Efficient Execution of DG-FEM workloads on GPUs via `CUDAGraphs`

Mit Kotak

Senior Thesis

Faculty Mentor: Andreas Klöckner

Graduate Mentor: Kaushik Kulkarni

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 these routines are optimized 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. While there have been a lot of efforts in scaling up n-d array applications through kernel and loop fusion, very 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.

1. INTRODUCTION

Array programming is a fundamental computation model 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` and several other array-based frameworks serve as drop-in replacements for accelarating `Numpy-like` operations on GPUs. 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.

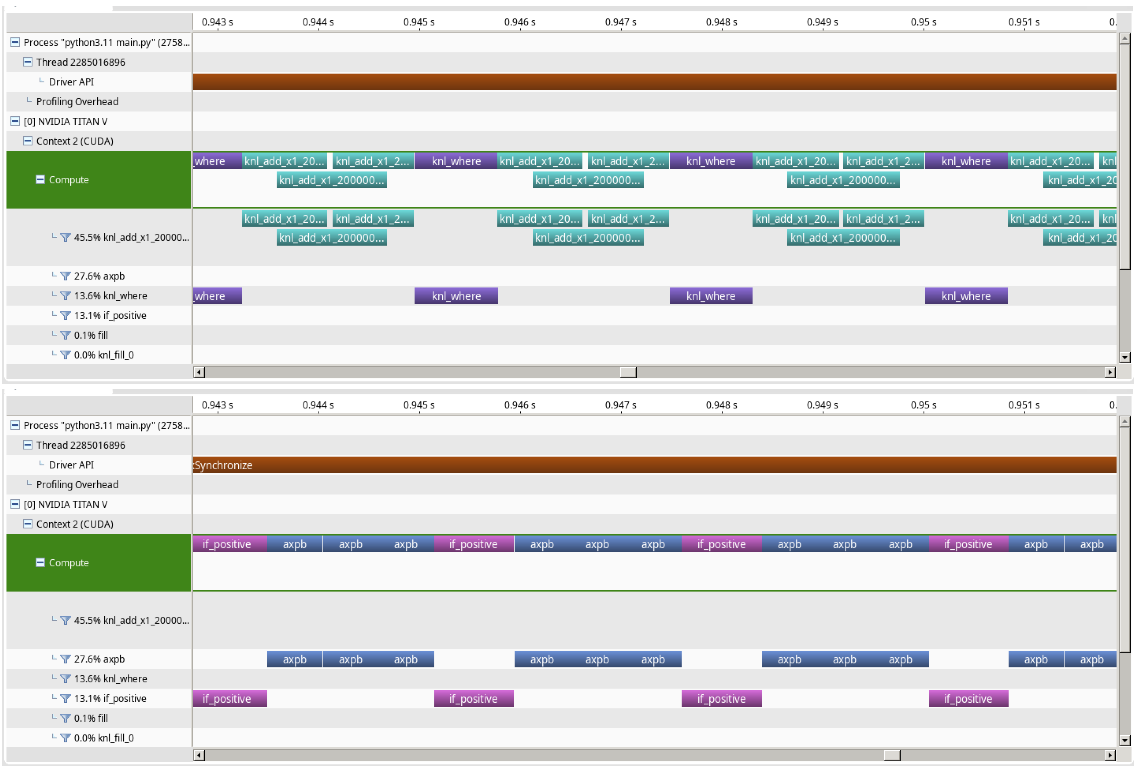

Figure 1. Profiles for CUDAGraph (top) and PyCUDA (bottom) for `where`(*condition, if, else*) `+ 1`

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.

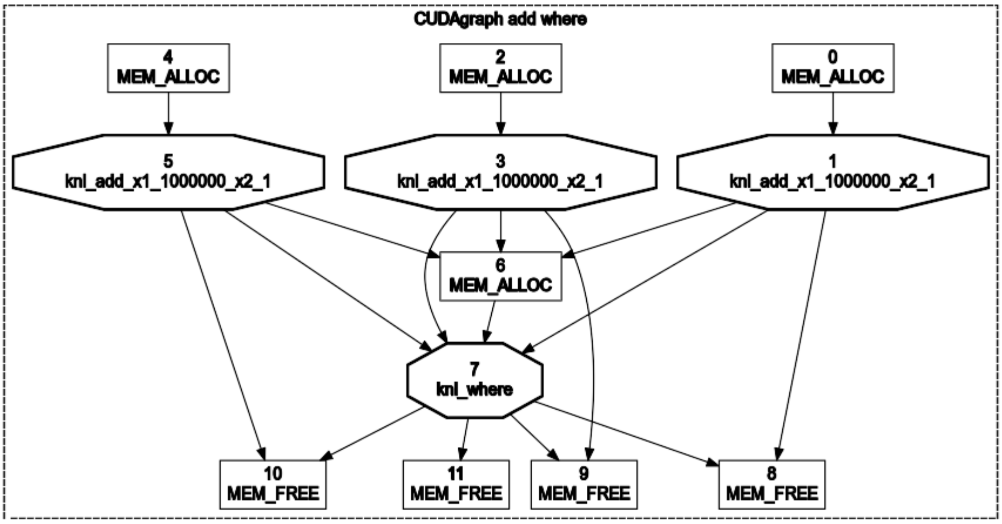

Figure 2. `CUDAGraph` API generated graph for `where`(*condition, if, else*) `+ 1`

We formulate our system by buiding a `CUDAGraph`-based `PyCUDA` target for `Pytato's` IR which captures the user-defined DAG. The 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.

