

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 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 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** 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 arbitrary 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 serve as drop-in replacements for accelerating 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 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.

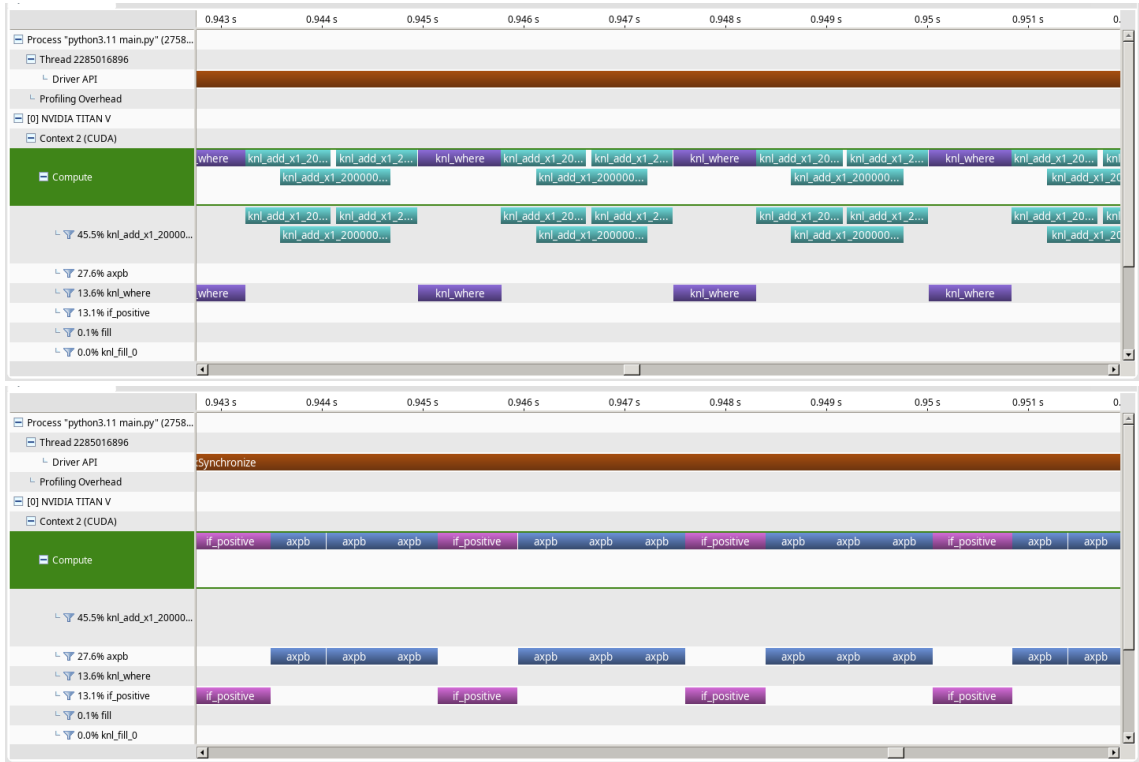


Figure 1. Profiles for CUDAGraph (top) and PyCUDA (bottom) for `where(condition, if, else) + 1`

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 separately from its execution through a custom API.

