynamics of the weather are then mostly an exchange of some quantities between horizontally or vertically neighbouring grid cells. So instead of tediously writing loops over those grids every time, we create a domain specific language (DSL) that allows us to easily generate those loops. Additionally we want to run those applications efficiently. So instead of optimizing the loops by hand by going through (potentially thousands) lines of code while experimenting, we just do that in the DSL translation program that generates the loops for us. There we can add different optimizations to improve the performance. Later this can also be extended for different architectures so that the code gets more portable.

In our project we started from a simple DSL translation program that generated python code, and changed it to generate C++ code. Our goal then is to add performance optimizations to that generated C++ code. We try unrolling of loops, vectorization, and parallelizing with `openmp`. The user of the DSL then does not have to worry about optimization but still gets faster running code.

2. THE DSL

The domain specific language considered in this project is limited to the narrow scope of only stencil programs on arrays of equal sizes. This enables us to perform some interesting optimizations which would be much more difficult to do for a more general language and also ensures that the scope of the project doesn't explode into something unrealistic.

The language allows the user to define functions that take mutable references to one or more arrays, all of the same dimension. Typically, one would pass two arrays, one with input data which is only read from, and the other one as an output. It is also possible to read from or write to multiple arrays but for the sake of simplicity we decided that we don't explicitly support the case of both reading and writing to the same array withing a single inner loop. Interleaving reads and writes like that would prevent us from applying optimizations that reorder some of the operations. While it is still possible to do that, the user has to be aware of the vectorization transformation that is applied to the code.

Listing 1. Basic Laplace of Laplace Stencil

```
@computation
def lapoflap(out_field, in_field, temp_field):
  with Vertical[start : end]:
    with Horizontal[start+1 : end-1,
                    start+1 : end-1]:
      temp_field[0, 0, 0] = (
        in_field[1, 0, 0] +
        in_field[-1, 0, 0] +
        in_field[0, 1, 0] +
        in_field[0, -1, 0] -
        4 * in_field[0, 0, 0]
      )
    with Horizontal[start+1 : end-1,
                    start+1 : end-1]:
      out_field[0, 0, 0] = (
        temp_field[1, 0, 0] +
        temp_field[-1, 0, 0] +
        temp_field[0, 1, 0] +
        temp_field[0, -1, 0] -
        4 * in_field[0, 0, 0]
      )
```

The language supports two loop constructs: a loop over the vertical dimension, and a loop over both horizontal di-

The authors thank Markus Püschel and Jelena Kovacevic for provding this LaTeX template.

mensions. The user doesn't have to think about the resolution of the arrays at the time of writing, but instead can define the loop bounds in terms of the margins. For example, defining loop bounds as `[start + 1, end - 1]` skips the first and last element, looping only over $n - 2$ elements.

Horizontal loops can only be used inside vertical loops and it is not possible to nest multiple loops of the same kind. Inside the horizontal loops, assignment statements can be used to modify the arrays using expressions containing values from other arrays as well as float literals. The values inside the square brackets used for array access do not refer to absolute indices but rather to offsets.

2.1. Parsing the code

The starting point to build the C++ code generator we are presenting in this report is a Python code generator, toyDSL, provided by Tobias Wicky [1]. The original code makes use of the Python library dedicated to abstract syntax trees, ast.py [2], to parse the DSL code, from there on we can then transform it to a custom intermediate representation and later to some executable output. This is only possible because the DSL syntax is close enough to Python syntax so that the `ast.parse` function can just about handle it. Note however that the DSL code is not valid Python and going any step further in the Python interpreter beyond the abstract syntax tree generation would most likely result in errors.

We extend the parsing functionalities of toyDSL in order to process binary operations and minus signs. With this additions, we have a parser powerful enough to build a simple domain specific language and generate C++ code.

2.2. C++ code generation

After the DSL code is converted to the custom intermediate representation (IR), we still have to generate something that can actually be executed. In Tobias Wicky's original code, the IR is converted to Python code which is written to a file and loaded as a Python module at runtime.

