IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 19, 2022, accepted May 15, 2022, date of publication May 18, 2022, date of current version May 23, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3175987

Comparative Study of CUDA GPU Implementations in Python With the Fast Iterative Shrinkage-Thresholding Algorithm for LASSO

YOUNSANG CHO[1], JAEOH KIM[2], AND DONGHYEON YU[1], (Member, IEEE)
[1]Department of Statistics, Inha University, Incheon 22212, South Korea
[2]Department of Data Science, Inha University, Incheon 22212, South Korea

Corresponding authors: Jaeoh Kim (jaeoh.k@inha.ac.kr) and Donghyeon Yu (dyu@inha.ac.kr)

This work was supported in part by Inha University Research Grant, and in part by the National Research Foundation of Korea under Grant NRF-2020R1F1A1A01048127.

ABSTRACT A general-purpose GPU (GPGPU) is employed in a variety of domains, including accelerating the spread of deep natural network models; however, further research into its effective implementation is needed. When using the compute unified device architecture (CUDA), which has recently gained popularity, the situation is analogous to use the GPUs and its memory space. This is due to the lack of a gold standard for selecting the most efficient approach for CUDA GPU parallel computation. Contrarily, as solving the least absolute shrinkage and selection operator (LASSO) regression fully consists of the basic linear algebra operations, the computation using GPGPU is more effective than other models. Additionally, its optimization problem often requires fast and efficient calculations. The purpose of this study is to provide brief introductions to the implementation approaches and numerically compare the computational efficiency of GPU parallel computation with that of the fast iterative shrinkage-thresholding algorithm for LASSO. This study contributes to providing gold standards for the CUDA GPU parallel computation, considering both computational efficiency and ease of implementation. Based on our comparison results, we recommend implementing the CUDA GPU parallel computation using Python, with either a dynamic-link library or PyTorch for the iterative algorithms.

INDEX TERMS Compute unified device architecture, graphics processing unit, fast iterative shrinkage-thresholding algorithm, LASSO, Python.

I. INTRODUCTION

A general-purpose GPU is a graphics processing unit that can be used for general calculations typically conducted on a central processing unit (CPU) rather than graphics-related calculations such as graphics rendering. Modern GPUs have more than thousands of lightweighted cores and provide higher computational capabilities and memory bandwidth at comparable prices than modern CPUs [1]. However, the high computational capability of the GPU is fully utilized only for the calculations that are adequate for single-instruction multiple data (SIMD) parallelism. Basic linear algebra operations like matrix-vector or matrix-matrix multiplications are ideally suited to SIMD parallelism. The CPU is still more powerful than the GPU in terms of task-level parallelism.

The associate editor coordinating the review of this manuscript and approving it for publication was Massimo Cafaro.

Although the efficiency of the GPU parallel computation is restricted to SIMD parallelism, many researchers have developed efficient GPU-parallel algorithms and demonstrated a significant reduction in execution time for tackling target problems in various fields [2]–[6]. More recently, parallel and distributed algorithms have been developed in multi-GPUs or distributed GPU environments [7]–[9], and the popularity of the GPU computation has grown consistently.

To exploit GPU computational capabilities, we must use a specific programming language that handles GPUs and memory space on a GPU device. There are two programming models, the CUDA (Compute Unified Development Architecture) programming model [1], only supported by NVIDIA's GPU devices, and the OpenCL (Open Computing Language) [10], supported by many graphics device vendors, including NVIDIA. Among them, the CUDA programming model has gained more popularity and is widely used

in several applications as the CUDA environment evolves rapidly and NVIDIA provides various efficient libraries for the CUDA programming model, including the `cuDNN` library [11]. This is one of the basic requirements for the state-of-the-art deep neural network programming libraries, such as TensorFlow [12] and PyTorch [13], to accelerate the learning algorithms with the GPU parallel computation. Moreover, the CUDA C programming model provides syntax similar to C/C++ languages [1], allowing researchers who are familiar with C/C++ languages to easily develop the CUDA GPU computation programs.

