Multi-GPU Project

Irem Naz Celen
MoMA Lab 2024 Summer Internship

Project Description

The goal of this project is to determine the best CUDA-enabled framework to simple Machine Learning algorithms that with options to:

- Manage kernel configurations such as block and grid dimensions.
- To distribute the workload and utilize multiple GPUs.

The project trajectory started with focus on only CUDA and only Cupy implementations and then has expanded.

Initial explorations are omitted for consistency.

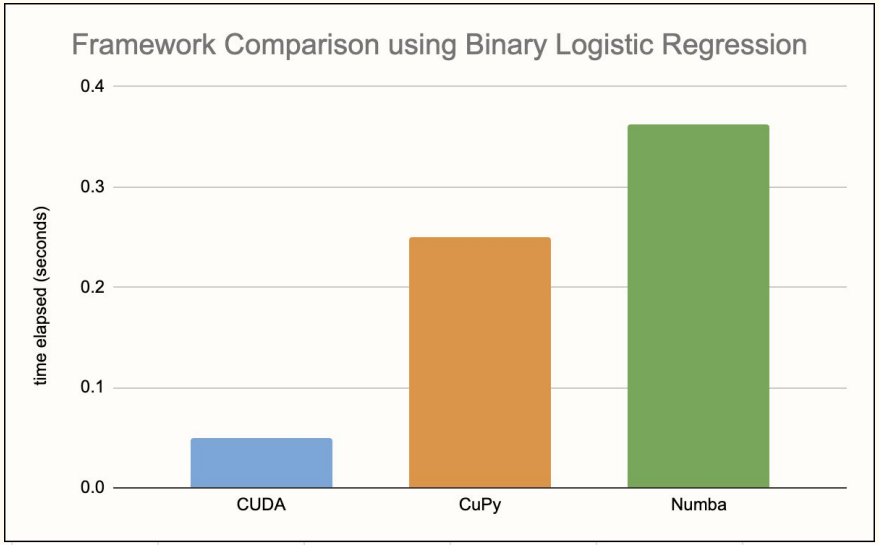
Comparing Frameworks

CUDA-enabled frameworks that allow low level thread management:

1. CUDA
2. CuPy with custom kernels
3. Numba with JIT

With the above set-up the 3 models are compared

for a simple binary logistic regression on single GPU:



Pytorch is also capable of doing low level thread management with custom kernels using custom CUDA kernels and wrappers to link them to python library, which proved to be more complex to establish. Hence CuPy and PyTorch are compared using Binary Logistic Regression with their own functions:

- Cupy taking 0.689 seconds
- PyTorch taking 0.684 seconds

Choosing a Framework

CuPy is used as proposed initially for the rest of the Project, however PyTorch, when implementation difficulties are resolved, can be explored according to needs as a second option.

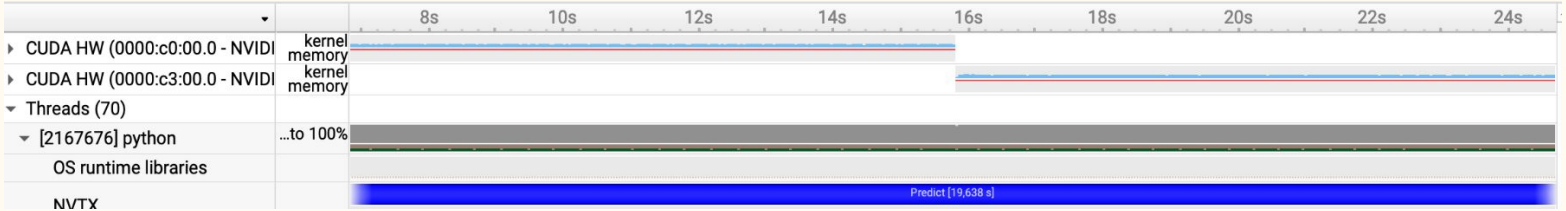
Multi-GPU Setup

Benefit of Multiple GPU setup either comes from equally distributing the input data or the model itself, to be combined later on.

