Using CUDA Graphs and Custom CUDA Kernels to Speed up CuPy

HPC for Weather and Climate Report

Eliott Bezençon, Julia Mietz, Nicolas Vetsch, Alexander Maeder

August 2024

Contents

1 Introduction

While GPUs are increasingly used for high-performance computing tasks in weather and climate modeling [1], there are some hardware-specific obstacles to achieving the best possible performance. For instance, the launch of operations on a GPU always entails a submission overhead. Especially when accumulated over many small kernel launches, this submission overhead can lead to diminishing returns with regards to speedup [2]. To address this issue, CUDA graphs allow for capturing a sequence of GPU operations (such as kernel launches or memory copies) and then replaying them in a single CPU operation (Figure 1). Once captured, they can be replayed as needed.

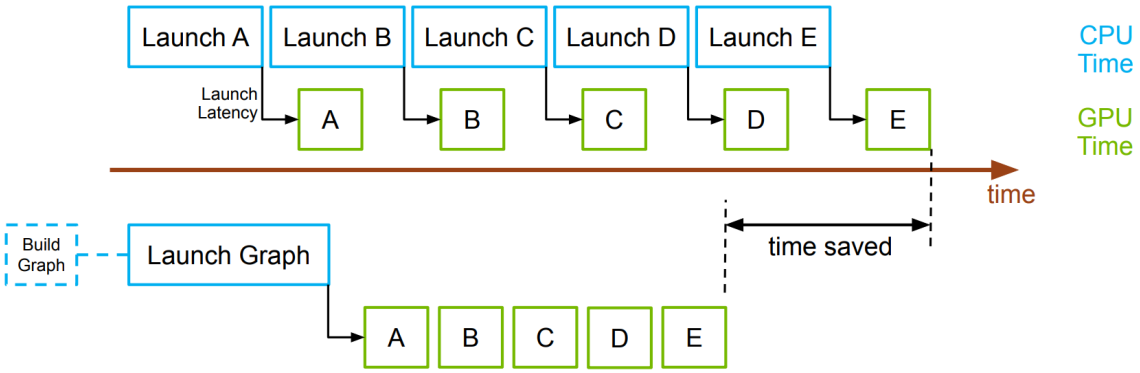


Figure 1: Schematic showing a default stream launch (top) and the modified graph launch using a CUDA graph (bottom) [3].

In previous work, Demirsü and Lervik [4] compared CUDA graphs for breadth-first search. The results showed varying performance improvements depending on the number of nodes in the breadth-first search. For small workloads, the speedup was negative, showing that the graphs can be detrimental when the kernel work is small. An increase of performance of 14% was measured for medium workloads and no performance difference was shown for large workloads. Hence, this work shows that CUDA graph capture can improve performance for certain applications.

A frequent computational problem in the context of weather and climate models is repeated launches of comparatively small kernels when doing iterative stencil computations. It is a common practice to apply a numerical diffusion filter to the output of a model to reduce the impact of noise, e.g. originating from numerical dispersion or nonlinear instability of discontinuous physical processes [5]. To compute this diffusion term, the same operation is conducted repeatedly for every time-step of the calculation. Thus, this operation is a good candidate to obtain a speedup by reducing the submission overhead using CUDA's graph capture API [6].

The experimental set-up for this work firstly entails the comparison of a 2D stencil code written in Python employing only basic CuPy [7] functionality with a base CUDA graph capture version. Furthermore, we will compare it with a custom kernel implementation. Finally, both graph and custom kernel versions are fused to investigate combined effects. Based on the aforementioned previous research by Demirsü and Lervik [4], it is expected that the results will vary with the size of the workload. Hence, all versions will be tested on differently-sized data. The effect of changing the number of iterations will also be analyzed since the approach of replaying operations is expected to be more effective for a higher number of iterations.

2 Methods

In this section, we will discuss the algorithmic motif of stencil computation and present a simple 2D stencil program written in Python using the CuPy framework. This will form the baseline for the subsequent benchmarks. Furthermore, we will introduce CUDA graphs and show how these can be instantiated in Python via stream capture. It is then discussed how this technique was applied to the 2D stencil program.

