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.). These routines are often implemented in interpreted languages like Python which are too unconstrained for performance tuning. 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 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 conclude with some insights on NVIDIA's runtime scheduling algorithm using a set of micro-benchmarks to motivate future performance modelling work.

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 edoing "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.

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 with knl_where python code + Stream/Graph cartoon + overlap picture)

Our system automatically realises this concurrency across array operations using `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.

We implement this system by:

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. OVERVIEW

2.1. 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. Refere 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`

Here's a simple example demonstrating the CUDAGraph functionality:

a) Create a `Graph` through *cuGraphCreate*. Define and load the kernel function using *cuModuleLoadData*/*cuModuleGetFunction* calls baked into `SourceModule` abstraction.

b) Create and allocate memory for Numpy arrays. Transfer the memory to GPU via *cuGraphAddMemcpyNode*.

c) Add a kernel node with the `memcpy_htod_node` as dependency using *cuGraphAddKernelNode*.

d) Transfer the memory back to host via *cuGraphAddMemcpyNode* with `kernel node` and `memcpy_dtoh_node` as dependencies.

e) Instantiate the graph through *cuGraphInstantiate* and execute it through *cuGraphLaunch*.

```
#!/usr/bin/env python

g = drv.Graph() # Create Graph

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

a = np.random.randn(4, 4).astype(np.float64) # Random input array
a_doubled = np.empty_like(a)                 # Empty Result Array
a_gpu = drv.mem_alloc(a.nbytes)              # Allocating input memory

memcpy_htod_node = g.add_memcpy_htod_node(a_gpu, a, a.nbytes) # HtoD

kernel_node = g.add_kernel_node(a_gpu, func=doublify, block=(4, 4, 1),
                                dependencies=[memcpy_htod_node]) #Kernel

memcpy_dtoh_node = g.add_memcpy_dtoh_node(a_doubled, a_gpu, a.nbytes,
                                [kernel_node, memcpy_htod_node]) #DtoH

g_exec = drv.GraphExec(g)                    # Instantiate Graph
g_exec.launch()                              # Execute Graph
```

2.2. 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")])
```

2.3. Pytato

Pytato is a lazy-evaluation programming based Python package that offers a subset of Numpy operations for manipulating multidimensional arrays. This provides ease of convenience in managing 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.

We make use of the following Pytato nodes in our pipeline:

1. *Placeholder*: A named placeholder for an array whose concrete value is supplied during runtime.

2. *IndexLambda*: A canonical representation for capturing array operations where operations are supplied as Pymbolic expressions and operands are stored inside a dictionary which maps strings that are valid Python identifiers onto PlaceHolder objects.

Here's an example capturing the PyCUDA-CUDAGraph workflow shown above.

```
#!/usr/bin/env python

import pytato as pt
import numpy as np
x = pt.make_placeholder(name="x", shape=(4,4), dtype="float64")
result = pt.make_dict_of_named_arrays({"2x": 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)

# }}}
```

3. Array Operations to CudaGraph Transformation

In the context of `CUDAGraphs`, the compilation pipeline is split into the following parts:

1) `Pytato` IR that encodes user defined array compuations as a `DAG` where nodes correspond to array operations and edges represnting dependencies between inputs/outputs of these operations.

2) `Pytato` IR's visitor which in this case is a `PyCUDA-CUDAGraph` mapper onto `Pytato's` canonical representation `IndexLambda`.

3) `PyCUDA-CUDAGraph` Code Generation

Algorithm 1

Step 1: Capture user-defined DAG through Pytato's array interface.

Step 2: Run a topological sort on the graph to facilitate optimal FLOP choice and store the graph as a `DictOfNamedArrays`. Create a corresponding `CUDAGraph` object.

Step 3:

for a ϵ Nodes in `DictOfNamedArrays` **do**

Source Node Type	Target Node Type
`DataWrapper`	`PlaceHolder`
`Roll`	`IndexLambda`
`AxisPermutation`	`IndexLambda`
`IndexBase`	`IndexLambda`
`Reshape`	`IndexLambda`
`Concatenate`	`IndexLambda`
`Einsum`	`IndexLambda`

done

Step 4:

for a ϵ Nodes in `DictOfNamedArrays` **do**

if a == `PlaceHolder`:

→ Link to user provided buffers or generate new buffers via `GPUArrays`.

→ Emit `CgenMapperAccumulator` node tracking the allocated buffer and dependencies.

else if a == `IndexLambda`:

→ 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 with dependencies from child CgenMapperAccumulator nodes going into the
      dependencies field.
    → Emit CgenMapperAccumulator node tracking the result buffer and dependencies.
done

Step 5: Instantiate and cache the executable graph.

Step 6: For every subsequent graph launch the input buffers get updated

for a ϵ Nodes in PlaceHolders do
    → Replace kernel node temporary buffers with buffers from corresponding
      CgenMapperAccumulator nodes using batched_set_kernel_node_arguments
done
```

The generated code can be split into the following parts:

1) *Kernel Creation*: Load cuModules using kernel strings derived from Loopy.

2) *CUDAGraph Building*: Build and cache execution graph by traversing user-defined DAG.

3) *Memory Allocation:* Update execution graph with allocated/linked buffers and launch the graph.

```
#!/usr/bin/env python

_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)
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}")

@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('doublify'), 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)

    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. 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.

5. RESULTS

(Plots using the latest version dg_benchmarks)

(Show speedup by changing height and width for different kernels)