We follow a similar concept with our C++ generator. The C++ code is generated in in memory and then written to a `.cpp` file. As a debugging convenience, we also apply Clang-Format to the generated code which makes it easier to read. The generated C++ code also contains some Boost, Python boilerplate code which allows us to then call the C++ function from Python code. That way we achieve the same seamless integration as the Python version but with all the added benefits of using a compiled language. The code is compiled into a shared object using CMake.

Compiling the C++ code usually takes multiple seconds, we make sure that this annoyance does not apply to repeated runs by first hashing the DSL code and reusing the generated shared object if it has already been generated before.

2.3. Optimizations

While generating the C++ code, we apply 3 modifications to the generated code to speedup the computation: unrolling the loops, use of vectorized instructions and shared memory parallelism. Note that all our optimizations happen during code generation, we don't apply any transformations to the IR before passing it through the generator.

2.3.1. Unrolling

We apply loop unrolling by keeping track of an unroll factor as we traverse the IR tree. In practice this factor is only modified in the inner loop of a `with Horizontal` block, and then used when traversing a list of assignment statements.

The implementation is straightforward, we modify the generated `for` loop to increment the loop index by the unroll factor instead of always by one. Inside the loop we generate multiple assignment statements with incrementing index offsets to cover the same cases that would have been covered by the unmodified loop.

Because the array dimensions are currently not considered to be known at compile time, we always have to generate a small loop for the remaining elements for the case when then array dimensions are not evenly divisible by the unroll factor. Our generator structure allows us to reset the unroll factor to one inside this remainder loop, which means that the statements are no longer repeated there and thus we don't have to add a special case to the code printing the list of instructions in the remainder loop.

2.3.2. Vector instructions

Given the structure of our unrolling implementation, we can easily combine it with the use of vector instructions. Since vector instructions work on multiple values at once, we can just divide the unroll factor by the corresponding value, after that the vectorizing transform works without having to implement any major changes on the IR tree traversal.

Our code generator uses AVX / AVX2 instructions, they are compatible with all architectures that we want to run the code on: our personal computers, as well as the Intel Xeon processors at CSCS.

Since we cannot guarantee that the data is aligned, we have to use unaligned instructions. This should however not be an issue on modern architectures. For writing results to a contiguous array we use `_mm256_storeu_pd`, reading is done with `_mm256_loadu_pd`. When literals are used we generate a `_mm256_set1_pd` instructions that repeats the given double value four times. The correct instructions for

basic arithmetic operators are automatically inserted by the compiler. Details about all these instructions can be found on the Intel Intrinsics Guide [3].

2.3.3. OpenMP

To take advantage of multi-core computer architectures, we use OpenMP to implement shared memory parallelism. Only the most outer loop (vertical dimension in our case) is parallelized via the pragma instruction *pragma omp parallel for default(none)*. To define if a field will be defined shared or private, we traverse the abstract syntax tree and determine if a field is both read and written and in this case, declare the field as private. For all other possibilities, only read, only written or neither, the field is declared shared.

2.3.4. Possible further optimizations

The here presented list doesn't exhaust all the possible optimizations that could be applied. One further idea would be to pass the array dimensions directly to the C++ generator, thus enabling the compiler to see if the array dimensions are evenly divisible by the unroll factor, meaning that no remainder loop has to be generated.

3. EXPERIMENTAL RESULTS

3.1. Validation

To ensure that the generated code is correct, we observe the result of two known operations on some data and verify that we obtain the expected result. We also verify that the code generated by a simple copy of the input fields into the output fields yields a correct result. Since the resulting plots are not interesting, we are not showing them here.

3.1.1. Basic Stencil code

We apply the basic stencil code given in the listing 1. On the figure 1 we have the input data for the first vertical slice consisting of a 256x256 field of zero values with a square of one values in the middle. After 2048 iterations, we obtain the figure 2. We observe a diffusion, which is the result expected after applying twice the laplace operator on the input data.