Since simple ML frameworks are more suitable for data parallelization, that is taken the goal as Multi-GPU setup.

Moreover, inside each GPU and between GPUs, mechanisms to allow parallel data transfer and kernel execution must be deployed.

If the overlap is unsuccessful product results in a larger overhead due to deploying other GPUs that all sequentially execute, which was the case for first trials for applying Multiple GPU setup in CuPy with Custom Kernels, as seen below:



Establishing a Multi-GPU Example

Hence first the conversion basics are established by finding a Multiple- GPU example in CUDA, and trying to replicate it using CuPy with Custom Kernels. The following GitHub repo is used for a CUDA example.

The time taken inside profiling from memCopy from Host to Device and back memCopy from Device to Host: **in CUDA 7.6 ms, in CuPy 9 ms.**

Cuda:

CuPy:

Setting Precedents for Multi-GPU Context

- Creating non-blocking Streams under each GPU context
- Using a class structure to note each component that lives inside each GPU, such as the specific setup of data, streams. Ex:

```
class GPUclass:
    def __init__(self):
        # The 2D distance array for each x_test value (row) distances to each x_train value is recorded (column)
        self.d_distances = None
        self.d_Xtrain = None
        # Subset of the larger x_test array sent for this GPU
        self.d_Xtest = None
        self.d_yTrain = None
        self.stream = None
```

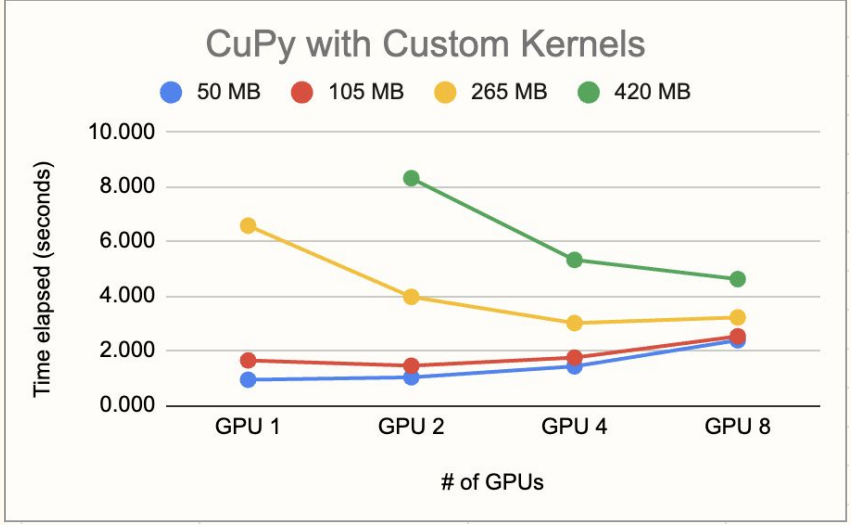
- Using CuPy functions that are asynchronous to initialize/transfer data. Ex: cupy.asarray, cupy.zeros, cupy.random.randint
- Using CuPy functions that are asynchronous to transfer data. Ex: cupy.set, cupy.get
- Changing memory pool of Cupy to Asynchronous or completely disable. Ex: cp.cuda.set_allocator(cp.cuda.MemoryAsyncPool().malloc)
- Carefully handling data distribution between GPUs, as well as their commencement.