In the third part, we will show how CuPy's built-in functionality can be extended by creating custom, user-defined CUDA kernels. As opposed to the CUDA graph instantiation via stream capture, which is provided by CuPy, the creation of user-defined kernels requires including a C++ CUDA kernel in the source file. Multiple custom kernels are discussed, showcasing different approaches towards optimizing the stencil code.

2.1 Stencil Computations

Stencil computations are algorithms on regular grids, where each output value at a certain grid point only depends on the input values in a compact neighborhood of the grid point. The output value for every grid point is computed from its respective neighboring input points in the exact same way, hence the name "stencil". As a simple yet representative example, we use a fourth-order non-monotonic diffusion filter, defined by

$$\frac{\partial \phi}{\partial t} = -\alpha_4 \Delta_h(\Delta_h \phi). \tag{2.1}$$

Here, ϕ is an arbitrary spatio-temporal variable and α_4 is the diffusion coefficient. The h subscript denotes that the Laplacians only act on a horizontal xy-plane. Equation 2.1 is a simplified version of the filter described by Xue [5], where we choose $n = 4$ and $S = 0$, i.e. the source term is dropped. We also do not consider a flux-limiting scheme.

For the domain discretization, a uniform grid is used, formalized as

$$\phi^n_{i,j,k} = \phi(i\Delta x, j\Delta y, k\Delta z, n\Delta t), \tag{2.2}$$

where Δx, Δy, Δz, and Δt are the respective discretization steps in space and time. The horizontal Laplacians on this grid are discretized by employing a straight-forward second-order central-difference approach

$$\Delta_h \phi \approx \frac{1}{\Delta x \Delta y}\left(-4\phi^n_{i,j} + \phi^n_{i-1,j} + \phi^n_{i+1,j} + \phi^n_{i,j-1} + \phi^n_{i,j+1}\right), \tag{2.3}$$

where the k-index is dropped, as the vertical component remains unaffected. For the discretization of the time-derivative, a forward Euler scheme is used:

$$\frac{\partial \phi}{\partial t} \approx \frac{1}{\Delta t}\left(\phi^{n+1}_{i,j} - \phi^n_{i,j}\right). \tag{2.4}$$

After some rearranging, it can be seen that the computation of $\phi^{n+1}_{i,j}$ in each xy-plane follows the stencil motif. However, for computing the field values at the boundary grid points, the stencil will extend beyond the domain boundaries. This is mitigated by introducing a set of halo grid points surrounding our domain. These are updated by enforcing periodic boundary conditions.

The procedure of applying this discretized diffusion filter to some input field is expressed as pseudo-code in Algorithm 1. This algorithm will serve as the baseline for comparisons with our optimization approaches. It is implemented in Python, using only basic CuPy functionality, i.e. `cupy.ndarray` indexing, slicing, and arithmetic operations.

Algorithm 1: Discretized fourth-order non-monotonic diffusion

Input : 3D-Field in, number of iterations N, diffusion parameter $\alpha = \alpha_4 \frac{\Delta t}{\Delta x \Delta y}$
Output: 3D-Field out

1 **for** $i \leftarrow 1$ **to** $N - 1$ **do**
2 in $\leftarrow$ halo_update(in) ;
3 temp $\leftarrow$ laplacian(in) ;
4 out $\leftarrow$ laplacian(temp) ;
5 out $\leftarrow$ in $-\alpha$ out;
6 swap(in, out) ; // Pointer swap.
7 **end**
8 out $\leftarrow$ halo_update(out);

2.2 CUDA Graphs

In Algorithm 1, one can see that when applying the diffusion filter, we perform the exact same set of operations repeatedly $N-1$ times. As already described in Section 1, CUDA graphs might allow us to reduce launch latency in exactly such a scenario.

```
stream = cupy.cuda.Stream(non_blocking=True)
with stream:
    stream.begin_capture()
    # Capture a set of small CUDA kernel calls...
    graph = stream.end_capture()

# Repeatedly launch the captured graph.
for _ in range(num_iterations):
    graph.launch(stream)
```

Listing 1: Performing CUDA graph capture in CuPy.

