Improving the performance of a domain-specific language compiler

Semester Project

Langwen Huang[1], Supervisor: Eddie Davis[2], and Oliver Fuhrer[2]

[1]Department of Mathematics, ETH Zurich
[2]Vulcan Inc.

January 15, 2021

Abstract

As many supercomputers evolve into hybrid node designs with CPUs and GPUs, it become quite hard to make existing high performance computing applications run efficiently on them. GT4Py is a domain-specific language(DSL) compiler written in Python that helps porting high performance computing applications to those platforms by compiling stencil computations into executable codes targeted for various devices including GPUs. It has a prototype `cuda` backend written in pure Python that provides more flexibility compared to the existing `gtcuda` backend. While the former has not been as heavily optimized, the latter suffers from long compile times other inherent limits of the templated C++ code being generated. This project implemented a flexible optimization pass for the `cuda` backend containing various optimization techniques including loop reordering, K-loop unrolling, K-caching, prefetching, read-only caching and blocksize adjusting for the `cuda` backend. We benchmark the two backends using a set of representative stencil computations in GT4Py. All the techniques show positive effect on the Thomas solver stencil. All of them combined can improve 7% computation throughput and 12% memory throughput for the stencil. Meanwhile, the performance of other stencils varies according to combination of techniques. One of the best combinations reaches 3% improvement of performance for the `Riem_Solver3` stencil and 8% for the Thomas solver stencil. The performance of optimized `cuda` backend on Thomas solver is on par with that of the `gtcuda` backend. With more optimization techniques added, the `cuda` backend is hoping to close the performance gaps between the two backends which holds the promise of combining the best of two worlds with the `cuda` backend potentially replacing the other in the future.

Contents

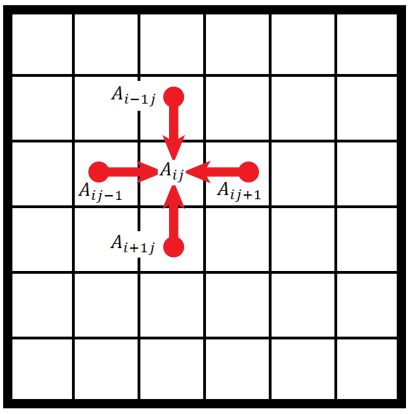
1 Introduction

1.1 Computational challenges for weather and climate modeling

Since the invention of digital computers, weather and climate prediction using numerical models has always been a computational challenge. To achieve more accurate weather and climate predictions, there are continuous efforts of adapting numerical models to the world's largest supercomputers. In recent years, Moore's law for exponential growth of CPU performance has come to an end (Theis & Philip Wong 2017). This has led many current and emerging supercomputers to hybrid node designs where most of the computational performance is delivered from some form of accelerators such as graphics processing units (GPUs). However, this raises the problem of adapting numerical weather and climate models to perform well on such platforms. GPUs require the programmer to expose massive concurrency in what is called the single-instruction-multiple-thread (SIMT) programming paradigm in order to run efficiently on GPUs. Compilers fail at automatically transforming the existing Fortran source code of numerical models optimized for traditional multi-core CPUs into executabels that run efficently on GPUs using the SIMT paradigm.

Yashiro et al. (2016) made the first attempt to adapt the NICAM numerical weather model to GPUs on the TSUNAMI 2.5 supercomputer using OpenACC compiler directives, followed by Fu et al. (2017) running multiple models on the TaihuLight supercomputer with a similar OpenACC approach and Bertagna et al. (2020) running non-hydrostatic HOMME model on the Summit supercomputer using Kokkos. OpenACC can be used to port existing code to GPU by annotating parallelizable loops in the code while Kokkos - a templated C++ library - allows writing parallel code targeting multiple accelerators including GPUs. While Kokkos already provides some abstractions for writing performance protable code, users of OpenACC have to think of low-level details to make the code efficient. Fuhrer et al. (2018) made an attempt to increase the level of abstraction rewriting parts of the COSMO model used in operations at several national weather services. Their work represents a step towards further decoupling high level numerical code and low level code for the dynamical core using the GridTools C++ library. The LFric model from UK MetOffice takes a similar approach using PSyclone (Adams et al. 2019). Both GridTools and PSyclone are domain-specific languages (DSLs). Model developers can write in or interpret existing code into such DSLs while performance engineers or computational scientists optimize DSL compilers for a given set of hardware targets. The two combined can make an efficient model running on any supercomputer platforms.

