Efficient Execution of DG-FEM workloads on GPUs via CUDAGraphs

Mit Kotak

Senior Thesis

Faculty Mentor: Andreas Klöckner

Graduate Mentor: Kaushik Kulkarni

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Engineering Physics
in the Grainger College of Engineering of the
University of Illinois Urbana-Champaign, 2023

Abstract

Array programming paradigm offers routines to express the computation cleanly for a wide variety of scientific computing applications (Finite Element Method, Stencil Codes, Image Processing, Machine Learning, etc.). While there's ongoing work to provide efficient data structures and fast library implementations for many common array operations, the performance benefits are tied to optimized method calls and vectorized array operations, both of which evaporate in larger scientific codes that do not adhere to these constraints. There have been a lot of efforts in scaling up n-d array applications through kernel and loop fusion, but little attention has been paid towards harnessing the concurrency across array operations. The dependency pattern between these array operations allow multiple array operations to be executed concurrently. This concurrency can be targeted to accelerate the application's performance. NVIDIA's `CUDAGraph`s offers a task programming model that can help realize this concurrency by overcoming kernel launch latencies and exploiting kernel overlap by scheduling multiple kernel executions in parallel. In this work we map the array operations onto a precise data-flow graph and expose that to a GPU via `CUDAGraphs`. To evaluate the soundness of this approach, we port a suite of DG-FEM operators that represent real life workloads to our framework and observe a speedup of up to 32x over a version where the array operations are executed one after the other.

ACKNOWLEDGMENT

I would like to take this opportunity to thank all of the people without whom this thesis would have never happened. First and foremost, I would like to thank my advisor, Andreas Klöckner for his guidance and always being there to support me. My graduate mentor, Kaushik Kulkarni also deserves a huge thank you for guiding me through my research journey. I'm very grateful for the time and attention they have invested in me. I consider myself lucky to have had them as my mentors.

I am grateful to the *CEESD* (Center for Exascale-Enabled Scramjet Design) team for letting me be a part of their research endeavor and the broader NCSA (National Center for Supercomputing Applications) community for the opportunity to engage with undergraduate research. I would also like to thank the Office of Undegraduate Research for travel support which helped publicize this work.

I would like to thank my parents and my family in the US for their unwavering support.

Thank you, all of you.

Table of contents

List of figures

List of tables

1. Introduction

Array programming is a programming paradigm that supports a wide variety of features, including array slicing and arbitrary element-wise, reduction and broadcast operators allowing the interface to correspond closely to the mathematical needs of the applications. `PyCUDA`[16] and several other array-based frameworks (CuPy[22], Bohrium[17], Numba[19], Legate[8]) serve as drop-in replacements for mapping `Numpy` operations onto GPU memory. Incase of `PyCUDA`, this support is provided through the `GPUArray` interface. While abstractions like `GPUArray`'s offer a very convenient abstraction for doing "stream" computing[27] on these arrays, they are not yet able to automatically schedule and manage overlapping array operations onto multiple streams. The concurrency available in the dependency pattern for these array routines can be exploited to saturate all of the available execution units as shown in Fig 1.

Currently the only way to tap into this concurrency is by manually scheduling array operations onto multiple CUDA streams which typically requires a lot of experimentation since information about demand resources of a kernel such as GPU threads, registers and shared memory is only accessible at runtime. See [3] for a survey on the complexity of scheduling problems and [20] for a specific simpler case of independent task scheduling. Moreover, GPUs have many shared resources (caches, buses) and exhibit complex memory access patterns (NUMA effects), that render the precise estimation of the duration of these array operations extremely difficult.