In Listing 1, the graph capture and launch API as exposed by CuPy is demonstrated in a schematic way. In the most naïve graph capture approach towards optimizing our stencil diffusion program, we would simply capture all operations contained in lines 2-6 of Algorithm 1 in the graph, launch the graph $N - 1$ times and then perform the final halo update. However, it turns out that the pointer swap on line 6 becomes problematic in this simple approach: The captured graph will always replay the operations on the same memory addresses. Therefore, even if the pointers for the in- and out field are swapped before the end of the stream capture, the pointers will be reset for the next launch of the graph. This behavior is different from the original loop where the pointer indirection persists for the following iteration. The output of the algorithm will thus be incorrect when using this version.

To obtain a correct output, one possible solution would be to include an explicit memory copy between the two arrays in the graph. However, as should be intuitive, this is detrimental to the performance of the algorithm as it is innately slower than a pointer swap. We will show later that this leads to even worse performance than when not using CUDA graphs at all.

Hence, we consider two alternative strategies. Both are based on the idea of "unrolling" the loop in Algorithm 1. The first method is to capture two subsequent iterations in a single graph and launch it $\frac{N-1}{2}$ times. The second approach is to capture two graphs: One for even iterations and one for odd ones. In between the two captures, we swap the pointers. Depending on the

parity of the current iteration i, we then decide whether to launch one or the other. Thus, the two graphs alternately perform the same operations, but on data in different memory locations. The resulting implementations are summarized in Table 1.

Table 1: Implemented versions with graph capturing.

Version	Implementation description
CuPy baseline	Only basic CuPy functionality.
Single graph	A single graph, capturing a single iteration including an explicit memory copy.
Two graphs	Two graphs with swapped memory locations that are launched alternately.
Unrolled graph	A single graph, capturing two subsequent iterations.

2.3 Custom CUDA Kernels

In addition to the graph capturing, increasingly optimized code versions were implemented using custom CUDA kernels. The general idea is that, while CUDA graphs are a quick way to reduce kernel launch overhead by capturing and replaying code, manual optimizations allow to speed up and reduce the number of kernels.

```
modadd_kernel = cupy.RawKernel(r'''
extern "C" __global__
void my_modadd(const float* a, const float* b, float* c) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    c[tid] = a[tid] + 2*b[tid];
}
''', 'my_modadd')
# Instantiate three CuPy arrays a, b, c....
modadd_kernel((5,), (5,), (a, b, c))  # Grid size, block size and kernel arguments.
# Now array c is equal to a + 2*b.
```

Listing 2: Example of a simple user-defined modified addition kernel in CuPy. This example is adapted from the CuPy API documentation [8].

Custom kernels for this project were specified in C++ CUDA. The resulting kernels fuse multiple basic operations, as can be seen in Listing 2, which fuses addition and scalar multiplication. This reduces GPU global memory access and therefore improves performance in memory-bound applications like stencil computations. Additional parallelism can be achieved by computing independent universal functions concurrently.

An overview of the different custom kernel versions used is shown in Table 2. The implementation of most kernels is straightforward since grid points are independent, which means that race conditions are normally not an issue.

One exception is the 2D distributed Laplacian kernel using shared memory. The shared memory is a user-controlled L1 cache, shared among each thread block. With this implementation, each thread only needs to load its local elements from global memory during the stencil computation. The other elements of the stencil are loaded implicitly by neighboring threads, which means that overall fewer memory operations are performed. The general structure of such a kernel is shown in Algorithm 2. To the right of the algorithm, one can see the corresponding tiling of the

Table 2: Implemented version with custom CUDA kernels.

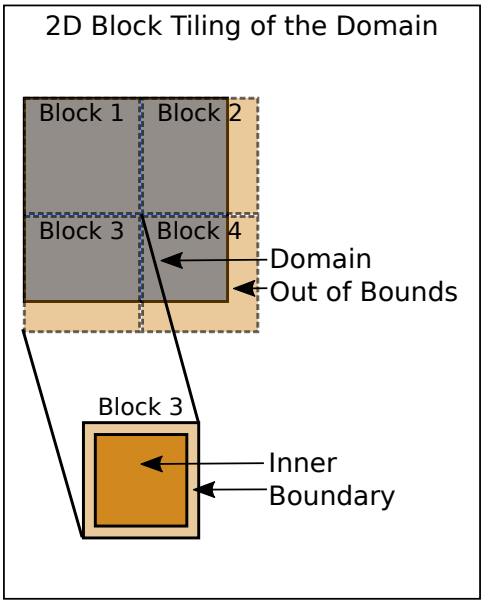
Version	Implementation description
Laplacian	Custom Laplacian kernel with either 1D or 2D thread block distribution.
Shared Memory	Custom Laplacian kernel with 2D distribution where threads load block local data into shared memory to decrease global memory access.
Halo Update	Custom Laplacian and Halo update kernel.
Field Update	Custom Laplacian, Halo update, and outfield update kernel.
Fused Laplacian	Single custom kernel for both Laplacians with either 1D or 2D thread distribution.
Graph	Fused Laplacian with graph capturing.