2. Related work

The literature on task-based array programming can be classified roughly according to their choice of task granularity.

Function: Castro et give an overview of the current task-based `Python` computing landscape by mentioning several libraries 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. Libraries such as `PyCOMPs`, `Pygion`, `PyKoKKos` and `Legion` make use of this core principle to accelarate *vanilla* `Python` code. `PyCOMPs` and `Pygion` both rely on `@task` decorator to build a task dependency graph and define the order of execution. `PyKoKKos` ports into the `KoKKos` API and passes the `@pk.workunit` decorator into the `parallel_for()` function. `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`, 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`, decorated functions are first translated into an intermediate representation and then analyzed by a scheduler which updates the execution graph as each task is finished.

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

Stream: CuPy serves as a drop-in replacement to Numpy and uses NVIDIA's in-house CUDA frameworks such as cuBLAS, cuDNN and cuSPARSE to accelerate its performance. Julia GPU programming models use CUDA.jl to provide a 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. Implicit CUDAGraph construction is more flexible and general, but requires to wrangle with con-concurrency details through events and streams.

Graph: JAX optimizes GPU peformance by translating *high-level traces* into XL HLO and then performing vectorization/parallelization, automatic differentiation, and JIT compilation. Deep learning symbolic mathematical libraries such as TensorFlow and Pytorch allow neural networks to be specified as DAGs along which data is transformed. Just like CUDAGraphs, in TensorFlow, computational DAGs are defined statically so that their compilation and execution yield maximum performance. PyTorch on the other hand offers more control at runtime by allowing the modification of executing nodes facilitating the implementation of sophosticated training routines.

Kernel: StarPU supports a task-based programming model by scheduling tasks efficiently using well-known generic dynamic and task graph scheduling policies from the literature, and optimizing data transfers using prefetching and overallaping. Each StarPU task describes the computation kernel, possible implementations on different architectures (CPUs/GPUs), what data is being accessed and how its accessed during comptuation (read/write mode). Task dependencies are inferred from data dependencies.

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 introduces 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. The table below lists commonly used PyCUDA-CUDAGraph functions. Refer to [link] for a comprehensive list of wrapped functions.

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

Here's a simple example demonstrating CUDAGraph functionality:

```
# Create Graph
g = drv.Graph()

# Create and load kernel module
mod = SourceModule("
      #define bIdx(N) ((int) blockIdx.N)\n#define tIdx(N) ((int)
      threadIdx.N)\n\nextern "C" __global__ void __launch_bounds__(16)
      doublify(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)] = 2.0 * _in1[4 * (tIdx(x) / 4) + tIdx(x) + -4 * (
      tIdx(x) / 4)];\n  }\n}")

# Get kernel function
doublify = mod.get_function("doublify")

# Initialize input array
a = np.random.randn(4, 4).astype(np.float64)

# Initialize result array
a_doubled = np.empty_like(a)

# Allocate memory on GPU for input array
a_gpu = drv.mem_alloc(a.nbytes)

# Add memcpy node for host to device transfer
memcpy_htod_node = g.add_memcpy_htod_node(a_gpu, a, a.nbytes)

# Add kernel node for array operation
kernel_node = g.add_kernel_node(a_gpu, func=doublify, block=(4, 4, 1),
                                dependencies=[memcpy_htod_node])

# Add memcpy node for device to host transfer
memcpy_dtoh_node = g.add_memcpy_dtoh_node(a_doubled, a_gpu, a.nbytes,
                                [kernel_node, memcpy_htod_node])
```

```
# Instantite execution graph
g_exec = drv.GraphExec(g)

# Launch execution graph on default stream
g_exec.launch()
```

3.2. Loopy

Loopy 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 is a lazy-evaluation programming based Python package that offers a subset of Numpy operations for manipulating multidimensional arrays. This provides the convenience of realzing one-dimensional layout of memory buffer for large scale multidimensional scientific computing workloads (PDE-based numerical methods, deep learning, computational statistics etc.) where the higher dimensional vizualization of data is close to the mathematical notation.