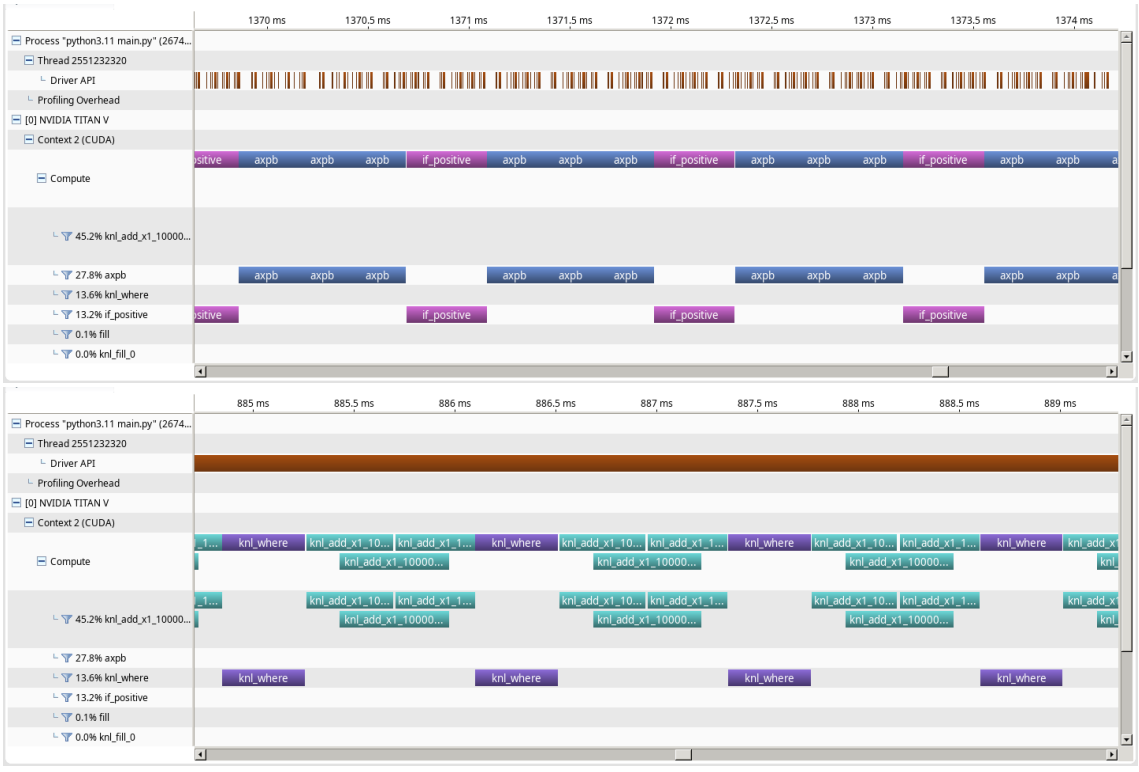


Figure 1. Profiles for CUDAGraph (bottom) and PyCUDA (top) for `where`(*condition, if, else*) + `1`

Our framework realizes this concurrency across array operations through `NVIDIA's CUDAGraphs`[1]. `CUDAGraph` is a task-based programming model that allows asynchronous execution of a user-defined Directed Acyclic Graph (DAG). See Fig 2. for an example of a `CUDAGraph`. When one places a kernel into a stream, the host driver performs a sequence of operations in preparation for the execution of the kernel. These operations are what are typically called "kernel overhead". To reduce this cost, CUDA graphs amortize the overheads of multiple kernel launches into one graph launch. However, since computing a graph is more expensive than running kernels directly[2], the performance gains only become apparent for large computations that *fill* the GPU.

One such class of discretizations that is able to scale up to modern GPU architectures is Discontinuous Galerkin Finite Element Method (DG-FEM). The characteristics of DG-FEM for PDE computations on GPUs have been investigated in [14], coming to the conclusion that speed-ups of factor 50 or more are possible. This makes DG-FEM an appealing application for `CUDAGraphs`. While our framework is generic, we evaluate the profitability of `CUDAGraphs` by targeting three end-to-end DG-FEM operators and observe a speedup of up to 32x.