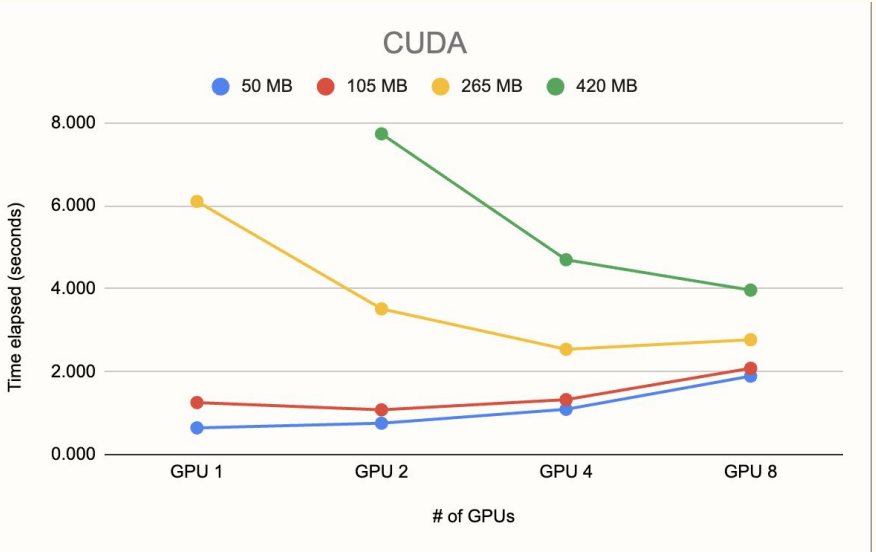
KNN Example Setup

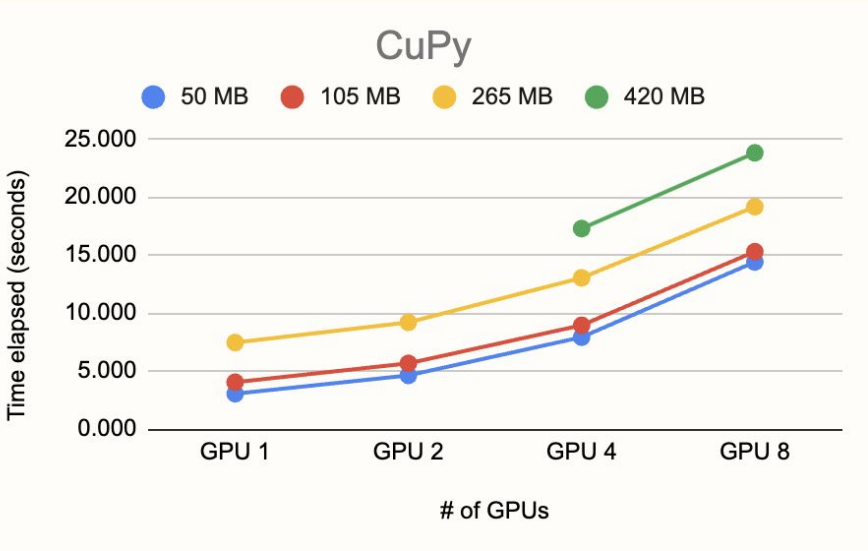
- Dataset: MNIST, total size 210 MB for 32-bit
- Method: KNN, k=3, distance=euclidean
- Versions:
  - CUDA (implemented)
  - CuPy (implemented)
  - CuPy with Custom Kernels (implemented)
  - Scikit-learn library KNN (CPU)
  - cuML KNN library (single GPU)
- Datatype:
  - Float32
  - Int32
- Accuracy across models: 97% - 97.5%
- Test size: 0.25