Several implementation approaches exist to write and use the CUDA GPU computation programs such as implementing an executable program or a dynamic-link library written by the low-level languages (e.g., C and Fortran), and using the existing GPU computation libraries (e.g., PyCUDA [14] and PyTorch [13]). However, ascertaining the most efficient approach for the CUDA GPU parallel computation, considering computational efficiency and ease of implementation, remains uncertain. To answer this question, we first briefly introduce the existing implementation approaches and numerically compare their computational efficiency with the fast iterative shrinkage-thresholding algorithm (FISTA) [15] for the least absolute shrinkage and selection operator (LASSO) regression model [16]. There are two reasons for considering the FISTA for LASSO in this paper. First, the FISTA is applicable to a wide range of applications as it is a general iterative algorithm for minimization problems whose objective function is the sum of differentiable and non-differentiable convex functions. Second, it is well suited for CUDA GPU computation, which is comprised entirely of the basic linear algebraic operations (see more details in Section III). Moreover, considering the convenience of utility, we restricted the implementation approaches to be callable in Python. The main contributions of this study are as follows:

- We provide a brief implementation guideline for each implementation approach, including examples on how to use the implemented functions in Python with examples. These are useful for readers who want to develop their own parallel GPU algorithms running on Python.
- We provide results comparing the computational efficiency of the various implementation approaches. The comparison results reveal that the Python, with the dynamic-link library, and PyTorch, with the basic linear operations, are the most efficient among the other implementation approaches for the FISTA for LASSO.

The remainder of this paper is organized as follows. Section II briefly introduces various implementation approaches, including Python with the dynamic-link library directly using CUDA C [1], PyCUDA [14], Numba [17], TensorFlow [12], and PyTorch [13] with a simple matrix-matrix multiplication example. Section III introduces the FISTA and the FISTA for LASSO as well. In Section IV, we numerically compare the implementation approaches in terms of their computational efficiency and ease of implementation with the FISTA for LASSO. Finally, we conclude this paper with a brief summary and some remarks in Section V.

II. CUDA GPU IMPLEMENTATION APPROACHES

In this section, we provide a brief introduction of the existing implementation approaches and implementation guidelines with a simple matrix-matrix multiplication example provided in Section 3.2.4 in [1]. To be specific, for $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbf{R}^{n\times n}$, we consider a calculation of $\mathbf{C} = \mathbf{AB}$. As we focus on using CUDA GPU computation in Python, we assume that a matrix is stored in row-major order. For a fair comparison, we investigated the implementation of naive functions for the matrix-matrix multiplication without any techniques that are solely relevant for a specific implementation approach, such as Python with a dynamic-link library utilizing the memory hierarchy in CUDA C (e.g., shared memory). Additionally, we considered the implementation with single-precision since most GPU devices have better computation performance with single-precision than with double-precision. Note that the readers need a CUDA-enabled GPU device listed in https://developer.nvidia.com/cuda-gpus and must install CUDA Toolkit from https://developer.nvidia.com/cuda-downloads to proceed with the examples provided in this paper.

A. PYTHON WITH A DYNAMIC-LINK LIBRARY BY CUDA C USING KERNEL FUNCTION

We first introduce how to implement and use the CUDA GPU parallel computation in Python with a dynamic-link library by CUDA C. To distinguish the functions and memory spaces related to CPU and GPU, we use the terms *host* and *device* for CPU and GPU, respectively. A host function, for example, refers to a function running on the CPU and a device memory denotes a memory accessed by the GPU.

The *kernel function*, a special function in the CUDA C programming model, is callable in a host function and running on GPUs. To define a kernel function, we simply add `__global__` declaration specifier to a usual C function having `void` return type. To execute the kernel function, we must specify the thread-block-grid hierarchy, where a grid consists of three-dimensional blocks, a block consists of three-dimensional threads, and a thread is the smallest computing unit in the CUDA C programming model. To support the thread hierarchy, the CUDA C programming model uses a new execution configuration syntax `<<<BlocksPerGrid, ThreadsPerBlock>>>`, where `BlocksPerGrid` and `ThreadsPerBlock` are `int` type (i.e., one-dimensional integer type) or `dim3` type (i.e., three-dimensional unsigned integer vector type) variables. To handle the threads, the CUDA C programming model also provides special index variables such as `threadIdx.x`, `threadIdx.y` and `threadIdx.z`. Similarly, the index variables for the thread blocks are also provided. There are several limitations on the dimensions and sizes of the thread blocks and grids. For the current GPUs, for example, the maximum number of threads

per block is 1024. Further details are provided in Chapter 2 and Appendix K in [1].