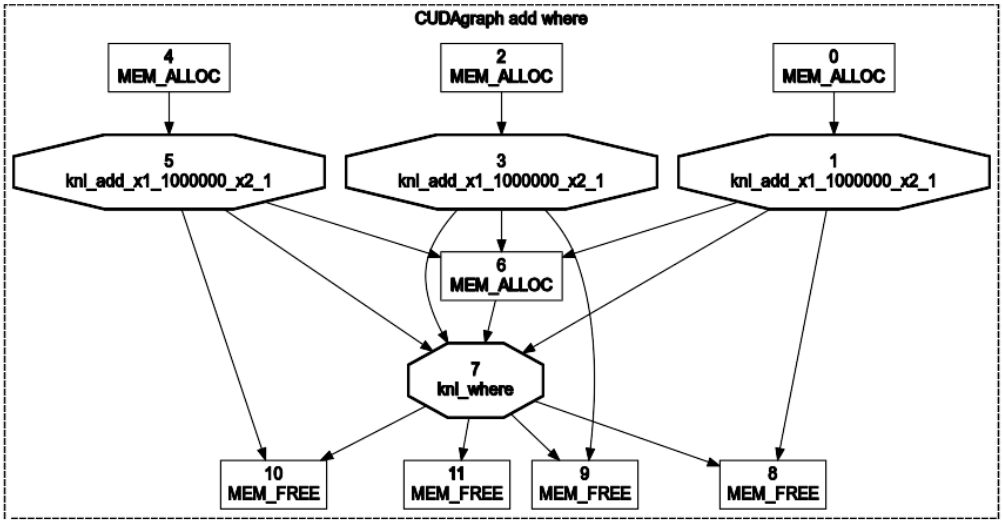


Figure 2. `CUDAGraph` API generated graph for `where`(*condition, if, else*) `+ 1`

We formulate our system by building a `CUDAGraph`-based `PyCUDA` target for `Pytato's` IR which captures the user-defined DAG. The process is *transparent*. The key technical contributions of this thesis involve:

1. Extending `PyCUDA` to allow calls to the `CUDAGraph` API

2. Mapping the array operations onto a DAG through `Pytato's` IR to generate `PyCUDA-CUDAGraph` code.

3. Providing an evaluation for profitability of `CUDAGraphs` for DG-FEM workloads.

2. Related work

Castro et el [11] gives an overview of the current task-based `Python` computing landscape by mentioning `PyCOMPs`[26], `Pygion`[25], `PyKoKKos`[6] and `Legion` [7] that rely on *decorators*. A decorator is an instruction set before the definition of a function. The decorator function transforms the user function (if applicable) into a parallelization-friendly. `PyCOMPs` and `Pygion` both rely on `@task decorator` to dynamically add tasks to the data dependency graph. The scheduling policy is *locality-aware* where the runtime system computes a score for all of the available resources and chooses the one with the highest score. The score is the number of task input parameters that are already present on that resource, thus minimizing delays between task executions. The main program of the application is a sequential Python script (or scripts) that contains calls to tasks. `Legion` uses a data-centric programming model which relies on *software out-of-order processor* (SOOP), for scheduling tasks which takes locality and independence properties captured by logical regions while making scheduling decisions.

In `Jug` [12] arguments take values or outputs of another tasks and parallelization is achieved by running more than one `Jug` processes for distributing the tasks. In `Pydron`[21], decorated functions are first translated into an intermediate representation and then analyzed by a scheduler which modifies the execution graph as each task is finished.

While all of these frameworks are able to leverage task-based parallelism, expressing array codes continues to remain a challenge.

`CuPy` serves as a drop-in replacement to `Numpy` and targets to `cuBLAS`, `cuDNN` and `cuSPARSE`. `Julia`[9] GPU programming models use `CUDA.jl` to provide high level mechanics to define multidimensional arrays (`CUArray`). Both `CuPy` and `Julia` offer interfaces for *implicit* graph construction which *captures* a `CUDAGraph` by recording all the operations on single or multiple streams. Although capturing all the operations on streams leads to terse application code, staging computations within a user-code with interleaving in-graph and out-of-graph operations cannot be expressed. The is can lead to repeated computations of the same sub-graphs.

`JAX`[10] optimizes GPU performance by translating *high-level traces* into XLA[24] HLO and then performing vectorization/parallelization and `JIT` compilation. Deep learning (DL) symbolic mathematical libraries such as `TensorFlow`[4] and `PyTorch`[23] allow neural networks to be specified as DAGs along which data is transformed. Both of them follow a delayed execution model where the DAG is built at run time, not at compile-time or eagerly. Each kernel's historical performance and scheduling is tracked to allow the creation of heuristics that guide future scheduling of the same kernel. The operators are then sequentially scheduled to a single computation stream in the GPU.

Both `StarPU`[5] and `ParSEC`[13] provide excellent support for heterogenous hardware on distributed systems. Both of them share a number of common features: tasks appear to execute in program order, dependencies between tasks are determined by the arguments supplied to task calls along with the privileges requested by tasks, and tasks can be offloaded to available GPUs (with data movement managed by the system). `ParSEC` in particular uses a DSL compiler to read a program representation (a recursive, algebraic description of a task graph) and generate code to execute the tasks described in the

program.