1.2 GT4Py - a DSL for stencil computation



$$A_{ij} = \frac{1}{4}\left(A_{ij-1} + A_{i-1j} + A_{ij+1} + A_{i+1j}\right), \forall i, j$$

Figure 1: Diagram of a stencil for Laplacian operator on a 2D grid

Following GridTools C++ and PSyclone's approach, GT4Py is a Python library for generating high performance implementations of stencil kernels from a high-level definition using regular Python functions. GT4Py exposes a DSL embedded in the Python language named GTScript. The high-level specification

of stencil kernels in Python is transpiled into low level C++/CUDA code that can execute efficiently on multiple hardware targets including CPU and GPU.

A stencil computation refers to a computational pattern that occurs frequently in codes that solve partial differential equations on a computational grid. Discretized numerical operators translate into updating all the values of a field defined on a grid according to the same pattern — called a 'stencil' which is a combination of values at neighboring gridpoints on the grid. For example, a Laplacian operator applied to a field defined on an infinitely large two-dimensional regular grid can be expressed as a stencil computation, where every value is updated as the average of its upper, lower, left and right neighbors (Figure 2, assuming grid spacing equal to one in both dimensions). Most numerical weather and climate models can be expressed as an sequence of stencil computations, with intermediate results stored in temporary arrays before they are being further processed by the next stencil.

1.3 GT4Py internals

Users of GT4Py write stencil definitions (Figure 2 left) with the GTScript DSL in Python. A stencil definition contains a hierarchy of elements: a function definition decorated by `@gtscript.stencil()` which takes in 3 dimensional fields as arguments, multiple `with computation(DIRECTION)` blocks where `DIRECTION` can be `FORWARD`, `BACKWARD` and `PARALLEL` specifying computation direction in the third dimension ("K" dimension), and multiple `with interval(BEGIN,END)` blocks with independent iteration region of K dimension. Inside each `with interval` block, there are expressions defining stencil computations in the iteration region by updating input fields. Fields are referenced with 3-integer-tuples representing relative position of the element and fields referenced with `0,0,0` can be abbreviated to just field names. GT4Py will generate code that apply those expressions to each position in the iteration region where relative field references are interpreted as absolute ones according to that position.

GT4Py follows a standard compilation procedure to transform stencil definitions into executable code: firstly, it parses the stencil definition into a Python AST (Abstract Syntax Tree), then it transforms the Python AST into a internal Definition IR (Intermediate Representation) tree, followed by a series of analysis and optimization passes to optimize the underlying computation represented by the Definition IR, and then emits the Implementation IR. The Implementation IR maps the hierarchy of elements in stencil definitions into a tree of IR nodes: StencilImplementation node corresponds to function definition, Multistage and Stage nodes correspond to `with computation` blocks, Applyblock nodes correspond to `with interval` blocks, and low level nodes correspond to low level elements in GTScript like operators, numbers, and assignments. GT4Py backends consume the Implementation IR to generate device-specific code.

There are two backends targeted for GPUs: the `gtcuda` backend generates GridTools C++ code which expands to CUDA code according to C++ template rules, the other `cuda` backend is an experimental backend that directly generates CUDA code. The `gtcuda` backend provides best performance on platforms with GPUs as GridTools adds optimization techniques during template expansion. However, relying heavily on C++ templates makes it time and memory consuming during compilation, and is hard to maintain. The `cuda` backend frees GT4Py developers from maintaining both the Python code to generate GridTools code and the obscure C++ template code in GridTools. It is written in pure Python and directly generates human-readable CUDA code making it much faster and less memory demanding when compiling stencils as compared to the `gtcuda` backend. Currently, it is a direct translation of Implementation IR (Figure 2 right), and as a result, the code it generates is less efficient than the `gtcuda` one.