domain with thread blocks. The code has to differentiate between threads in the inner and the boundary of the thread block since boundary threads require data outside the local elements.

Algorithm 2: Shared memory Laplacian kernel

Input : 3D-Field in, thread indices tid_x, tid_y
Output: 3D-Field out

```
1  id_x, id_y ← get_array_indices(tid_x, tid_y) ;
2  if id_x, id_y ∈ Domain then
3  |  shm [id_x, id_y ] ← in [tid_x, tid_y ] ;
4  end
5  Synchronize threads ;
6  if id_x, id_y ∈ Domain then
7  |  if tid_x, tid_y ∈ Inner Thread Block then
8  |  |  out [tid_x, tid_y ] ← stencil(shm) ;
9  |  else if tid_x, tid_y ∈ Boundary of Thread Block
   |  |  then
10 |  |  out [tid_x, tid_y ] ← stencil(shm, in) ;
11 end
```



However, this branching is a major caveat of the implementation since it leads to so-called "warp divergence". A warp is a subgroup of threads in a block. Generally, all threads in a warp have to carry out the same instruction. Thus, a conditional statement in the code leads to the operations in both branches being serialized and threads having to stall which decreases overall performance [9].

To address this, an alternative shared memory Laplacian implementation could involve loading boundary elements into shared memory by fixed warps. This eliminates warp divergence but introduces work imbalance. Another option is a kernel where only inner threads perform work which would require a tiling of the domain with overlapping thread blocks.

3 Results

In this section, the various implementations of the diffusion filter which were described in the previous section are benchmarked. We compare the implementations' performance for varying numbers of grid points in the xy-plane, as well as for different numbers of iterations N. The benchmark is conducted on a single node of the Attelas cluster at the Institute of Integrated Systems at ETH Zürich. The node specifications relevant to our application are shown in Table 3. In the following, all reported data points are in fact averages over 32 individual measurements.

Cluster	Attelas
CPU	Intel(R) Xeon(R) CPU E5-2680 v4 (28 Core)
GPU	NVIDIA Tesla P100-PCIE-16GB (4.7 TFLOP/s)
Python Version	3.11.0
CuPy Version	13.2.0
CUDA Runtime Version	11.8
CUDA Driver Version	12.2

Table 3: Hardware and software setup used to benchmark our implementations.

