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 fulfilment of the requirements
for the degree of Bachelor of Science in Engineering Physics
in the Granginer College of Engineering of the
University of Illinios Urbana-Champaign, 2023

Abstract

Array programming paradigm offers routines to express the computation cleanly for 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 harnesssing 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 accelarate the application's performance. NVIDIA's `CUDAGraph` API offers a task programming model that can help realise this concurrency by overcoming kernel launch latencies and exploiting kernel overlap by scheduling multiple kernel executions in parallel. In this work we create a task-based lazy-evaluation array programming interface by mapping array operations onto `CUDAGraphs` using `Pytato's` IR and `PyCUDA's` GPU scripting interface. To evaluate the soundness of this approach, we port a suite of complex operators that represent real world workloads to our framework and compare the performance with a version where the array operations are executed one after the other. We observe a performance of upto X for Wave operators, Y for Euler Operators and X for Compressible Navier Stokes.

ACKOWLEDGEMENT

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 endevaour and the broader NCSA (National Center for Supercomputing Applications) community for the opportunity to engage with undergradaute 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 extended 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 arbitary element-wise, reduction and broadcast operators allowing the interface to correspond closely to the mathematical needs of the applications. `PyCUDA`[11] and several other array-based frameworks (CuPy[16], Bohrium[12], Numba[14], Legate[3]) serve as drop-in replacements for accelarating `Numpy-like` operations on GPUs. All of them offer array interfaces 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 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 [Fig. 1].

Currently the only way to tap into this concurrency is by manually scheduling array operations onto mutliple 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.

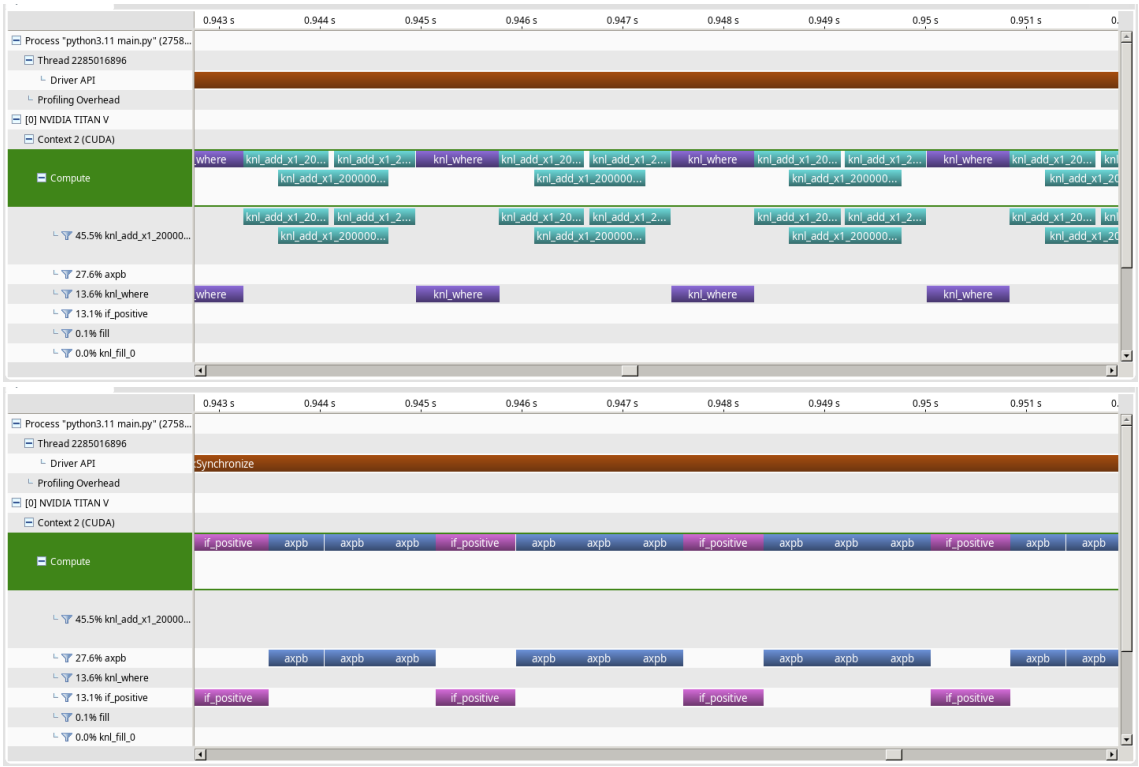

Figure 1. Profiles for CUDAGraph (top) and PyCUDA (bottom) for `where`(*condition, if, else*) `+ 1`