1.4 Motivation

Currently, the dynamical core of the numerical weather and climate model FV3 is being ported to GT4Py by Vulcan Inc. and has been successfully validated to produce the same results as the original Fortran version. When running FV3 using GT4Py on platforms with GPUs, the `cuda` backend is not as performant as the `gtcuda` backend. However, since the `gtcuda` backend eventually generates CUDA code as well, there is potential to improve the performance of the `cuda` backend. This project aims to

Stencil Definition

```
@gtscript.stencil()
def thomas_solver_forward(
  a: FIELD_FLOAT,
  b: FIELD_FLOAT,
  c: FIELD_FLOAT,
  d: FIELD_FLOAT,
  x: FIELD_FLOAT):
"""
[b0 c0            ] [x0]   [d0]
[a1 b1 c1         ] [x1]   [d1]
[   a2 b2 c2      ] [x2] = [d2]
[      ...        ] [..]   [..]
[         ... cn-1] [..]   [..]
[            an bn] [xn]   [dn]
"""

  with computation(FORWARD):
    with interval(1, None):
      w = a/b[0, 0, -1]
      b = b - w*c[0, 0, -1]
      d = d - w*d[0, 0, -1]
```

generated CUDA code

```
__global__ void multi_stage__31_kernel(
  /* Definition of arguments*/) {
  /* Definition of local variables */
  // stage__16
  for (k=k_min+1+k_block; k<=k_max; k+=k_inc) {
    for (i=i_min+i_block; i<=i_max; i+=i_inc) {
      for (j=j_min+j_block; j<=j_max; j+=j_inc) {
        idx_data_ijk=i*data_strides[0]+
                     j*data_strides[1]+
                     k*data_strides[2];
        idx_data_ijkm1=i*data_strides[0]+
                       j*data_strides[1]+
                       (k-1)*data_strides[2];
        float64_t w;
        w=a[idx_data_ijk]/b[idx_data_ijkm1];
        b[idx_data_ijk]=b[idx_data_ijk]-(w*c[idx_data_ijkm1]);
        d[idx_data_ijk]=d[idx_data_ijk]-(w*d[idx_data_ijkm1]);
      }
    }
  }
}
```

Figure 2: Stencil definition in GTScript (left) and the corresponding CUDA code generated by the `cuda` backend

design, implement and test different optimization strategies for the `cuda` backend in order to address the aforementioned performance gap.

2 Method

2.1 Benchmark set

It is important to quantify the performance of a specific backend before optimizing it. A benchmark set is made choosing some of the more time consuming stencils from the FV3 project including: `fillz`, `sim1_solver` and `saturation_adjustment`. Note that `sim1_solver` is called from two Riemann solver stencils `riem_solver_c` and `riem_solver3`, we will only benchmark the two instead of `sim1_solver` as input data for it is not available. In addition, a Thomas solver (tridiagonal solver) stencil is added to the benchmark set, because it is very common in many numerical weather and climate models.

For input data of FV3 stencils, we use the validation dataset shipped with the FV3-GT4Py project which has appropriate input data for each stencils. We use random data for Thomas solver stencil, and the running time should be the same for any given data since it is a non-oblivious algorithm.

We measure the time of each stencil in the benchmark set running 100 times with a specific GT4Py backend ignoring the time of reading input data. Then the measured times are treated as performance index for the backend to guide adaptation of optimization techniques. All the timings are performed in the `n1-standard-8` node from Google Cloud with a P100 GPU and 30GB memory, because the ported FV3 usually runs on the supercomputer Piz Daint which has P100 GPU nodes.

2.2 Optimization techniques

A preliminary profiling to the benchmark set shows that most of the stencils are memory-bound which means we need to find optimization techniques that reduces time of memory operations. Apart from

improving specific algorithms which is not possible from a backend's perspective, the general way to reduce memory operations is to utilise the cache hierarchy. The mordern GPU architecture maintains a cache hierarchy like CPUs where a cascade of storage devices called caches ranging from fast but low-capacity registers to slow but high-capacity last level cache. Frequent access to the same memory location or continuous access to memory enables GPUs to cache those memory locations. The more frequent one location is accessed, the longer it can reside in faster higher level cache. In the definition of stencils, memory accesses only have partial-order meaning that any order satisfying the partial order will produce the same intended result, but the performance of the stencil with different memory access orders may differ because the cache is utilised differently. So, a backend can improve the performance of a memory bound stencil by rearranging memory accesses in a smart way to make efficient use of the cache hierarchy.