3.1 CUDA Graphs

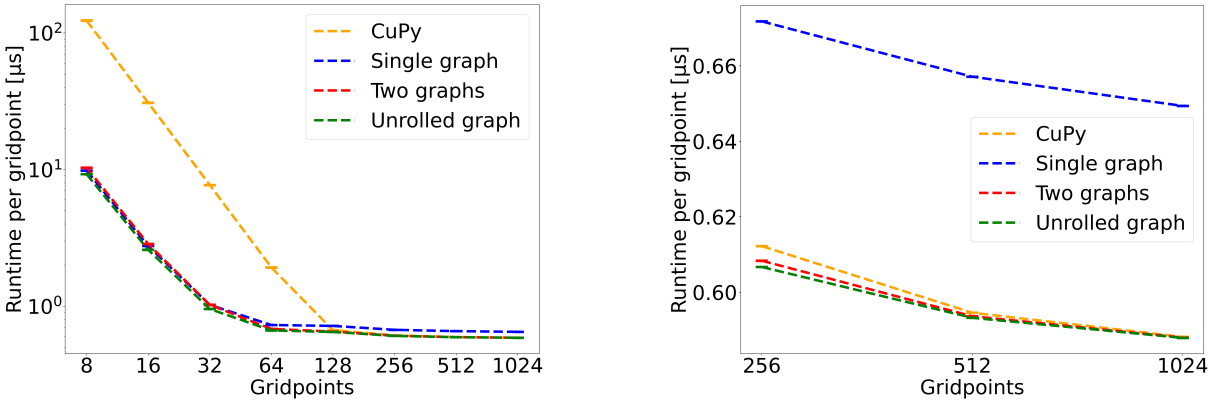


Figure 2: Left: Performance benchmarks of the graph capturing implementations. We compare the runtime per grid point ($\frac{time}{n_x n_y n_z}$) of the simple CuPy implementation baseline with the different graph capturing implementations for varying horizontal grid sizes (n_x, n_y). The number of vertical points and the number of iterations are fixed ($n_z = 64$, $N = 1025$). Right: Zoomed in at large grid sizes.

Figure 2 shows a comparison of the runtime per grid point for the implemented graph capture versions of the code as they were described in Section 2.2. One can refer to Table 1 for a short overview. For small horizontal grid sizes, a clear speedup can be observed for all CUDA graph capture implementations compared to the bare CuPy version. This can be ascribed to the fact that especially at small grid sizes, kernel launch latency could become a dominating factor. For increasing grid sizes, the runtime per grid point decreases rapidly and ultimately plateaus at a similar value for all implementations. Notice however that the naïve CUDA graph implementation, which includes an explicit memory copy, is quickly outperformed by even the bare CuPy version. We anticipated this behavior in Section 2.2 and will therefore disregard this implementation altogether in the following since there is no benefit to it.

For larger numbers of horizontal grid points, all implementations (except the "naïve" one) tend to converge towards a runtime per grid point of around 0.59 μs. This is expected as most of the runtime will be due to kernel execution, i.e. any potential overhead is hidden.

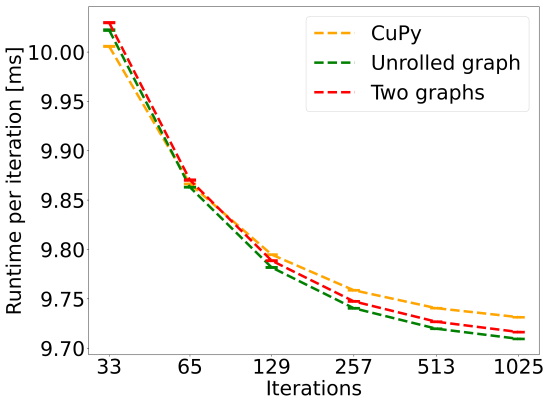


Figure 3: Comparison of the runtime per iteration ($\frac{time}{N}$) of the CuPy, two graphs, and unrolled graph implementations for different numbers of iterations with a fixed grid ($n_x = 128$, $n_y = 128$, $n_y = 64$)

As discussed in the previous sections, the expected advantage in employing graph capture is the reduction of launch latency when repeatedly performing the same set of computations. A key metric for that is the runtime per iteration. Figure 3 depicts the runtime per iteration of the different graph capture implementations compared to the bare CuPy version. We use a fixed grid size of $128 \times 128 \times 64$ and only vary the number of iterations. It can be observed that the runtime per iteration decreases in all cases. This is because we have the overhead of device synchronization and allocating memory, which happen only once. While the graph capture implementations perform worse than the bare CuPy implementation for small numbers of iterations N, they begin to break even around $N = 65$. For large values of N, graph capture consistently outperforms the bare CuPy implementation. A potential reason for the negative speedup for small N is the initial overhead of capturing a CUDA graph. This is in line with the fact that the implementation where we capture two graphs always performs slightly worse than the single unrolled graph version.

3.2 Custom CUDA Kernels

The left part of Figure 4 shows the runtime per grid point for different custom kernels from Table 2 compared to the CuPy baseline with a fixed number of iterations. The number of iterations is set to $N = 1025$ such that it is large enough to make the initial overhead insignificant. Each optimization improves performance with the gains generally stemming from fewer kernel launches and less global memory access. An exception to that is the shared memory kernel: While it matches the Laplacian kernel for small grids, it slows down for larger grids. This is due to the fact that the reduction in global memory access from shared memory can not offset the performance loss from warp divergence, which is not significant for small grids.