3. Overview

3.1. CUDA Graphs

CUDAGraphs provide a way to execute a partially ordered set of compute/memory operations on a GPU, compared to the fully ordered CUDA streams: : a stream in CUDA is a queue of copy and compute commands. Within a stream, enqueued operations are executed on the GPU in the same order as they are placed into the stream by the programmer with a single active task at a given instant. Thus, two kernels in the same stream cannot execute in parallel, even without data dependencies. This can lead to underutilization of GPU resources as shown in Fig 1. The solution is to run different CUDA streams in parallel through the use of CUDA events, which allow streams to synchronize with each other without blocking the host execution. However, using CUDA events to efficiently synchronize multiple complex streams by hand can be cumbersome.

CUDAGraphs offer a means to efficiently schedule kernel launches on multiple streams through a user-defined DAG. A CUDAGraph is a set of nodes representing memory/compute operations, connected by edges representing run-after dependencies.

> CUDA 10 introduced explicit APIs for creating graphs, e.g. *cuGraphCreate*, to create a graph; *cuGraphAddMemAllocNode*/*cuGraphAddKernelNode*/*cuGraphMemFreeNode*, to add a new node to the graph with the corresponding run-after dependencies with previous nodes to be executed on the GPU; *cuGraphInstantiate*, to create an executable graph in a stream; and a *cuGraphLaunch*, to launch an executable graph. We wrapped this API using PyCUDA which provided a high level Python scripting interface for GPU programming. Table 1. summarizes some of the operations offered by our PyCUDA-CUDAGraph interface.

Operations	**PyCUDA routines**
Memory Allocation	`add_memalloc_node`
Kernel Execution	`add_kernel_node`
Host to Device Copy	`add_memcpy_htod_node`
Device to Device Copy	`add_memcpy_dtod_node`
Device to Host Copy	`add_memcpy_dtoh_node`
Memory Free	`add_memfree_node`
Graph Creation	`Graph`
Graph Instantiation	`GraphExec`
Update ExecGraph arguments	`batched_set_kernel_node_arguments`
Graph Launch	`launch`

Table 1. PyCUDA wrapper functions around CUDAGraph API

3.2. Loopy

Loopy[15] is a Python-based transformation toolkit to generate transformed kernels. We make use of the following components in our pipeline to generate performance tuned CUDA kernels:

1. *Loop Domains*: The upper and lower bounds of the result array's memory access pattern in the OpenCL format sourced from the shape attribute within IndexLambda and expressed using the isl library.

2. *Statement:* A set of instructions specified in conjunction with an iteration domain which encodes an assignment to an entry of an array. The right-hand side of an assignment consists of an expression that may consist of arithmetic operations and calls to functions.

3. *Kernel Data*: A sorted list of arguments capturing all of the array node's dependencies.

Algorithm 1: Loopy kernel for doubling operation

```
lp.make_kernel(
  domains = "{[_0]:0<=_0<4}}",
  instructions = "out[_0]=2*a[_0]",
  kernel_data = [lp.GlobalArg("out", shape=lp.auto, dtype="float64"),
                 lp.GlobalArg("a", shape=lp.auto, dtype="float64")])
```

3.3. Pytato

Pytato[18] is a lazy-evaluation programming based Python package that offers a Numpy-like frontend for recording array expressions.

Pytato offers an IR which encodes user defined array computations as a DAG where nodes correspond to array operations and edges represent dependencies between inputs/outputs of these operations. Refer to Fig. 3 for an example. In this work, we are interested in the normalized form of Pytato IR, which is comprised of the following two node types:

1. *Placeholder*: A named abstract array whose shape and dtype is known with data supplied during runtime. This permits the automated gathering of a self-contained description of a

piece of code without incurring the penalty faced by repeated memory transfers from the device's DRAM to lower levels of cache.

2. *IndexLambda*: Represents an array comprehension recording a scalar expression containing per index value of the array computation. This helps create a generalized expression for expressing array computations.