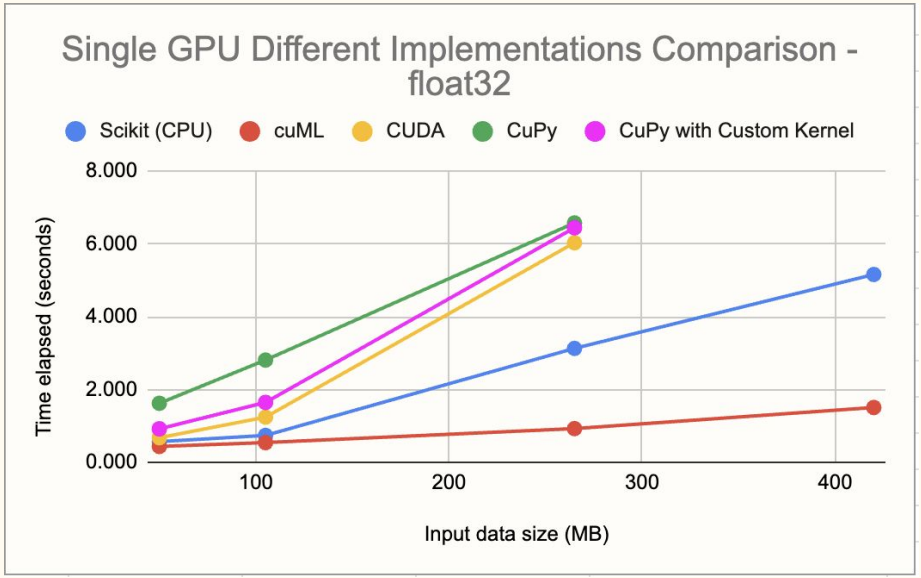
Timed Section:

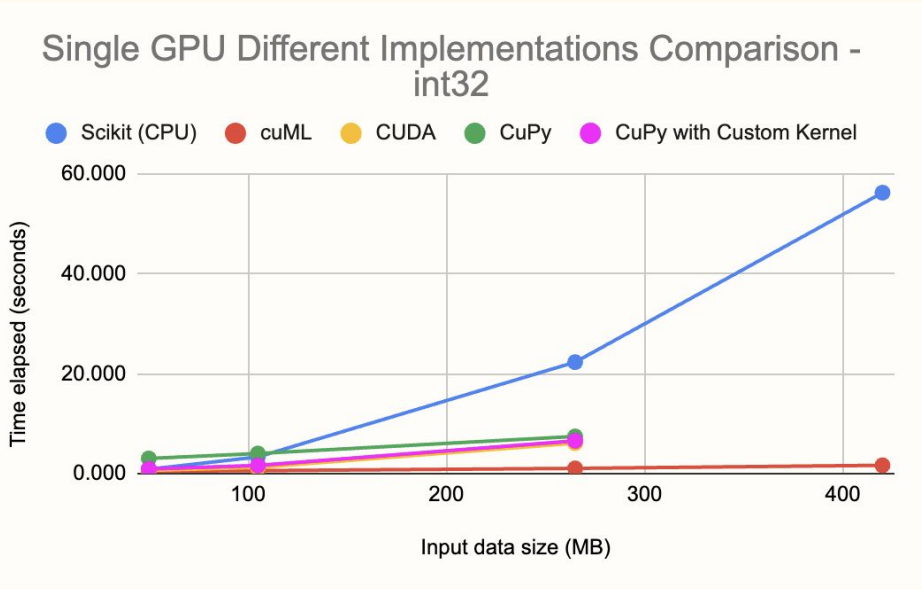
- Constructing GPU classes
- Transferring data into each GPU
- Euclidean Distance Calculation
- Weighted Voting and Label Assignment