Specifically, because the GT4Py is made for 3D stencil computations which allows specifying computation order for the third dimension ("K" dimension or height), we explored possibilities of optimizing memory accesses for the K dimension including loop reordering, K-loop unrolling, K-caching. We also exploited the staticness of stencil computing in the sense of memory access pattern is determined at compile time by using prefetching and read-only caching. In addition, we added an option to vary blocksize configurations to adapt to specific GPU platforms.

Before implementing optimization techniques into the backend, we made a small CUDA program based on the generated CUDA code of Thomas solver stencil and handcrafted those techniques in the program as a proof of concept. It is also to make sure those techniques are effective or at least not harmful.

2.2.1 Loop reordering

The memory layout of multidimensional array in GT4Py is row-major where `x[i,j,k]` and `x[i+1,j,k]` are continuous in memory. The `cuda` generate stencils into KIJ loops ("I" "J" as first and second dimension) to ensure the correct result when the loop body have offset memory accesses in both K and IJ directions. But KIJ loops are inefficient for IJ-independent loops where there's only offsets in K direction compared with IJK loops because IJK loops have better time-locality for memory accesses. For example, `x[i, j, k] = x[i, j, k-1];` requires 1 memory read and 1 memory write in KIJ loops but may only requires 1 cache read and 1 memory write in IJK loops if the cache size is big enough so that `x[i, j, k]` is not flushed out of the cache. Therefore, to exploit such time-locality, one would have to identify whether a loop is IJ-independent loop and then swap the loop order.

Loop reordering

```
/* Before: */
for (k=k_min+1+k_block; k<=k_max; k+=k_inc)
  for (i=i_min+i_block; i<=i_max; i+=i_inc)
    for (j=j_min+j_block; j<=j_max; j+=j_inc)
      x[i, j, k] = x[i, j, k-1];
/* After loop reordering: */
for (i=i_min+i_block; i<=i_max; i+=i_inc)
  for (j=j_min+j_block; j<=j_max; j+=j_inc)
    for (k=k_min+1+k_block; k<=k_max; k+=k_inc)
      x[i, j, k] = x[i, j, k-1];
```

2.2.2 K-loop unrolling

Inspired by GridTools, one can unroll the last level K-loop based on IJK loop optimization to enable the CUDA compiler optimizing redundant memory accesses into register accesses and to give the compiler more flexibility of reordering statements in loop body to hide memory access latency. This technique make use of CUDA compiler using `#pragma unroll N` where N is determined in the optimization pass according to number of instructions in the loop body. Currently, N is determined such that the number

of instructions in unrolled loop body is the largest multiple of the original number that is smaller or equal to 8.

K-loop unrolling

```
for (i=i_min+i_block; i<=i_max; i+=i_inc)
  for (j=j_min+j_block; j<=j_max; j+=j_inc)
#pragma unroll N // <----K-loop unrolling
    for (k=k_min+1+k_block; k<=k_max; k+=k_inc)
      x[i, j, k] = x[i, j, k-1];
```

2.2.3 K-caching

Also inspired by GridTools and the Dawn compiler (Osuna et al. 2020), K-caching manually inserts cache arrays for memory accesses in along K direction. Since offsets in the K direction are usually ± 1, K cache of size 2 is sufficient for most case. The K-caching technique replaces memory accesses by cache accesses, then inserts cache read at the beginning of the loop body and write-back-to-memory statements at the end if the cache is modified. Also at the end of the loop, the second element of the cache is moved to the first to match the memory position in the next iteration.