Our framework realises this concurrency across array operations through `NVIDIA's CUDAGraph` API. `CUDAGraph`, first introduced in `CUDA 10`, is a task-based programming model that allows asynchronous execution of a user-defined Directed Acyclic Graph (DAG). These DAGs are made up of a set of node representing operations such as memory copies and kernel launches, connected by edges representing run-after dependencies which are defined separetly from its execution through a custom API.

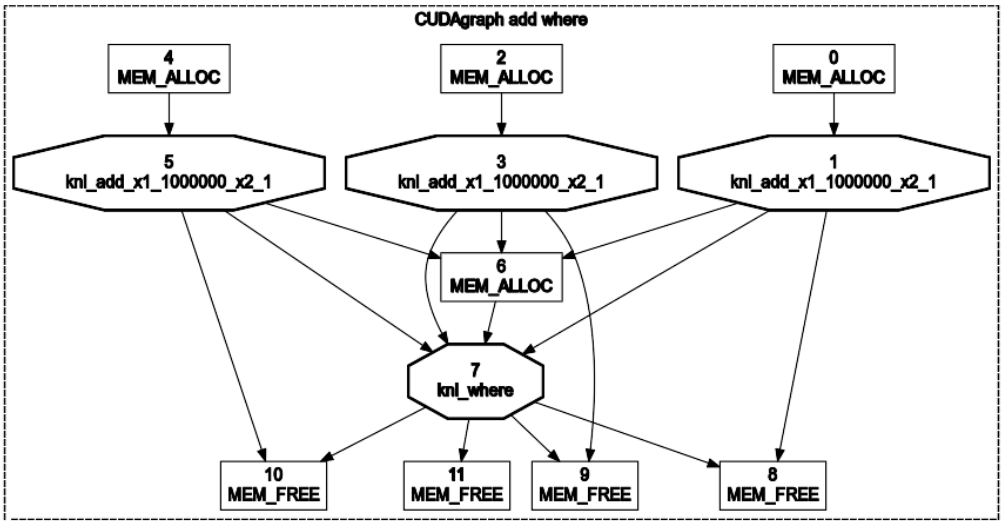

Figure 2. `CUDAGraph` API generated graph for `where`(*condition, if, else*) + `1`

We formulate our system by buiding a `CUDAGraph`-based `PyCUDA` target for `Pytato's` IR which captures the user-defined DAG. The process is *transparent*. The key technical contributions of our system 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 profitablity of `CUDAGraphs` for DG-FEM workloads.

2. Related work

Castro et el [6] gives an overview of the current task-based `Python` computing landscape by mentioning `PyCOMPs`[?], `Pygion`[19], `PyKoKKos`[?] and `Legion` [?] 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 version. `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. `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` [7] 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`[15], 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.

Since all of these frameworks rely on explicit taks declarations, they are not yet able to realise the concurrency available across array operations.

`CuPy` serves as a drop-in replacement to `Numpy` and support NVIDIA's in-house CUDA frameworks such as `cuBLAS`, `cuDNN` and `cuSPARSE`. `Julia`[4] GPU programming models use `CUDA.jl` to provide high level mechanics to define multidimenstional arrays (`CUArray`). Both `CuPy` and `Julia` offer interfaces for *implcit* graph construction which *captures* a `CUDAGraph` using existing stream-based APIs. Although capturing all the operations on a stream 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`[5] optimizes GPU peformance by translating *high-level traces* into XLA[18] HLO and then performing vectorization/parallelization and `JIT` compilation. Deep learning (DL) symbolic mathematical libraries such as `TensorFlow`[1] and `PyTorch`[17] allow neural networks to be specified as DAGs along which data is transformed. Current DL frameworks conduct GPU task scheduling during *run time*. Tensorflow represents a neural network as a computation graph of DL operators, and schedule athe GPU tasks of an operator at run time once the operator's dependencies are met. Meanwhile, for PyTorch , GPU tasks are scheduled at run time as Python code is interpreted line by line. Just like `CUDAGraphs`, in `TensorFlow`, computational DAGs are defined statically to achieve close to roofline performance for DL applications. `PyTorch` on the other hand offers more control at runtime by allowing the modification of executing nodes facilitating the implementation of sophosticated training routines.