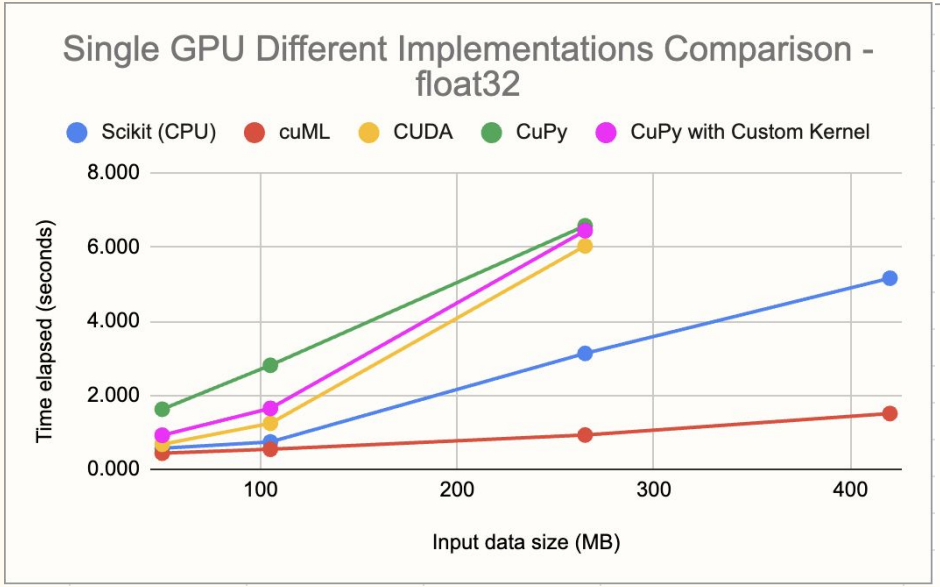

CuPy with Custom Kernels
50 MB
105 MB
265 MB
420 MB
10.000
8.000
6.000
4.000
2.000
0.000
Time elapsed (seconds)
GPU 1
GPU 2
GPU 4
GPU 8
# of GPUs

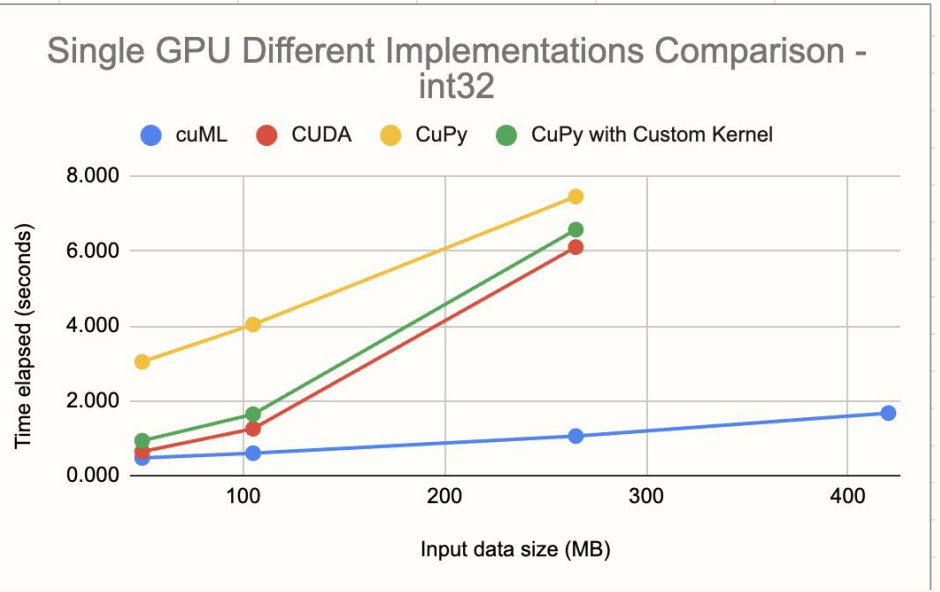

CUDA
50 MB
105 MB
265 MB
420 MB
8.000
6.000
4.000
2.000
0.000
Time elapsed (seconds)
GPU 1
GPU 2
GPU 4
GPU 8
# of GPUs


CuPy
50 MB
105 MB
265 MB
420 MB
25.000
20.000
15.000
10.000
5.000
0.000
Time elapsed (seconds)
GPU 1
GPU 2
GPU 4
GPU 8
# of GPUs