Pytato's IR encodes user defined array computations as a DAG where nodess correspond to array operations and edges representing dependencies between inputs/outputs of these operations. We map the set of nodes provided by Pytato's IR onto the following two node to simplify code generation:

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

```
# 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

# {{{ execute

import pyopencl as cl
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
prg = pt.generate_loopy(result, cl_device=queue.device)
a = np.random.randn(4, 4).astype(np.float64)
_, out = prg(queue, x=x)

# }}}
```

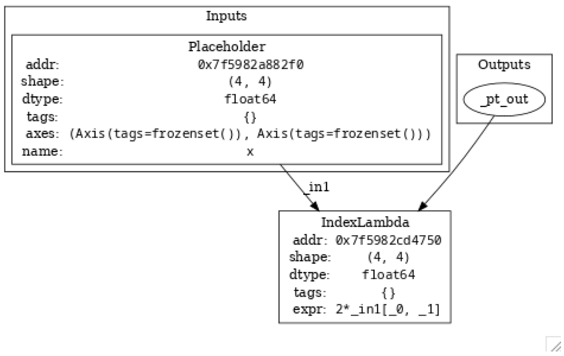

Figure 3. Pytato IR corresponding to doubling operation

4. Array Operations to CudaGraph Transformation

Pytato provides a `pt.compile` decorator which triggers a two-stage code generation process that traces the array program and generates `PyCUDA-CUDAGraph` code.

```
# {{{ 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_indexlambda(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)] = 2.0 * _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_indexlambda'), 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, *, _pt_data):
    _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, _pt_data.gpudata])})
    _pt_exec_g.launch()
    _pt_tmp = {'2a': _pt_result}
    return _pt_tmp

 # }}}
```

4.1. 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. This frontloads the *sink* nodes which helps avoid array recomputations during DAG discovery. Initialize a `pycuda.Graph` object.

Step 2:
for $n \in$ 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 $n \in$ {`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.

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

4.2. Update `CUDAGraphExec`

Algorithm 3 gets executed for every graph launch.

Algorithm 3: Buffer update in `CUDAGraphExec`

for n $\in$ kernel nodes in `pycuda.GraphExec` with temporary buffers **do**
- Replace temporary buffers with allocated/linked buffers from corresponding `PlaceHolder` nodes.

done

5. RESULTS

We evaluate the performance of our framework on three end-to-end DG-FEM operators with real-world applications on NVIDIA Titan V. We evaluate these operators on 3D meshes with tetrahedral cells and evaluate our speedup against `PyOpenCL` which supports sequential stream execution. Table 2. summarizes our experimental parameters.

Equation	**Polynomial Degree**	**No. of mesh elements**
3D Wave	1	1.25×10^5
	2	5.0×10^4
	3	2.5×10^4
	4	1.4×10^4

Table 2. Experimental parameters for DG-FEM operators

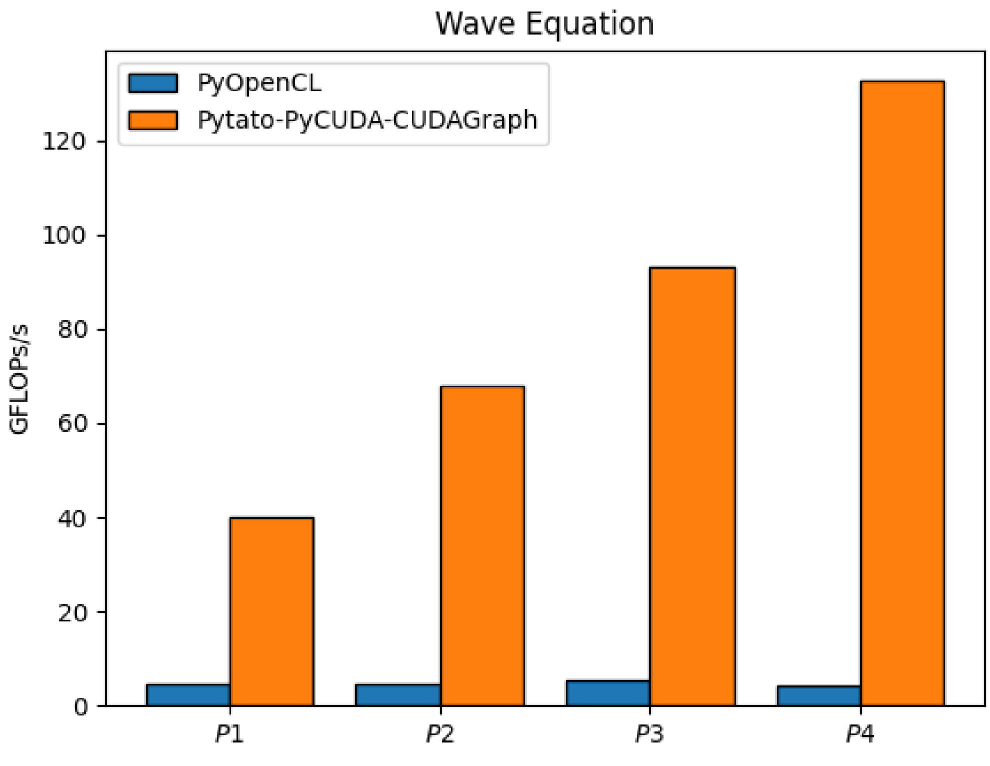

Figure 4. Performance of our framework (`Pytato-PyCUDA-CUDAGraph`) over sequential stream execution (`PyOpenCL`)