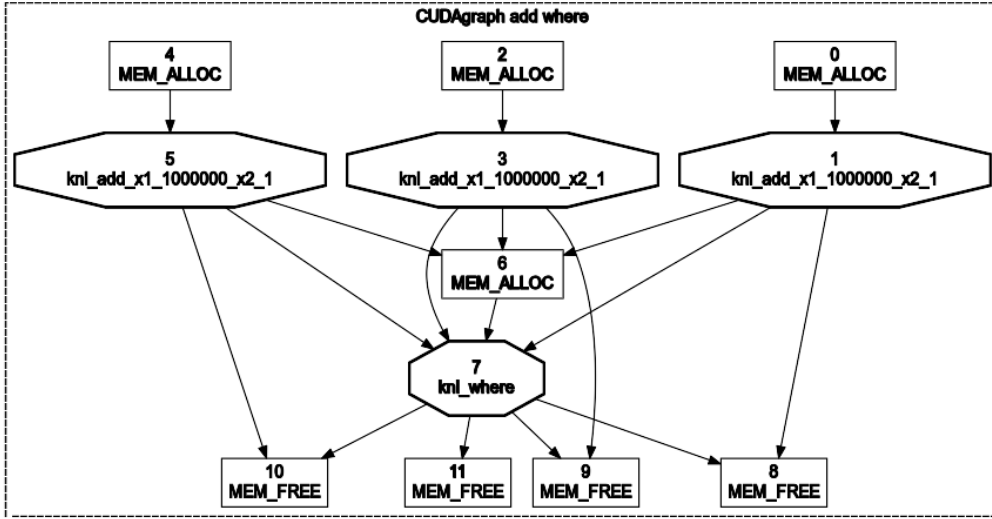


Figure 2. CUDAgraph API generated graph for `where(condition, if, else) + 1`

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 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

We provide an overview of the literature on dataflow based array programming which can be classified roughly according to their choice of task granularity.

Function: Castro[8] gives an overview of the current task-based **Python** computing landscape by mentioning PyCOMPs[17], Pygion[16], PyKoKKos[2] and Legion [4] 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. On a similar note, 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 **Pydrone**, decorated functions are first translated into an intermediate representation and then analyzed by a scheduler which changes the execution graph as each task is finished.

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

Stream: CuPy[13] serves as a drop-in replacement to Numpy and support NVIDIA’s in-house CUDA frameworks such as cuBLAS, cuDNN and cuSPARSE. Julia[5] 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 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. This can lead to multiple DAGs which can potentially trigger recomputations for every graph launch.

Delayed Execution model: JAX[7] optimizes GPU performance by translating *high-level traces* into XLA[15] HLO and then performing vectorization/parallelization and JIT compilation. Deep learning (DL) symbolic mathematical libraries such as TensorFlow[1] and PyTorch[14] 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 the 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 sophisticated training routines.

Kernel: Both StarPU[3] and PARSEC[6] provide excellent support for heterogeneous hardware on distributed systems. Both emphasize GPU contexts, data prefetching and have heterogeneous data-aware scheduling policies based on HEFT[18] heuristic. ParSEC in particular uses lightweight tasks driven by CUDA events.

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 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 asynchronous compute and copy, meaning that CPU cores dispatch commands without waiting for their GPU-side completion: 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 overlapping in time.

CUDAGraphs facilitate 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 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. 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:

[illegible]

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

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

3.2. Loopy

Loopy[9] 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.

```
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[12] is a lazy-evaluation programming based Python package that offers a subset of Numpy operations for manipulating multidimensional arrays. This provides the convenience of realizing 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 visualization of data is close to the mathematical notation.

Pytato offers an IR which encodes user defined array computations as a DAG where nodes 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


```

# 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

# {{{ 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)

# }}}

```

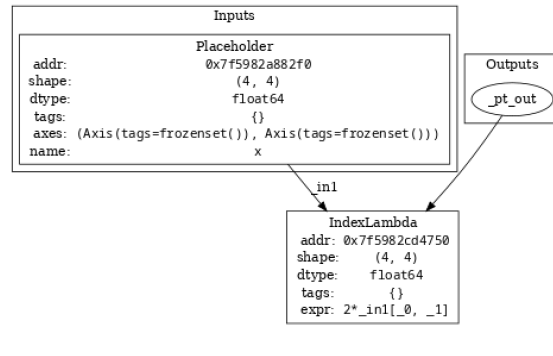


Figure 3. Pytato IR corresponding to doubling operation

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}\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. 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. 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  $n \in \{\text{Placeholder}, \text{DataWrapper}\}$  {
    • PLACEHOLDERMAPPER( $n$ )
    • Link to user provided buffers or generate new buffers via GPUArrays.
    return  $\{n\}$ 
  }
  else {
    • INDEXLAMBDA MAPPER( $n$ )
    • 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 temporary buffer information.
     $n\_deps \leftarrow \{\}$ 
    for  $c \in n$  dependencies sourced from Pytato IR do
       $c\_deps \leftarrow \text{GRAPHTRAVERSE}(c)$ 
       $n\_deps \leftarrow n\_deps \cup c\_deps$ 
    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  $n \in$  kernel nodes in pycuda.GraphExec with temporary buffers do
  • Replace temporary buffers with allocated/linked buffers from corresponding
    Placeholder nodes.
done

```

5. RESULTS

We demonstrate 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[10] which supports sequential stream execution. Table 2. summarizes our experimental parameters.