We usually write two CUDA C functions to use the CUDA GPU computation in Python: a host function called by Python and a kernel function called by the host function. Only the host function is required while using the cuBLAS library. We will present two example codes by using CUDA C with the user-defined kernel functions and CUDA C with the cuBLAS library for the matrix-matrix multiplication. To compile a CUDA C code, we need to add CUDA C headers `cuda.h` and `cuda_runtime.h`. The header file `cublas_v2.h` should be included for the cuBLAS library. We omit these header files in the provided code examples to focus on the implementation of CUDA C functions. It is also worth noting that the cuBLAS library uses column-major order to store a matrix, whereas Python uses row-major order.

Now, we introduce Python with a dynamic-link library by CUDA C implementation approach using the kernel function. First, we need to write a CUDA C kernel function for a matrix-matrix multiplication as provided in Code 1. As described in Code 1, each thread indexed by `i` and `j` performs the operation $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.

```
1  __global__ void d_MatMul(float *A, float *B,
       float *C, int n) {
2    int i = blockIdx.y*blockDim.y+threadIdx.y;
3    int j = blockIdx.x*blockDim.x+threadIdx.x;
4
5    if ((i < n) && (j < n)) {
6          float value = 0.0;
7      for (int k = 0; k < n; ++k) {
8          value += A[i*n+k]*B[k*n+j];
9      }
10     C[i*n+j] = value;
11   }
12 }
```

Code 1. CUDA C kernel function for the matrix-matrix multiplication.

The host function is required after writing the kernel function in Code 1 in order to call the kernel function `d_MatMul()` and to connect the Python environment. The implemented host function for the matrix-matrix multiplication is provided in Code 2. The declaration specifier `extern "C"` at line 3 in Code 2 is required to make the host function visible in the Python environment. In this host function example code, we used two-dimensional threads and blocks in a grid where each block has 32×32-dimensional threads, which is the maximum number of threads per block on current GPU devices. The number of blocks is calculated by the dimension of threads per block and the dimension of the input matrix in lines 10 and 11.

In the third step, we create a dynamic-link library using the `nvcc` compiler provided by NVIDIA. Before proceeding, we assume that the source code `matmul_kernel.cu` contains the codes provided in Codes 1 and 2 and we work on the Linux platform. To make the dynamic-link library, we executed the following command:
We used the option `-o` of `nvcc` to name the output file `matmul_kernel.so`. The numeric expression `61`

```
1  #define HTD cudaMemcpyHostToDevice
2  #define DTH cudaMemcpyDeviceToHost
3  extern "C"
4  void MatMul(float *A,float *B,float *C,int *N) {
5    float *dA, *dB, *dC;
6    int n = *N;
7    size_t sz_nn = n*n*sizeof(float);
8
9    dim3 TPB(32,32), BPG;
10   unsigned int b_x = (int) ceil((float)n/TPB.x);
11   unsigned int b_y = (int) ceil((float)n/TPB.y);
12   BPG.x = b_x;
13   BPG.y = b_y;
14
15   cudaMalloc(&dA, sz_nn);
16   cudaMalloc(&dB, sz_nn);
17   cudaMalloc(&dC, sz_nn);
18   cudaMemcpy(dA, A, sz_nn, HTD);
19   cudaMemcpy(dB, B, sz_nn, HTD);
20
21   d_MatMul<<<BPG, TPB>>>(dA, dB, dC, n);
22   cudaMemcpy(C, dC, sz_nn, DTH);
23
24   cudaFree(dA);
25   cudaFree(dB);
26   cudaFree(dC);
27 }
```

Code 2. CUDA C host function for the matrix-matrix multiplication.

```
nvcc matmul_kernel.cu -o matmul_kernel.so
   --shared -Xcompiler -fPIC
   -gencode arch=compute_61,code=sm_61
```

Code 3. Example of `nvcc` for making a dynamic-link library.

in `compute_61` and `sm_61` denotes the CUDA compute capability 6.1 of the NVIDIA GeForce GTX 1080 Ti used in this study. The CUDA computing capability varies from the CUDA-enabled GPU devices and readers can find the computing capability of their GPU device from https://developer.nvidia.com/cuda-gpus.