Both `StarPU`[2] and `ParSEC`[8] 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, enqueueud operations are implicitly synchronized by the GPU in order to execute them in the same order as they are placed into the stream by the programmer. Streams allow for aschnronous compute and copy, meaning that CPU cores dispatch commands without waiting for their GPU-side completition: even in asynchronous submissions, little to no control is left to the programmer with respect to when commands are inserted/fetched to/from the stream and then dispatched to the GPU engines, with these operations potentially overallaping in time.

CUDAGraphs faciliate the mapping of independent 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 exected 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.

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[10] 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 specificed in conjuction 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.

```
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[13] 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. 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

Here's a simple example demonstrating Pytato usage

```
# CreatePlaceholder 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
```

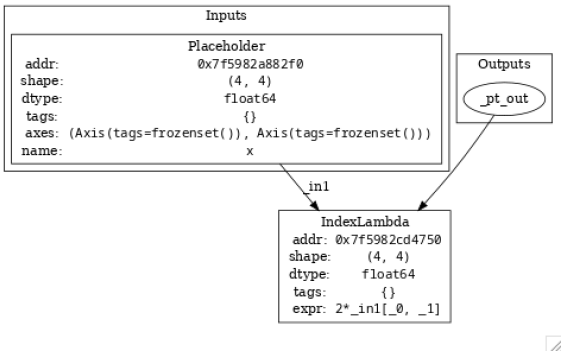

Figure 3. Pytato IR corresponding to doubling operation

4. Lowering Array Operations to CUDAGraphs

Pytato provides a pt.generate_cudagraph function which triggers a two-stage code generation process that traces the array program and generates PyCUDA-CUDAGraph code. Our array programming system is based on lazy-evaluatation which offers us the flexibilty of avoiding repeated memory transfer's from the DRAM while transforming the array expressions to CUDAGraph code. Since Pytato's computation graph could be exactly mapped onto the CUDAGraph interface, we implemented a Pytato-CUDAGraph vistor for rewriting

Pytato IR expressions. For the kernel nodes, instead of manually searching the parameter space, we offloaded decisions such as address spaces of buffers, mapping to device axes, loop tiling, etc. onto Loopy.

The following code is generated by passing the computation graph described in Fig 3 into pt.generate_cudagraph

```
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`

Alg 1 only gets executed only once during compilation with a $\Theta(V+E)$ complexity for Alg 2

Algorithm 1: DAG Discovery for building `CUDAGraph`

Step 1: Run a topological sort on `Pytato` IR using Kahn's algorithm [9]. This frontloads the *sink* nodes which helps avoid array recomputations during DAG discovery. Initialize a `pycuda.Graph` object.

Step 2:
for $n \epsilon$ nodes in `Pytato` IR which only have incoming edges **do**
    GraphTraverse(n)
done

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

Algorithm 2: Pytato IR Traversal

function GraphTraverse(n)
    if $n \epsilon$ {`PlaceHolder`, `DataWrapper`} {

- PlaceHolderMapper(n)
- Link to user provided buffers or generate new buffers via `GPUArrays`.

    return {n}
    }
    else {

- IndexLambdaMapper(n)
- Generate kernel string and launch dimensions by plugging `IndexLambda` expression into `lp.make_kernel`.
- Add kernel node with temporary buffer arguments and corresponding result memalloc node to `pycuda.Graph` object with dependencies sourced from `Pytato` IR.

- Update `Pytato` IR with termporary buffer information.

n_deps ← {}
for c ϵ *n* dependencies sourced from `Pytato` IR **do**
c_deps ← GraphTraverse(*c*)
n_deps ← *n_deps* $\bigcup$ *c_deps*
done
return *n_deps*
}
end function

4.2. Stage 2: Update `CUDAGraphExec`

Alg 3. gets executed for every graph launch.