Alg. 2 shows a simple example demonstrating `Pytato` usage

Algorithm 2: Pytato expression building for doubling operation

```
# Create Placeholder node for storing array description

x = pt.make_placeholder(name="x", shape=(4,4), dtype="float64")

# Express array computation as a scalar expression using Indexlambda

result = 2*x
```

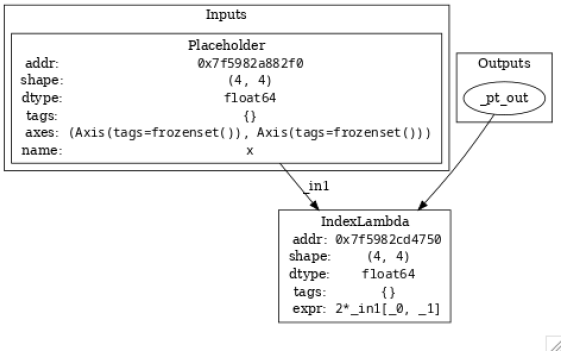


Figure 3. Pytato IR corresponding to doubling operation

4. Lowering Array Operations to CUDAGraphs

We make use of lazy evaluation where the definition and submission of work are decoupled. `CUDAGraphs` follow a delayed execution model where no operations to the GPU are submitted during graph building. The scheduler is made aware of a defined dependency graph ahead of execution. Since `Pytato`'s computation graph maps precisely onto `CUDAGraphs`, we implement a `Pytato-CUDAGraph` visitor for rewriting `Pytato` IR expressions. `Pytato` code is lowered onto CUDAGraph through a two stage code generation process as shown in Alg 3. The code generation gets triggered by passing the `Pytato` expression created in Alg 2 to `pt.generate_cudagraph`.

Algorithm 3: Pytato-PyCUDAGraph generated code

```
import pycuda.driver as _pt_drv
import numpy as np
from pycuda.driver import KernelNodeParams as _pt_KernelNodeParams
from pycuda.compiler import SourceModule as _pt_SourceModule
from pycuda import gpuarray as _pt_gpuarray
from functools import cache

# {{{ Create and load kernel module

_pt_mod_0 = _pt_SourceModule("

#define bIdx(N) ((int) blockIdx.N)\n#define tIdx(N) ((int)
threadIdx.N)\n\nextern "C" __global__ void __launch_bounds__(16)
knl_mult_x1_1_x2_16(double *__restrict__ out, double const *__restrict__
_in1)\n{\n  {\n    int const ibatch = 0;\n\n    out[4 * (tIdx(x) / 4) +
tIdx(x) + -4 * (tIdx(x) / 4)] = 2l * _in1[4 * (tIdx(x) / 4) + tIdx(x) +
-4 * (tIdx(x) / 4)];\n  }\n}

")

# }}}
# {{{ Stage 1: Build and cache CUDAGraph

@cache
def exec_graph_builder():
    _pt_g = _pt_drv.Graph()
    _pt_buffer_acc = {}
    _pt_node_acc = {}
    _pt_memalloc, _pt_array = _pt_g.add_memalloc_node(size=128,
dependencies=[])
    _pt_kernel_0 = _pt_g.add_kernel_node(_pt_array, 139712027164672,
func=_pt_mod_0.get_function('knl_mult_x1_1_x2_16'), block=(16, 1, 1),
grid=(1, 1, 1), dependencies=[_pt_memalloc])
    _pt_buffer_acc['_pt_array'] = _pt_array
    _pt_node_acc['_pt_kernel_0'] = _pt_kernel_0
    _pt_g.add_memfree_node(_pt_array, [_pt_kernel_0])
    return (_pt_g.get_exec_graph(), _pt_g, _pt_node_acc, _pt_buffer_acc)

# }}}
# {{{ Stage 2: Update execution graph

def _pt_kernel(allocator=cuda_allocator, dev=cuda_dev, *, a):
    _pt_result = _pt_gpuarray.GPUArray((4, 4), dtype='float64',
allocator=allocator, dev=dev)
    _pt_exec_g, _pt_g, _pt_node_acc, _pt_buffer_acc =
exec_graph_builder()
    _pt_exec_g.batched_set_kernel_node_arguments(
    {_pt_node_acc['_pt_kernel_0']:
_pt_drv.KernelNodeParams(args=[_pt_result.gpudata, a.gpudata])})
    _pt_exec_g.launch()
    _pt_tmp = {'_pt_out': _pt_result}
    return _pt_tmp['_pt_out'].get()

 # }}}
```

4.1. Stage 1: Build CUDAGraph

Since the CUDAGraph runtime scheduler takes in a fully defined dataflow graph, we use Alg 4 to explore all of the array dependencies in the computation graph. We cache the resultant executable graph since the topology stays constant throughout the computation. This ensures that Alg 4 only gets executed only once during compilation with a $\Theta(V+E)$ complexity for Alg 5.