The halo update kernel is effective for small grids but loses significance for large grids due to its linear scaling as opposed to the quadratic scaling behavior of applying the Laplacian. In contrast, the field update kernel is beneficial across all grid sizes because it also scales quadratically. The fused Laplacian calls perform best, minimizing kernel launches and global memory accesses. As shown in Subsection 3.1, graph capturing further enhances performance for small grids, which is noticeable up to a size of 64. The right part of Figure 4 shows the runtime per iteration

for the fused Laplacian implementation with and without graph capturing for a fixed grid. As previously with graph capturing, the gain is less pronounced for fewer iterations as there is the additional overhead of creating the graph.

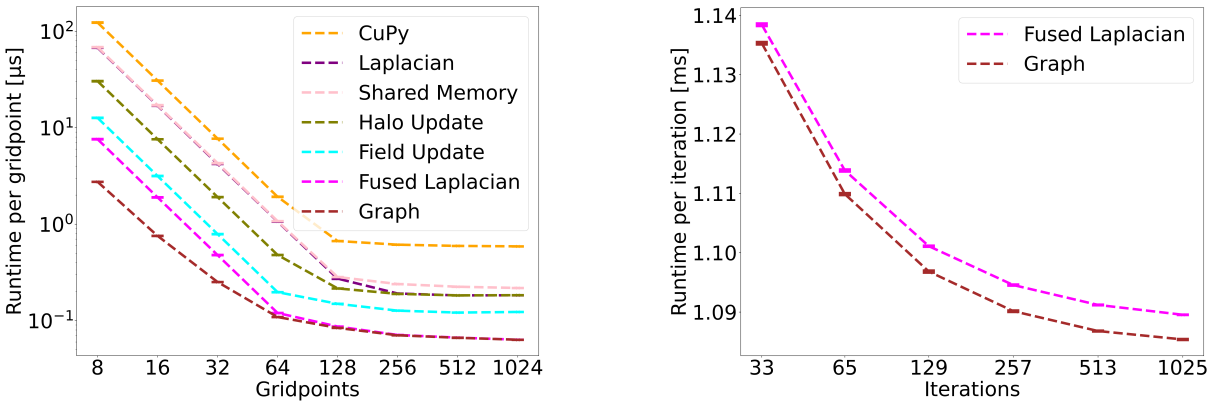


Figure 4: Left: Comparison of the runtime per grid point ($\frac{time}{n_x n_y n_z}$) of the CuPy baseline and the custom CUDA kernel implementation for different grid sizes (n_x,n_y) with a fixed number of vertical points and iterations ($n_z = 64$, $N = 1025$).
Right: Comparison of the runtime per iteration ($\frac{time}{N}$) of the Fused Laplacian implementation with and without graph capturing for different numbers of iterations with a fixed grid ($n_x = 128$, $n_y = 128$, $n_y = 64$).

4 Discussion

The benchmarks presented in the previous section leave some open questions warranting discussion: Which approaches would perform best if integrated into a full weather and climate modeling framework? Is the increased effort of writing user-defined kernels worth it, compared to the rather simple way in which we can extend an existing code with CUDA graphs?

Overall, the results are promising for the potential of CUDA graphs. From the benchmark with increasing grid size, it can be seen that CUDA graphs increase performance for small sizes, where the overhead of managing kernel launches is important compared to the time computations take to run. On the other hand, when the computations are on larger grid sizes, the overhead of launching the graphs becomes relatively insignificant. The CuPy and CUDA graphs implmentations show the same performance in that case. For large amounts of data, CUDA graphs are neither beneficial nor disadvantageous to performance.

The most significant improvement when using CUDA graphs can be found when looking at performance with an increasing number of iterations. CUDA graphs have an overhead linked to the capture and creation of the graph, which makes them slow compared to using only bare CuPy functionality for small amounts of iterations. Graphs become increasingly more efficient as the number of iterations. After about 50 iterations the CUDA graphs implementations already break even with the bare CuPy version and consistently outperform it at higher numbers of iterations. We further observe that the way in which the graph is captured can affect performance. The unrolled graph (one graph containing two loop iterations) may be a somewhat less obvious approach of defining the graph but results in a consistent performance improvement.