K-caching

```
/* Before: */
for (i=i_min+i_block; i<=i_max; i+=i_inc)
  for (j=j_min+j_block; j<=j_max; j+=j_inc)
    for (k=k_min+1+k_block; k<=k_max; k+=k_inc)
      x[i, j, k] = x[i, j, k-1];
/* After K-caching: */
float64_t x_cache[2];
for (i=i_min+i_block; i<=i_max; i+=i_inc)
  for (j=j_min+j_block; j<=j_max; j+=j_inc) {
    x_cache[0] = x[i, j, k_min+k_block];
    for (k=k_min+1+k_block; k<=k_max; k+=k_inc) {
      x_cache[1] = x[i, j, k];
      x_cache[1] = x_cache[0];
      x[i, j, k] = x_cache[1];
      x_cache[0] = x_cache[1];
    }
  }
```

2.2.4 Prefetching

Prefetching loads a memory section in to the specific cache before it is used to reduce cache miss which is usually the case for GT4Py stencils. It can be applied to any programs as long as the memory accesses can be predetermined at compile time which is the case for GT4Py. Note that the CUDA language does not officially support prefetching, we use inlined PTX code to insert prefetch instructions. While there are prefetch instructions for both L1 and L2 cache, only L1 prefetching is used because it is more efficient than L2 prefetching when profiling the Thomas solver CUDA program.

Prefetching

```
/* Before: */
for (k=k_min+1+k_block; k<=k_max; k+=k_inc)
  for (i=i_min+i_block; i<=i_max; i+=i_inc)
    for (j=j_min+j_block; j<=j_max; j+=j_inc)
```

```
      x[i, j, k] = x[i, j, k-1];
/* After prefetching: */
for (k=k_min+1+k_block; k<=k_max; k+=k_inc)
  for (i=i_min+i_block; i<=i_max; i+=i_inc)
    for (j=j_min+j_block; j<=j_max; j+=j_inc)
      prefetch(&x[i, j, k]);
      prefetch(&x[i, j, k-1]);
      x[i, j, k] = x[i, j, k-1];
```

2.2.5 Read-only caching

Similar to prefetching, the read-only caching inserts "load from global memory (LDG)" instructions to memory reads if that memory location is not modified in the CUDA kernel. The LDG instruction persuades GPU to cache the read-only data into read-only cache which has much less latency than normal caches and in turn increase performance.

Read-only caching

```
/* Before: */
for (k=k_min+1+k_block; k<=k_max; k+=k_inc)
  for (i=i_min+i_block; i<=i_max; i+=i_inc)
    for (j=j_min+j_block; j<=j_max; j+=j_inc)
      x[i, j, k] = a[i, j, k-1]; // <---- a is read-only
/* After read-only caching: */
for (k=k_min+1+k_block; k<=k_max; k+=k_inc)
  for (i=i_min+i_block; i<=i_max; i+=i_inc)
    for (j=j_min+j_block; j<=j_max; j+=j_inc)
      x[i, j, k] = __ldg(&a[i, j, k-1]);
```

2.2.6 Blocksize adjusting

For each memory accesses in 3D loops of `cuda` generated codes, the GPU accesses a block of data with a predetermined blocksize. The K dimension of the blocksize is always 1 to ensure the correctness of stencil computation with specific K-order requirement, while the blocksizes of IJ dimension can be arbitrary. The `cuda` backend currently set a constant blocksize of (32, 8, 1), but this may not be the optimal choice as GridTools has default blocksize of (64, 8, 1) or (256, 8, 1) (new version). So we varied the I dimension of the blocksize to find if there is any better choice. Moreover, since GPU read continuous data faster, it makes sense to set the J dimension of the blocksize to one if the stencil only contains IJ-independent loops.

2.3 Implementation of an optimization pass for `cuda` backend

As an attempt to decouple optimization from backend implementation, we added a new optimization pass between Implementation IR and the `cuda` backend. The pass follows the visitor pattern, which takes in the root node of the Implementation IR called StencilImplementation node and visit its children node using depth-first search. When it finds nodes of type ApplyBlock representing 3D loops in IR it passes the node to each optimization methods which themselves follow visitor pattern. Each optimization methods either modifies children nodes under the ApplyBlock node mimicking manual insertion/modification of expressions in the loop body, or add metadata to the ApplyBlock node which is identified in the backend to generate code of specific pattern.