Finally, after completing the three steps above, we can conduct the matrix-matrix multiplication running on the CUDA GPU device in Python. To call the host function `MatMul` in Code 2, we used `ctypes.CDLL()` function that loads a shared library into the Python process. The python example code for Python with a dynamic-link library using the kernel function is provided in Code 4.

The python example code in Code 4 comprises six parts. The first part involves loading the required libraries and functions at lines 1 and 2 in Code 4. To test the implemented `MatMul` function, we defined the matrices `A` and `B` with a random sample from the standard normal distribution in the second part at lines 4–8. The third part involves loading the shared library `matmul_kernel.so` with `ctypes.CDLL()` function and making an alias `matmul_ker()` for `MatMul()` at lines 10 and 11 in Code 4. In the fourth part at lines 13 and 14, we define the arguments and return types with the fundamental data types supported by `ctypes` (see more information in https://docs.python.org/3/library/ctypes.html). In the fifth part, at lines 16–20, we define the pointer variables using the `numpy.ndarray.ctypes.data_as()` function, which provides a memory address adequate for data types in

```python
import numpy as np
from ctypes import *

np.random.seed(2022)
n = 50
A = np.random.randn(n,n).astype(np.float32)
B = np.random.randn(n,n).astype(np.float32)
C = np.zeros_like(A).astype(np.float32)

lib_matmul = CDLL("./matmul_kernel.so")
matmul_ker = lib_matmul.MatMul

matmul_ker.restype = None
matmul_ker.argtypes = (POINTER(c_float),POINTER(
    c_float),POINTER(c_float),POINTER(c_int))

n = A.shape[0]
n_int = c_int(n)
A_ra = A.ctypes.data_as(POINTER(c_float))
B_ra = B.ctypes.data_as(POINTER(c_float))
C_ra = C.ctypes.data_as(POINTER(c_float))

matmul_ker(A_ra, B_ra, C_ra, n_int)
```

Code 4. Python with a dynamic-link library using the kernel function for the matrix-matrix multiplication.

`ctypes.CDLL()`. Note that the kernel function in Code 1 is implemented with the row-major ordered array and the `numpy` also stores a matrix with the row-major order as well. Thus, we do not need to convert the matrix-type data into the vector-type data in row-major order (i.e., `order="C"`). In the sixth part, we conduct the matrix-matrix multiplication on CUDA GPU with `matmul_ker()` function. As the `matmul_ker()` is conducted with the memory addresses of `A`, `B` and `C`, the computed matrix multiplication is stored in the `numpy.ndarray` type matrix `C`.

B. PYTHON WITH A DYNAMIC-LINK LIBRARY BY CUDA C USING cuBLAS LIBRARY

Another implementation approach for Python utilizes a dynamic-link library. If the target operations fully consist of the basic linear algebraic operations, we can conduct the CUDA GPU computation by writing a host function using only the cuBLAS library (i.e., we do not need to write a kernel function). Here, we provide the implementation example for the matrix-matrix multiplication utilizing the cuBLAS library.