Algorithm 4: DAG Discovery for building CUDAGraph

Input: Pytato array computation graph
Output: pycuda.Graph object

Step 1: Initialize a pycuda.Graph object.

Step 2:
Output ← nodes in array computation graph with no successors.
Code ← ψ
ArrayToBuffers ← ψ
for $n \in$ *Output* **do**
Code, *ArrayToBuffers* ← GraphTraverse(*n, Code, ArrayToBuffers*)
done

Step 3: Instantiate pycuda.Graph object and cache the resultant pycuda.GraphExec object to avoid triggering graph traversals for subsequent launches.

The exec_graph_builder function in Alg. 3 describes the graph building phase. Since arrays are being lazily evaluated during the building phase, all placeholders are replaced with temporary array buffers which are then updated during the execution phase. For every kernel node, the resulting array is allocated using add_memalloc_node and array operation is expressed through add_kernel_node. Instead of manually searching the parameter space, we generate the kernel string and the launch configuration by passing the corresponding IndexLambda to Loopy.

Algorithm 5: Array computation graph traversal

function PLACEHOLDERMAPPER(*n, ArrayToBuffers*)
    ArrayToBuffers[*n*] ← User provided buffer OR allocate new buffer using `GPUArrays`
    return *ArrayToBuffers*
end function

function LOOPYKERNEL(*n*)
    {Returns the kernel string and launch configuration}
end function

function INDEXLAMBDAMAPPER(*n, Code, ArrayToBuffers*)
    Insert `CUDAGraph` memalloc node code for result array
    ArrayToBuffers[*n*] ← Buffer corresponding to allocated result array
    kernelString, grid, block = LOOPYKERNEL(*n*)
    Insert `CUDAGraph` kernel node code with temporary buffers for bindings into *Code*
    return *Code, ArrayToBuffers*
end function

function GRAPHTRAVERSE(*n, Code, ArrayToBuffers*)
    if *n* ϵ {`PlaceHolder`, `DataWrapper`} {
      ArrayToBuffers ← PLACEHOLDERMAPPER(*n, ArrayToBuffers*)
      return {*n*}, *Code, ArrayToBuffers*
    }
    else {
      Code, ArrayToBuffers ← INDEXLAMBDAMAPPER(*n, Code, ArrayToBuffers*)
      n_deps ← ψ
      bindings ← `IndexLambda` bindings for *n*
      for c ϵ *bindings* **do**
          c_deps, Code, ArrayToBuffers ← GRAPHTRAVERSE(*c, Code,ArrayToBuffers*)
          n_deps ← *n_deps* ⋃ *c_deps*
      done
      return *n_deps*, *Code, ArrayToBuffers*
    }
end function

4.2. Stage 2: Update `CUDAGraphExec`

Since the input parameters of the computation graph change with every integration step, the corresponding `CUDAGraph` also changes. To avoid triggering a graph compilation for every iteration, we use `PyCUDA` wrappers around *cuGraphExecSetKernelParams* functionality.

Since the graph topology does not change over different iterations, we are able to update the cached executable graph with new kernel parameters. This helps us avoid the expensive instantiation of a new graph. Thus, instead of Alg. 4, Alg. 6 gets executed for every graph

launch with $\Theta(n)$ complexity where n is the number of kernel nodes with temporary buffers which is a subset of all the nodes in the graph. In Alg. 3, this corresponds to the routine enclosed in `_pt_kernel`.