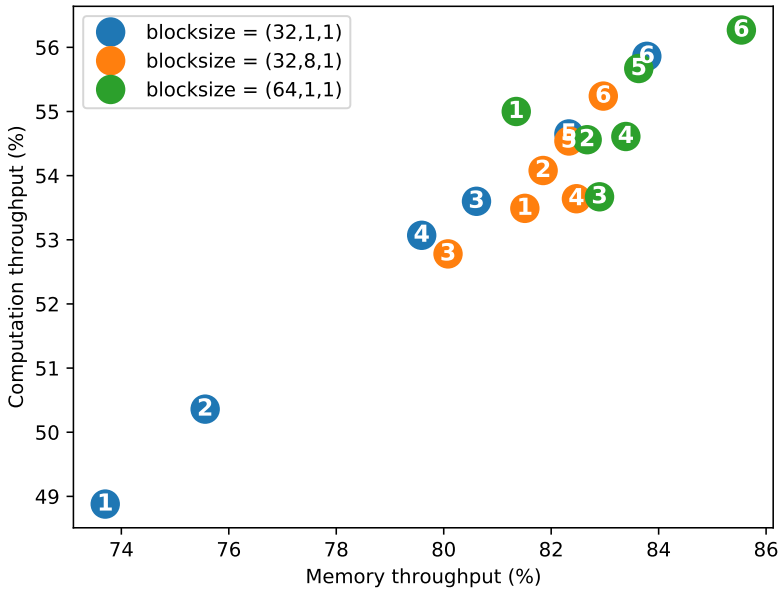
The structure of the optimization pass makes it easy to switch on/off any specific optimization technique,

so we made a grid search script that benchmarks every combinations of existing techniques to examine the effectiveness of those techniques.

3 Result

3.1 Profiling result of Thomas solver program

We profiled the handcrafted Thomas solver CUDA program using Nsight Compute to evaluate the effectiveness of optimization techniques before introducing into the `cuda` backend. The optimization techniques mentioned in the last section are added one by one to show the effect of the techniques given previous techniques presented, because it is hard to implement every combinations of those techniques. We also varied the blocksize with 3 different configurations for each set of techniques to evaluate the blocksize adjusting technique. Ideally, the profiling would happen on the P100 platform where the benchmark is performed, but unfortunately, as Nsight Compute is not compatible with P100, we did profiling on a PC with an NVIDIA RTX2060 GPU.



Number	IJK loop	Prefetching	Read-only caching	K-Caching	K-loop unrolling
1					
2	✓				
3	✓	✓			
4	✓	✓	✓		
5	✓	✓	✓	✓	
6	✓	✓	✓	✓	✓

Figure 3: Profiling result for handcrafted Thomas solver

As Figure 3 shows, enabling every techniques always results in best memory and computation throughput, but the effect of different techniques differs with different blocksizes. With blocksize (32, 1, 1), all techniques except read-only caching show positive effect. Meanwhile, with blocksize (32, 8, 1) and (64, 1, 1) only K-Caching and K-loop unrolling remains positive for both computation and memory throughput, but unoptimized stencil tend to have better performance compared with those in blocksize (32, 1, 1).

