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 has been a lot of effort in productionizing n-d array applications through kernel and loop fusion, very little attention has been paid to leveraging the concurrency across array operations. In this work, we target this concurrency by extending `Pytato`, a lazy-evalulation based array package with `Numpy`-like semantics, to wrap around NVIDIA's `CUDAGraph` API. 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

`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. In addition to releving the programmer from the burden of dealing with low-level details such as prefetching, data transfers, scheduling of tasks, or synchronizations, `CUDAGraphs` also enable a define-once-run-repeatedly execution flow by separating the definition and execution of the graph.

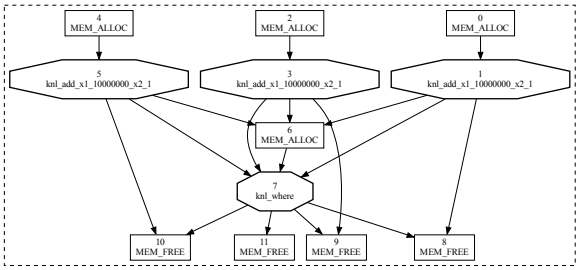

Figure 1.

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. The concurrency available across these array operation nodes offers an opportunity to the runtime scheduler to saturate all of the available execution units.

Our system attemps to realize this concurrency 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.

The roof-line performance of any task based application is influenced by its task types and dependencies among tasks. At runtime, the scheduler knows the (i) the state of different resources, (ii) the set of tasks that are currently processed by all non idle resources, (iii) the set of (independent) tasks whose all dependencies have been solved (iv) the location of all input data of all tasks (v) possibly an estimation of the duration of the each task of each resource and of each communication between each pair of resources and (vi) possible priorities associated to tasks and have been computed offline. We lay out the ground for future performance modelling work by reporting speedups over sequential execution by varying the width and height for different kernels for our DAGs.

2. OVERVIEW

2.1. CUDA Graphs

CUDAGraphs offer a graph-based alternative to the more traditional 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 improve on this approach by allowing the programmer to construct a graph of compute, host and copy operations with arbitary intra- and inter- stream synchronization, to then dispatch the previously displayed operations with a single CPU runtime function. Dispatching a CUDAGraph can be an iterative or periodic operation so GPU-CPU taskets can be implemented as periodic DAGs.

We wrapped the CUDAGraph driver API using PyCUDA which provided a high level Python scripting interface for GPU programming.

Operations	PyCUDA routines
Memory Allocation	add_memalloc_node
Kernel Executton	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
Graph Launch	launch

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

2.2. Loopy

We rely to `Loopy` which is `Python`-based transformation toolkit to generate transformed kernels which are then passed onto `PyCUDA's` run time code generation interface. We make use of the following components in our pipeline:

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.

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

Step 1: `Pytato` IR that encodes user defined array compuations as a `DAG`.

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

Step 3: `PyCUDA-CUDAGraph` Code Generation

Algorithm : DAG Exploration

```
function AddPycudaCudagraphNode(depNode)
  {Returns the Loopy kernel corresponding to an array operation}
  ...
end function

function VisitDependencies(Node)
  while Node.dependencies ≠ ϕ
    for depNode in Node.dependencies
      if depNode.visited = False
        VisitDependencies(depNode)
        depNode.visited = True
  AddPycudaCudagraphNode(depNode)
end function

function TraverseDAG(graph)
  graphSort = TopologicalSort(graph)
  TransformNodestoIndexLambda(graphSort)
  VisitDependencies(graphSort.resultNode)
end function
```

(Explain Algorithm)

```
_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
```

(Explain Generated Code)

3. Related work

(Parallelizing vanilla Python Code)

Castro et give an overview of the current task-based `Python` computing landscape by mentioning several libraries that rely on *decorators*. 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.

(Numerical Libraries)

`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. `JAX`, which is based on Accelarated Linear Algebra Compiler (XLA), optimizes GPU peformance through vectorization/parallelization, automatic differentiation, and `JIT` compilation

(Deep Learning Frameworks)

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 at runtime facilitating the implementation of sophosticated training routines.

(Performance Modeling)

The performance modelling work has been primarilty motivated by `StarPU` and its supportive works. `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. Refer to Augonnet et el for a complete description of different schedulers supported within `StarPU`. The `StarPU` scheduling system also offers a large set of features which include full control over the scheduling policy, support for hybrid platforms and efficient handling of data transfers.

4. RESULTS

(Plots using the latest version dg_benchmarks)

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