Equation	Polynomial Degree	No. of mesh elements
<i>3D Wave</i>	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

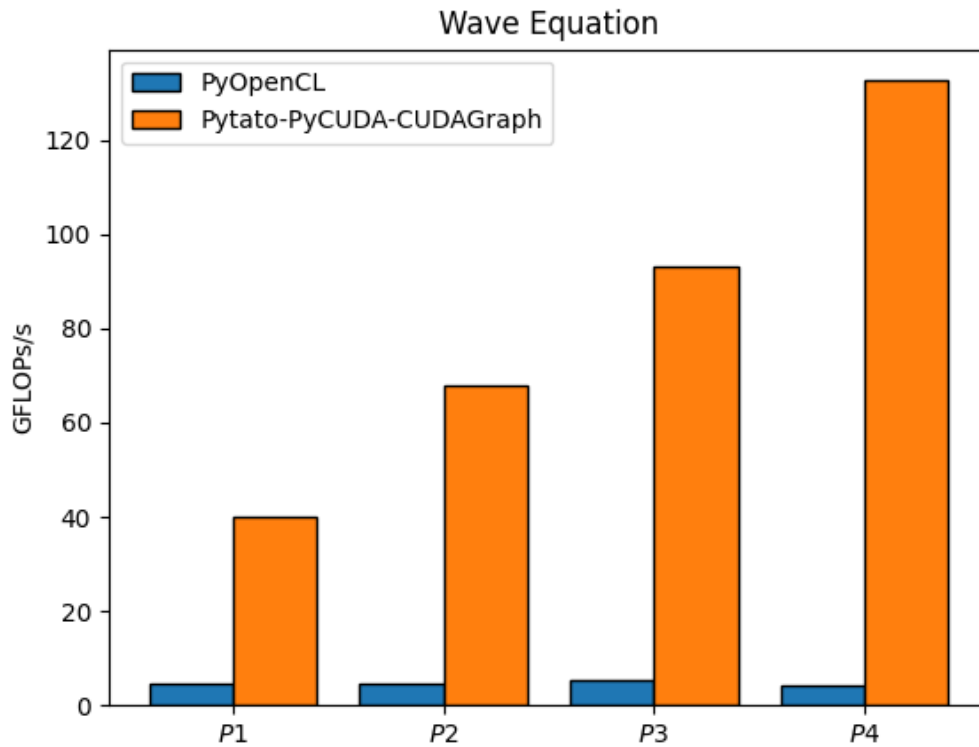


Figure 4. Performance of our framework (Pytato-PyCUDA-CUDAGraph) over sequential stream execution (PyOpenCL)

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 Operators 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] Martín 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. 03 2016.
- [2] Nader Al Awar, Neil Mehta, Steven Zhu, George Biros, and Milos Gligoric. Pykokkos: performance portable kernels in python. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 164–167. 2022.
- [3] 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(2):187–198, 2011.
- [4] Michael Bauer. Legion: programming distributed heterogeneous architectures with logical regions. 2014.
- [5] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: a fast dynamic language for technical computing. 2012.
- [6] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra. Parsec: exploiting heterogeneity to enhance scalability. *Computing in Science Engineering*, 15(6):36–45, 2013.
- [7] 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.
- [8] Oscar Castro, Pierrick Bruneau, Jean-Sébastien Sottet, and Dario Torregrossa. Landscape of high-performance python to develop data science and machine learning applications. 02 2023.
- [9] Andreas Klöckner. Loo.py: transformation-based code generation for GPUs and CPUs. In *Proceedings of ARRAY ‘14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming*. Edinburgh, Scotland., 2014. Association for Computing Machinery.
- [10] 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(3):157–174, 2012.
- [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(3):157–174, 2012.
- [12] Kaushik Kulkarni and Andreas Klöckner. Separating concerns in computational science without domain specific languages. 2023.
- [13] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: a numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. 2017.
- [14] 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. 12 2019.
- [15] Amit Sabne. Xla : compiling machine learning for peak performance. 2020.
- [16] Elliott Slaughter and Alex Aiken. Pygion: flexible, scalable task-based parallelism with python. In *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*. IEEE, nov 2019.
- [17] 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(1):66–82, jul 2016.

- [18] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.