As mentioned earlier, we should include `cublas_v2.h` to use the cuBLAS library as written in line 3 of Code 5. In this example code, we use the `cublasSgemm()` function that conducts

$$C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C, \tag{1}$$

where $\alpha, \beta \in \mathbf{R}$, $\mathrm{op}(A) \in \mathbf{R}^{m\times k}$, $\mathrm{op}(B) \in \mathbf{R}^{k\times n}$, $C \in \mathbf{R}^{m\times n}$, $\mathrm{op}(A)$ conducts one of the operations A, A^T, and A^H based on the related argument `transa` in `cublasSgemm()`, and A^T and A^H denote the transpose of A and the Hermitian of A, respectively. From the documentation of the cuBLAS library [19], the function `cublasSgemm()` has the following form:

```cpp
#include<cuda.h>
#include<cuda_runtime.h>
#include<cublas_v2.h>

#define HTD cudaMemcpyHostToDevice
#define DTH cudaMemcpyDeviceToHost

extern "C"
void MatMul_cuBLAS(float *A, float *B, float *C,
    int *N) {
  float *dA, *dB, *dC;
  int n = *N;
  size_t sz_1 = sizeof(float);
  size_t sz_nn = n*n*sizeof(float);
  float One = 1.0, Zero = 0.0;
  float *d_One, *d_Zero;

  cublasStatus_t stat;
  cublasHandle_t handle;

  cudaMalloc(&d_One, sz_1);
  cudaMalloc(&d_Zero, sz_1);
  cudaMalloc(&dA, sz_nn);
  cudaMalloc(&dB, sz_nn);
  cudaMalloc(&dC, sz_nn);

  cudaMemcpy(d_One, &One, sz_1, HTD);
  cudaMemcpy(d_Zero, &Zero, sz_1, HTD);
  cudaMemcpy(dA, A, sz_nn, HTD);
  cudaMemcpy(dB, B, sz_nn, HTD);

  cublasCreate(&handle);
  cublasSetPointerMode(handle,
                  CUBLAS_POINTER_MODE_DEVICE);
  stat = cublasSgemm(handle,CUBLAS_OP_N,
                     CUBLAS_OP_N,n,n,n,d_One,
                     dA,n,dB,n,d_Zero,dC,n);
  cublasDestroy(handle);

  cudaMemcpy(C, dC, sz_nn, DTH);

  cudaFree(dA);
  cudaFree(dB);
  cudaFree(dC);
}
```

Code 5. CUDA C host function for the matrix-matrix multiplication with the cuBLAS library.

```cpp
cublasStatus_t cublasSgemm(handle, transa, transb
    , int m, int n, int k, const float *alpha,
    const float *A, int lda, const float *B, int
    ldb, const float *beta, float *C, int ldc),
```

where `handle` is a `cublasHandle` type object that handles the cuBLAS library context, `transa` and `transb` are `cublasOperation` type object having one of the values `CUBLAS_OP_N` (i.e., $\mathrm{op}(A) = A$), `CUBLAS_OP_T` (i.e., $\mathrm{op}(A) = A^T$), and `CUBLAS_OP_C` (i.e., $\mathrm{op}(A) = A^H$), `m`, `n`, `k`, `alpha`, `beta`, `A`, `B`, and `C` correspond to the variables in the equation (1), and `lda`, `ldb`, and `ldc` are the leading dimensions of `A`, `B`, and `C`, respectively.

To call the functions of the cuBLAS library, we first declare a `cublasHandle` object variable as written in line 18 of Code 5 and create the `cublasHandle` instance as written in line 31 of Code 5 before using the `cuBLAS` functions. Note that the usual cuBLAS functions need the `cublasHandle` object as their argument. After all `cuBLAS` functions are completed, we can release the resources used by the cuBLAS library with the `cublasDestroy()` function as written in line 37 of Code 5. In the cuBLAS

example, we set CUBLAS_POINTER_MODE_DEVICE for the pointer mode of the cuBLAS functions using the cublasSetPointerMode() function.
The CUBLAS_POINTER_MODE_DEVICE only allows device scalar variables as the arguments of the cuBLAS functions. However, the CUBLAS_POINTER_MODE_HOST supports the host scalar variables, which makes the implementation simple and concise. However, we found that the CUBLAS_POINTER_MODE_HOST suffers from the undesired numerical errors in the precision of the calculation when we use the cuBLAS functions with single-precision floating representation. Hence, in this paper, we provide cuBLAS examples using CUBLAS_POINTER_MODE_DEVICE. If readers consider implementing a target algorithm with the double-precision, the CUBLAS_POINTER_MODE_HOST is not problematic in terms of calculation precision.