3.2 Grid search result

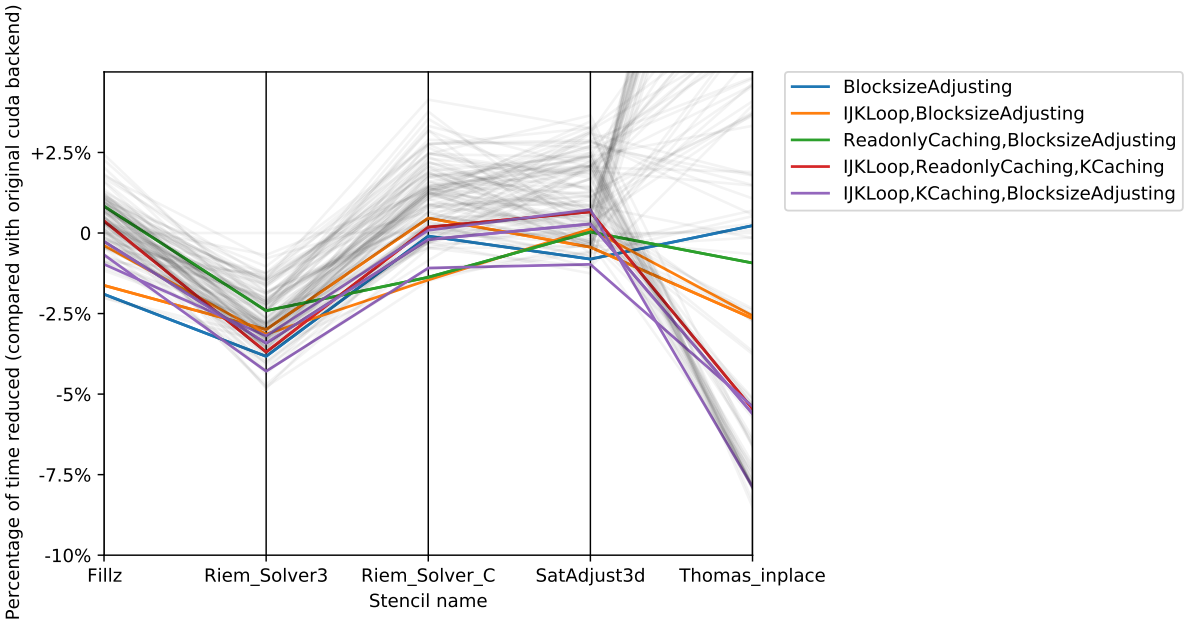


Figure 4: Relative time difference of median running time, highlighting elements on Pareto front with worst performance under +1%

Although enabling every techniques gives the best result when profiling the Thomas solver stencil, it is still possible that other combinations except those in Figure 3 may raise better result, and other stencils may react differently as well. Therefore, to make a comprehensive analysis for the performance of the improved `cuda` backend, we made a grid search exploring every combinations of techniques and varying blocksizes for each combination. For each configuration with a certain combination of techniques and blocksize, we run the benchmark set for 100 times and take the median of measured times to reduce effect of outliers.

The result of grid search is shown in Figure 4 as a parallel plot of relative time differences of median running times of each configurations and that of original `cuda` backend, where we highlight configurations on Pareto front (no other configurations are faster for all the stencils) with worst performance under +1%. The combination of loop reordering, K-caching and blocksize adjusting reaches 3% reduction of time for `Riem_Solver3` and 7% on Thomas solver while its improvement for `Fillz`, `Riem_Solver_C` and `SatAdjust3d` is marginal. Other combinations show similar result. On the histogram plot (Figure 5), the running time of optimized `cuda` backend is compared with those of original `cuda` and `gtcuda` backends. The histogram plot shows similar result as the parallel plot of relative time difference, and `gtcuda` is still faster than the `cuda` except in Thomas solver stencil.

4 Conclusion

We presented improvements to the `cuda` backend as an addition optimization pass in the backend. The optimization pass implemented loop reordering, K-loop unrolling, K-caching, prefetching, read-only caching and blocksize adjusting. With grid search of every combinations of the optimization techniques, we found the combination of loop reordering, K-caching and blocksize adjusting can improve 8% performance for the Thomas solver stencil reaching the level of the `gtcuda` backend and 3% for the `Riem_Solve3` stencil.

As the performance improvement on `Fillz`, `Riem_Solver_C` and `SatAdjust3d` is not significant, we will continue to add more optimization techniques especially from `gtcuda` as well as tuning existing ones to adjust to those stencils.