Algorithm 6: Buffer update in `CUDAGraphExec`

Nodes ← kernel nodes in `pycuda.GraphExec` with temporary buffers
for n ∈ *Nodes* **do**
    Replace temporary buffers with allocated/linked buffers from corresponding `PlaceHolder` nodes
done

5. Results

5.1. Experimental Setup

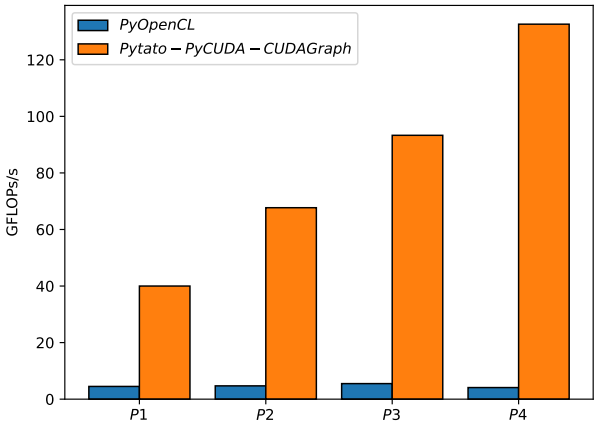
Applications: We demonstrate the performance of our framework on three end-to-end DG-FEM operators: Wave, Euler and Compressible Navier Stokes. We evaluated these operators on 3D meshes with tetrahedral cells. Our experimental parameters have been summarized in Table 2.

Tools: Two types of tools were used: (1) Compilers, NVCC V11 for CUDAGraphs and POCL-CUDA 3.1 for `PyOpenCL`, and (2) Analysis tools like `nvprof` and `nvvp`.

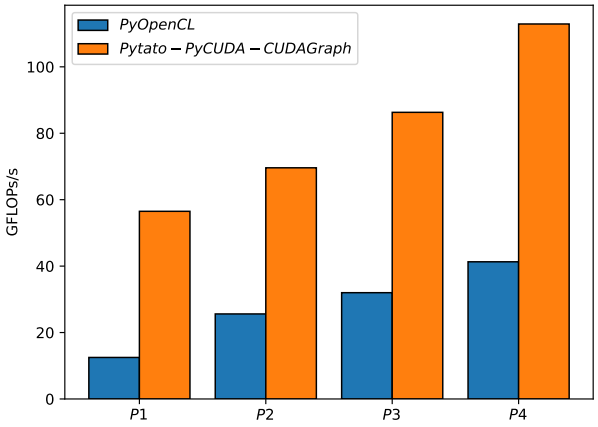
Platform: All of our experiments were performed on GPU NVIDIA TITAN V with 6144 GFLOPs/s peak double precision and 652.8 GB/s peak bandwidth.

Equation	**Polynomial degrees**	**# of tetrahedrons in the mesh**
Wave	1	1.25×10^5
	2	5.0×10^4
	3	2.5×10^4
	4	1.4×10^4
Euler	1	3.2768×10^4
	2	1.3284×10^4
	3	6.859×10^3
	4	4.913×10^3
Compressible Navier Stokes	1	8.2944×10^4
	2	4.8000×10^4
	3	2.4576×10^4
	4	1.0368×10^4

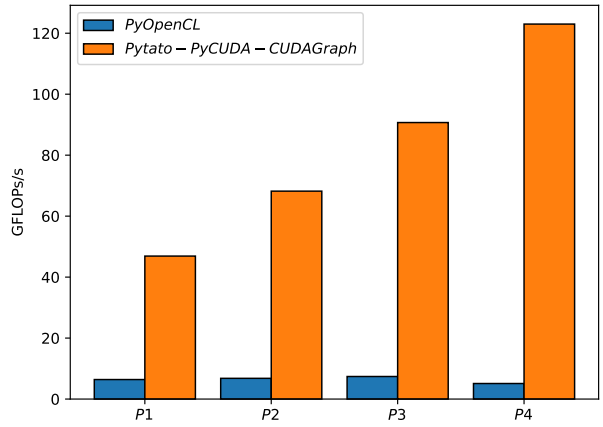
Table 2. Experimental parameters for DG-FEM operators



Wave



Euler



Compressible Navier Stokes

Figure 4. Performance of our framework (`Pytato-PyCUDA-CUDAGraph`) for DG-FEM operators over sequential stream execution (`PyOpenCL`).

5.2. Performance Evaluation

Methodology: We measure our speedup over `PyOpenCL` where the array operations are executed one after the other in a single stream. We used wall clock times for our measurement with 2 seconds being spent in the warmup loop and 5 seconds for the iteration loop.