Custom kernels generally significantly improve performance over the CuPy baseline. For evaluating the pure graph capture version versus the custom CUDA kernel, Figures 3 and 4 (right) can be compared. For smaller grids, it can be seen that pure graph capturing outperforms custom kernels. Table 4 summarizes the grid size break-even point, where the custom kernels begins to

outperform the CUDA graphs implementations. It can be seen that only the most optimized fused Laplacian implementation always outperforms graph capturing. This shows that only custom kernels with high levels of complexity can also outperform CUDA graphs, underlining the appeal of CUDA graphs for very small grid sizes. They are an easy way of extending an existing code, which can yield remarkable performance benefits.

It was expected that more manual optimization would reduce the threshold since the number of kernel launches and global memory access decreases. It can not be distinguished between both effects in the timings for small grid sizes if the performance gain is similar. The gain for large sizes stems only from memory access reduction as launch overhead can be neglected compared to kernel runtime.

Table 4: Grid point threshold where custom implementations outperform unrolled graph/two-graphs implementation.

Version	Grid size threshold (n_x, n_y)
Laplacian	128
Shared Memory	128
Halo Update	64
Field Update	32
Fused Laplacian	0

Overall, we find that the results are within the expected scope, i.e. CUDA graphs generally outperform the CuPy baseline and the most sophisticated custom CUDA kernels outperform the CUDA graphs. Writing a custom kernel for this application is the best option if the aim is achieving maximum in performance. However, other factors have to be taken into account, such as the higher amount of effort and experience needed for their implementation. In a scenario where a quick increase in performance is needed, adding the graph capture can be a more easily attainable option.

In the context of weather and climate models, it is crucial to reduce the total computational costs of operations that have to be run repeatedly. This would point us towards the use of custom kernels. On the other hand, the code of weather and climate models is continuously undergoing optimization and therefore needs to be easy to understand and change, which could speak for using the simpler, albeit less performant graph capture approach. Even if the extension of a code with CUDA graphs requires less complicated syntax on the surface level, problems occurring in the background may be less foreseeable, like the memory copy issue we discuss in Section 2.2.

5 Conclusion

In this work, we have compared implementations of a stencil diffusion code using CUDA graph capturing and custom CUDA kernels. The resulting implementations achieve up to 30x performance speed-up compared to the universal function CuPy baseline. In detail, graph capturing results in high-performance gain for small problem sizes whereas custom kernels are always beneficial. Graph capturing even outperforms implementations with a few custom kernels for small sizes.

Overall, graph capturing offers a cost-effective alternative to custom optimization for improving performance in small-scale applications. It has a lower implementation cost and preserves

the original, easily understandable code. This approach illustrates a trade-off between performance, maintainability, and ease of implementation, while it does need to be considered that the superficiality of the implementation can make debugging more complicated.

For the custom kernels, it remains unclear if the benefit for small grid sizes stems from kernel number or global memory access reduction. Detailed memory bandwidth and launch overhead benchmarks would be needed to differentiate between both effects. In addition, it would be interesting to investigate graph capturing for more complex weather and climate models with hybrid CPU and GPU computations because of the reduced CPU overhead of graph capture.

References

[1] Tosiyuki Nakaegawa. "High-performance computing in meteorology under a context of an era of graphical processing units". In: *Computers* 11.7 (2022), p. 114.

[2] Alan Gray. *Getting Started with CUDA Graphs*. 2019. URL: https://developer.nvidia.com/blog/cuda-graphs (visited on 08/16/2024).

[3] NVIDIA corporation. *CUDA Graphs*. URL: https://www.olcf.ornl.gov/wp-content/uploads/2021/10/013_CUDA_Graphs.pdf (visited on 08/19/2024).

[4] Mert Demirsü and Axel Lervik. *Evaluating the performance of CUDA Graphs in common GPGPU programming patterns*. 2023.

[5] Ming Xue. "High-order monotonic numerical diffusion and smoothing". In: *Monthly Weather Review* 128.8 (2000), pp. 2853–2864.

[6] *Graph Management*. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html (visited on 08/16/2024).

[7] 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.

[8] *User-Defined Kernels — CuPy 13.3.0 Documentation*. URL: https://docs.cupy.dev/en/stable/user_guide/kernel.html#raw-kernels (visited on 08/28/2024).

[9] In: *CUDA Application Design and Development*. Elsevier, 2011, pp. 85–108. DOI: 10.1016/b978-0-12-388426-8.00004-5.