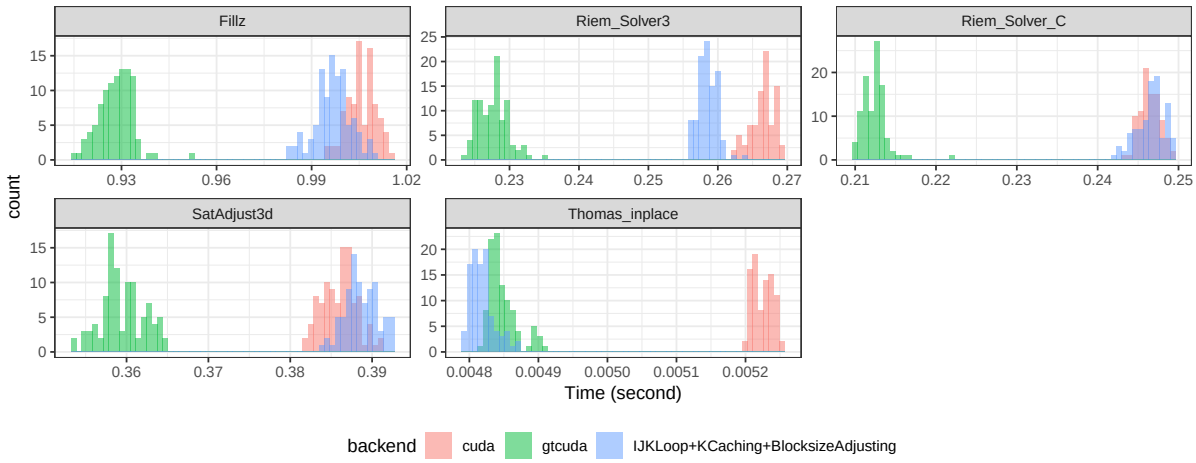


Figure 5: Histogram of running times for different configurations

Acknowledgement

The author would like to thank his supervisor Eddie Davis and Oliver Fuhrer and other members in Vulcan Inc. for their kind guidance and inspiring ideas.

References

Adams, S. V., Ford, R. W., Hambley, M., Hobson, J. M., Kavčič, I., Maynard, C. M., Melvin, T., Müller, E. H., Mullerworth, S., Porter, A. R., Rezny, M., Shipway, B. J. & Wong, R. (2019), 'LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models', *J. Parallel Distrib. Comput.* **132**, 383–396.

Bertagna, L., Guba, O., Taylor, M. A., Foucar, J. G., Larkin, J., Bradley, A. M., Rajamanickam, S. & Salinger, A. G. (2020), A Performance-Portable Nonhydrostatic Atmospheric Dycore for the Energy Exascale Earth System Model Running at Cloud-Resolving Resolutions, *in* 'Proc. Int. Conf. High Perform. Comput. Networking, Storage Anal.', SC '20, IEEE Press.

Fu, H., Liao, J., Ding, N., Duan, X., Gan, L., Liang, Y., Wang, X., Yang, J., Zheng, Y., Liu, W., Wang, L. & Yang, G. (2017), Redesigning CAM-SE for peta-scale climate modeling performance and ultra-high resolution on sunway taihulight, *in* 'Proc. Int. Conf. High Perform. Comput. Networking, Storage Anal. SC 2017', Association for Computing Machinery, Inc.

Fuhrer, O., Chadha, T., Hoefler, T., Kwasniewski, G., Lapillonne, X., Leutwyler, D., Lüthi, D., Osuna, C., Schär, C., Schulthess, T. C. & Vogt, H. (2018), 'Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0', *ETH Libr. Geosci. Model Dev* **11**, 1665–1681.
URL: *http://doi.org/10.5194/gmd-11-1665-2018*

Osuna, C., Wicky, T., Thuering, F., Hoefler, T. & Fuhrer, O. (2020), 'Dawn: a high-level domain-specific language compiler toolchain for weather and climate applications', *Supercomputing Frontiers and Innovations* **7**(2), 79–97.

Theis, T. N. & Philip Wong, H. S. (2017), 'The End of Moore's Law: A New Beginning for Information Technology', *Comput. Sci. Eng.* **19**(2), 41–50.
URL: *https://purl.stanford.edu/gc095kp2609.*

Yashiro, H., Terai, M., Yoshida, R., Iga, S. I., Minami, K. & Tomita, H. (2016), Performance analysis and optimization of nonhydrostatic icosahedral atmospheric model (NICAM) on the K computer and TSUBAME2.5, *in* 'PASC 2016 - Proc. Platf. Adv. Sci. Comput. Conf.', Association for Computing Machinery, Inc.