We observe speedups of up to 8-32x for Wave, 2-4x for Euler and 2-24x for Compressible Navier Stokes with the resuling performance being closely tied to the polynomial order as reported in [14]. The large variation in performance can be attributed to the difference in computation graph topologies for each operator. We also note that the performance is largely limited by memory bandwidth. As observed in Fig. 2, the scheduler maximally parallelizes the given CUDAGraph without limiting the stream usage. Thus, in large graphs , the GPU memory can explode which in this case limits the scaling to higher mesh resolutions.

6. Conclusion

In this work we realize the concurrency available across array operations through NVIDIA's CUDAGraph task programming model. `CUDAGraphs` overcome the limitations of single stream execution through a user defined DAG that can be executed on GPUs using multiple streams and low kernel latencies. Firstly, we extend the PyCUDA GPU scripting framework to wrap around the CUDAGraph API. Next, we implement a pipeline for lowering array operations onto CUDAGraph using `Pytato` which is a lazy evaluation-based array package. And finally we assess the profitability of `CUDAGraphs` for DG-FEM workloads by evaluating our framework on three end-to-end DG-FEM operators . We record a speedup of up to 32x for Navier Stokes operator over sequential stream execution.

For future work, we plan to come up with a performance model for the scheduling algorithm through a series of microbenchmarks. We will also leverage additional `Pytato` IR information to further optimize `Loopy` kernels.

Bibliography

[1] Effortless cuda graphs. in: nvidia gpu technology conference (gtc) (2021).

[2] Getting started with cuda graphs.

[3] http://www2.informatik.uni- osnabrueck.de/knust/class/. Complexity results for scheduling problems.

[4] Mart∈ Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: large-scale machine learning on heterogeneous distributed systems. 5 2016.

[5] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, 2011.

[6] Nader Al Awar, Steven Zhu, George Biros, and Milos Gligoric. A performance portability framework for python. ACM, 6 2021.

[7] Michael Bauer. Legion: programming distributed heterogeneous architectures with logical regions. 2014.

[8] Michael Bauer and Michael Garland. Legate numpy: accelerated and distributed array computing. 2019.

[9] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: a fast dynamic language for technical computing. 2012.

[10] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. Jax: composable transformations of python+numpy programs. 2018.

[11] Oscar Castro, Pierrick Bruneau, Jean-Sébastien Sottet, and Dario Torregrossa. Landscape of high-performance python to develop data science and machine learning applications. 5 2023.

[12] Luis Pedro Coelho. Jug: software for parallel reproducible computation in python. *Journal of Open Research Software*, 5:30, 10 2017.

[13] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. Dynamic task discovery in parsec- a data-flow task-based runtime. 2017.

[14] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. Nodal discontinuous galerkin methods on graphics processors. *Journal of Computational Physics*, 228, 2009.

[15] Andreas Klöckner. Loo.py: transformation-based code generation for gpus and cpus. Association for Computing Machinery, 2014.

[16] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda and pyopencl: a scripting-based approach to gpu run-time code generation. *Parallel Computing*, 38:157–174, 2012.

[17] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: unmodified numpy code on cpu, gpu, and cluster. *Proceedings of the Python for High Performance and Scientific Computing Workshop (PyHPC 2013)*, 2013.

[18] Kaushik Kulkarni and Andreas Klöckner. Separating concerns in computational science without domain specific languages. 2023.

[19] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a llvm-based python jit compiler. Volume 2015-January. 2015.

[20] Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46, 1990.

[21] Stefan C Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. Pydron: semi-automatic parallelization for multi-core and the cloud. Pages 645–659. USENIX Association, 10 2014.

[22] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: a numpy-compatible library for nvidia gpu calculations. 2017.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: an imperative style, high-performance deep learning library. 5 2019.

[24] Amit Sabne. Xla : compiling machine learning for peak performance. 2020.

[25] Elliott Slaughter and Alex Aiken. Pygion: flexible, scalable task-based parallelism with python. IEEE, 11 2019.

[26] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. Pycompss: parallel computational workflows in python. *The International Journal of High Performance Computing Applications*, 31:66–82, 7 2016.

[27] Suresh Venkatasubramanian. The graphics card as a streaming computer. *In SIGMOD Workshop on Management and Processing of Massive Data (June 2003).*