Algorithm 3: Buffer update in `CUDAGraphExec`

for n ϵ kernel nodes in `pycuda.GraphExec` with temporary buffers **do**

- Replace temporary buffers with allocated/linked buffers from corresponding `PlaceHolder` nodes.

done

5. Results

We demonstrate the performance of our framework on two end-to-end DG-FEM operators on NVIDIA Titan V in Fig 4. We evaluate these operators on 3D meshes with tetrahedral cells and measure our speedup against `PyOpenCL`[11] which supports sequential stream execution. Table 2. summarizes our experimental parameters.

Equation	Polynomial Degree	# of tetrahedrons in the mesh
3D Wave	1	27000
	2	15625
	3	3375
	4	1000
3D Euler	1	27000
	2	15625
	3	3375
	4	1000

Table 2. Experimental parameters for DG-FEM operators

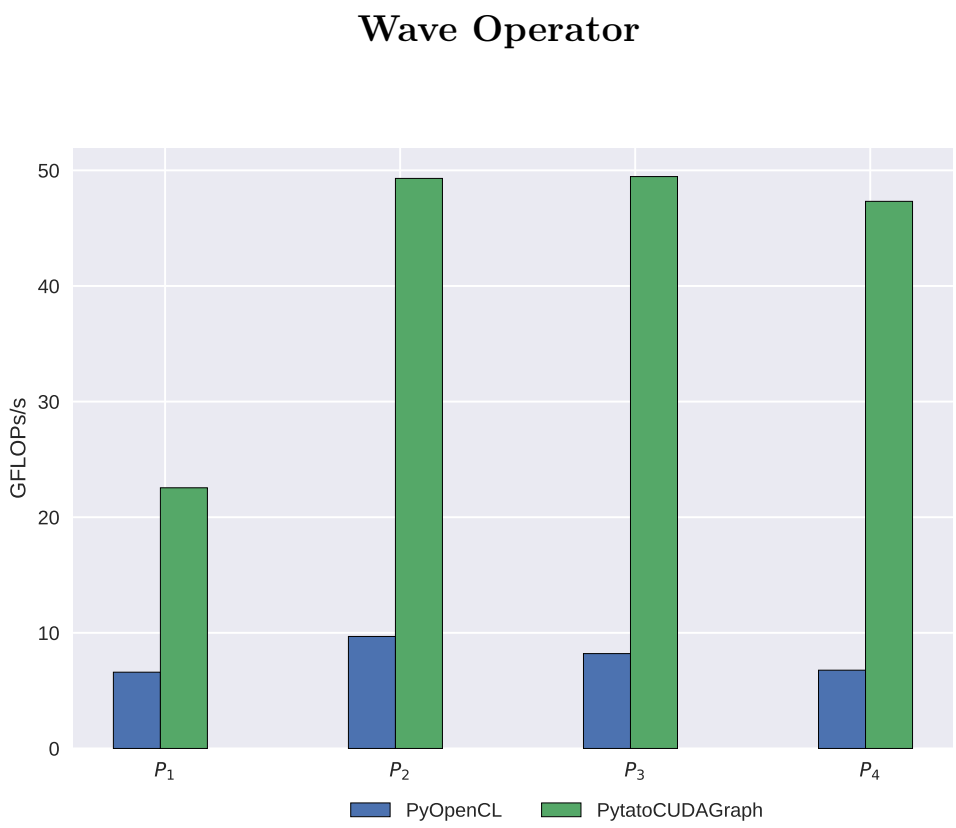

Figure 4. Performance of our framework (`Pytato-PyCUDA-CUDAGraph`) over sequentual stream execution (`PyOpenCL`)

6. Conclusion

In this work we implement an array framework powered by a task-graph based backend for realizing concurrency across array operations. This enables domain experts to write array-based DG-FEM code that can be efficiently executed on GPUs through on multiple streams and low kernel launch latencies. We demonstrate the performance of our system on Wave, Euler and Navier Stokes Opertators and recorded a speedup of upto X over sequential stream execution.

Some future extensions of this work involve (1) creating a performance model for the graph scheduling algorithm through a series of microbenchmarks, (2) providing a mechanism for hand-tuning `Loopy` kernels.

Bibliography

[1] 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.

[2] 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.

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

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

[5] 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.

[6] 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.

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

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

[9] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5, 1962.

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

[11] 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.

[12] 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.

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

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

[15] 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.

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

[17] 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.

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

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