3.1.2. Vertical blur

Here we apply a vertical blur, shown in listing 2, to a cube of ones in the middle of a 256x256x256 space. The figure 3 shows a side slice of the space. The figure 4 shows the result after 2048 iterations of the vertical blur computation. Again the result is the one expected.

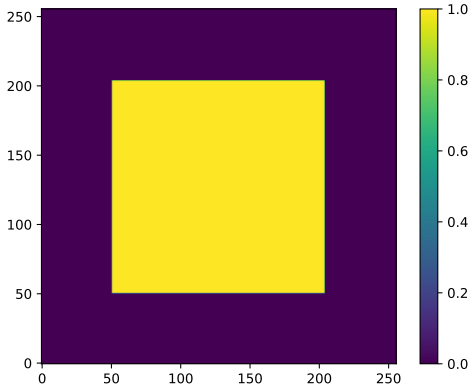

Fig. 1. Input for the basic stencil code

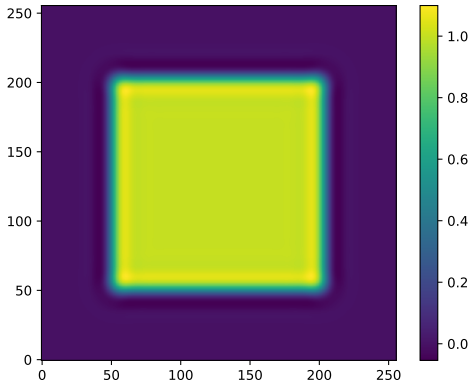

Fig. 2. Output of Figure 1 after 2048 iterations of the laplacian diffusion

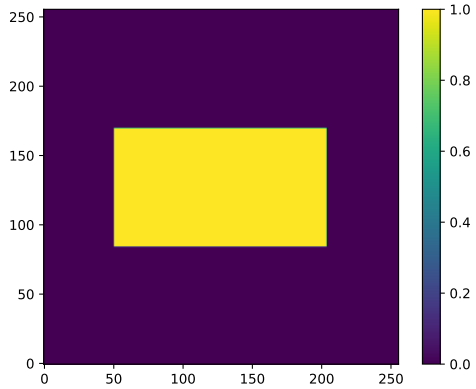

Fig. 3. Input for the vertical blur code

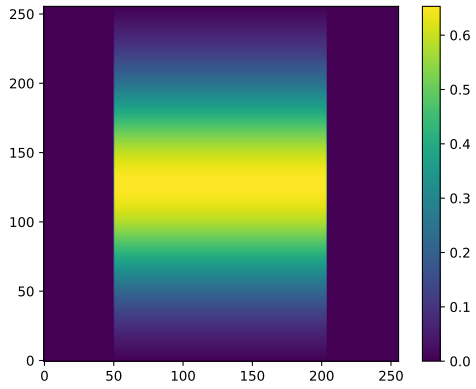

Fig. 4. Output of figure 3 after 2048 iterations of the vertical blur

Listing 2. Vertical blur

```
@computation
def vertical_blur(out_field, in_field):
    with Vertical[start+1:end-1]:
        with Horizontal[start : end,
                        start: end]:
            out_field[0, 0, 0] =
            (in_field[0, 0, 1] +
            in_field[0, 0, 0] +
            in_field[0, 0, -1]) / 3
```

3.2. Performances

For each data point of the results presented below, we do 10 runs and take the median among the 10 results. Before each measurements, we do 10 iterations to warm up the cache, therefore the results are not made on cold cache.

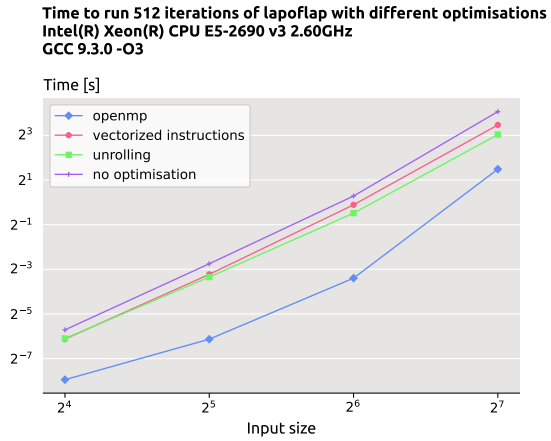

Fig. 5. Timing plot of the different optimization steps. The x-axis only shows the width of the space. The total space size is obtained by doing $width \times length \times height$, where $length = width$ and $height = width/2$

.