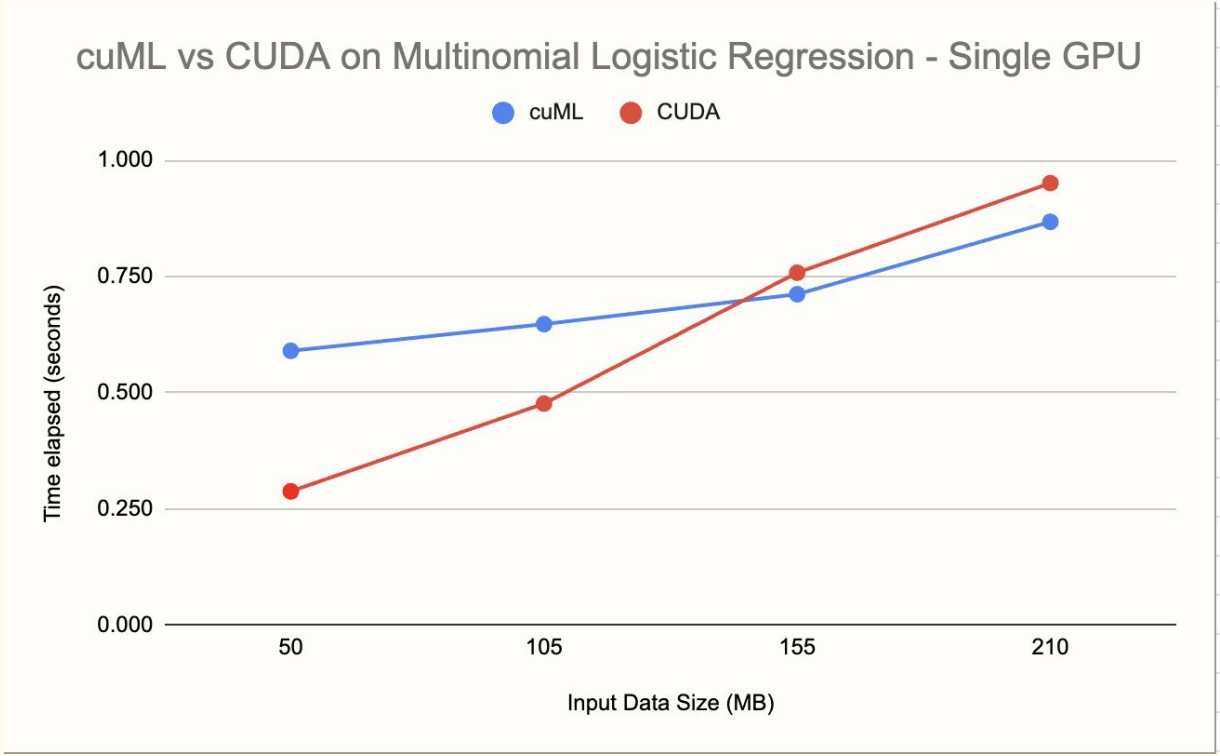

Single GPU Different Implementations Comparison - float32
Scikit (CPU)
cuML
CUDA
CuPy
CuPy with Custom Kernel
8.000
6.000
4.000
2.000
0.000
Time elapsed (seconds)
100
200
300
400
Input data size (MB)


Single GPU Different Implementations Comparison - int32
Scikit (CPU)
cuML
CUDA
CuPy
CuPy with Custom Kernel
60.000
40.000
20.000
0.000
Time elapsed (seconds)
100
200
300
400
Input data size (MB)


Single GPU Different Implementations Comparison - float32
Scikit (CPU)
cuML
CUDA
CuPy
CuPy with Custom Kernel
8.000
6.000
4.000
2.000
0.000
Time elapsed (seconds)
100
200
300
400
Input data size (MB)


Single GPU Different Implementations Comparison - int32
cuML
CUDA
CuPy
CuPy with Custom Kernel
8.000
6.000
4.000
2.000
0.000
Time elapsed (seconds)
100
200
300
400
Input data size (MB)

Exploring Multinomial Logistic Regression - MNIST dataset



Next Steps:

- Apply cuML's multi-GPU property through DASK and compare it to CUDA multi-GPU setup.
- Make CuPy with Custom Kernel enabled for multi-GPU execution and compare it with the above.
- Update GitHub accordingly.