3.2.1. Optimizations

Three different optimizations are applied the generated c++ code : Unrolling the loops, using vectorized instructions and shared memory parallelism using OpenMP. We add them to our code generation in the order of the last sentence (especially because vectorized instructions need unrolling).
The figure 5 shows the time it takes to complete the computation of the basic stencil code listed in listing 1 for 512 iterations and different input sizes. The total input space size for each computation is $width \times length \times height$, where *width* and *length* are equal to the value given on the x-axis of the plot 5 and *height* is equal to *width*/2. The input data is the same as the one shown by figure 1. Finally, the processor used for the computation has 12 physical cores, we used all of them, without hyper threading, for the shared memory parallelism.
About the results, we have that the code with OpenMP (as well as all other optimizations) is indeed the fastest by a factor of x6 for the largest input size in comparison with the code without optimization. However, applying the vectorized instructions by our self seems to slow down the computation. It is most likely that by doing it by hand, we prevent GCC to make some optimizations.

3.2.2. Numpy vs Generated code

We compare three different computations generated with our DSL using all the optimizations with an equivalent Numpy computation. The computations are the following: *lapoflap* is the double application of the laplace operator as in listing 1, *vertical blur* is as described above and in listing 2, *copy*

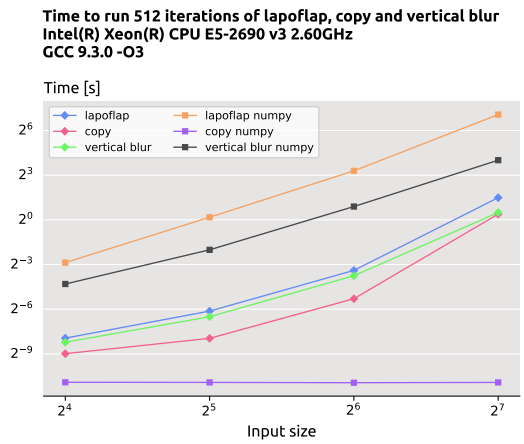

Fig. 6. Timing plot for comparing by DSL generated code and simple numpy code. The meaning of the x-axis is similar as in figure 5.

is a simple copy of the input field in the output field.The setup is the same as the one described in the previous section 3.2.1.
The result is shown in figure 6. We observe that we have a significant speed up between the *lapoflap* and *vertical blur* in comparison with their Numpy implementation. For the largest input size, the speedup is respectively x47 and x11. On the other hand, the *copy* version of Numpy is much faster. It has a constant runtime for all input size. The reason is probably that Numpy is not doing any computation to copy the array into the other and is simply referencing the output array as the input one.

4. CONCLUSIONS

Domain Specific Language is a practical solution to quickly write efficient code without requiring any knowledge in code optimization. We saw that we were able to outperform the performances given by a naive implementation in C++, as well as a Numpy implementation.
Even if the vectorized instructions optimization should probably be taken away by default when generating the code, it is still interesting to have the possibility to add them without the compiler intervention.
Further work on additional optimizations as well as to give the possibility to the user to add extra annotations to enable more complex optimizations would be interesting to conduct.

5. REFERENCES

[1] Tobias Wicky, "Original toydsl repository," https://github.com/twicki/toyDSL, Accessed: 2021-08-24.

[2] Python Software Foundation, "ast library repository," https://github.com/python/cpython/blob/3.9/Lib/ast.py, Accessed: 2021-08-24.

[3] Intel Corporation, "Intel intrinsics guide," https://software.intel.com/sites/landingpage/IntrinsicsGuide/, Accessed: 2021-08-28.