To create a shared library from the code using the cuBLAS library, we simply add an argument -lcublas in the command provided in Code 3 as shown below.

```
nvcc matmul_cublas.cu -o matmul_cublas.so
  --shared -Xcompiler -fPIC -lcublas
  -gencode arch=compute_61,code=sm_61
```

To conduct the implemented function with the cuBLAS library in Python, several codes should be rewritten in Code 4. First, we need to change the codes in lines 18–22 of Code 4 to the following codes.

```
A_c = A.ravel(order="F")
A_ca = A_c.ctypes.data_as(POINTER(c_float))
B_c = B.ravel(order="F")
B_ca = B_c.ctypes.data_as(POINTER(c_float))
C_c = C.ravel(order="F")
C_ca = C_c.ctypes.data_as(POINTER(c_float))

matmul_cublas(A_ca, B_ca, C_ca, n_int)
```

This modification was required because the cuBLAS library is conducted with the column-major order (i.e., order="F", Fortran-style order). Second, we also needed to change the codes loading the dynamic-link library at lines 10 and 11 in Code 4 to the following codes.

```
lib_matmul = CDLL("./matmul_cublas.so")
matmul_ker = lib_matmul.MatMul_cuBLAS
```

There are several advantages of using the cuBLAS library. First, we do not have to write a kernel function separated from the host function. Second, we do not have to consider about the dimensions and sizes of the thread blocks and grids. The cuBLAS library handles the thread-block-grid hierarchy. Third, although the functions provided by the cuBLAS library are not always faster than the user-defined kernel functions, they are well optimized and rapidly updated. Thus, implementing the CUDA C using the cuBLAS library is a good baseline implementation for the algorithms that consist of many linear algebraic operations.

C. PyCUDA

In this section, we briefly introduce the PyCUDA library [14], which enables the GPU *run-time code generation* (RTCG), also known as *just-in-time (JIT) compilation*, and allows to skip the compilation of the CUDA C source code on the command line by the developer as shown in Code 3. To use the PyCUDA library, the developers first need to install it in their own system. If the developer's system is Linux platform and has the anaconda environment [20], the installation can be done with the following simple command.

```
pip install pycuda
```

Note that the numpy library and the CUDA Toolkit should be installed before PyCUDA is installed. For the developers with a Windows system, we referred to the installation guide provided in PyCUDA Wiki (https://wiki.tiker.net/PyCuda/Installation/Windows/).

To use the PyCUDA library, we simply import the PyCUDA library in Python. Based on the tutorial provided in [21], the initialization of the PyCUDA is suggested by the following codes.

```
1 import pycuda.driver as cuda
2 import pycuda.autoinit
3 from pycuda.compiler import SourceModule
4 from pycuda import gpuarray
```

Code 6. Example code for importing PyCUDA library.

In Code 6, pycuda.driver is a module that contains all attributes and functions related to the CUDA device interface. The code import pycuda.autoinit automatically performs the initialization of the PyCUDA library. The function SourceModule() is used to create a pycuda.driver.Module class from the CUDA C source code by the JIT compilation with the nvcc compiler. For transferring the data from the host memory to the device memory, the PyCUDA provides a simple function pycuda.gpuarray.to_gpu(numpy array) that returns a GPUArray object, which is a numpy.ndarray-like object stored in the device memory.

With the PyCUDA library, we can directly call the CUDA C kernel function by get_function() of pycuda.driver.Module. We can also pass the specification of the thread-block-grid hierarchy defined by the three-dimensional tuples of integers as arguments of the CUDA C kernel function callable in Python. To describe the usage of the PyCUDA, we consider the matrix-matrix multiplication example as in Section II-A and assume that the source code of the kernel function d_MatMul() is stored in matmul_pycuda.cu. Furthermore, we assume that the PyCUDA is initialized with the codes in Code 6 and the data is generated by the codes in lines 4–8 of Code 4. Then, we conduct the matrix-matrix multiplication with CUDA C kernel function by the PyCUDA using the following codes.

The Python example code in Code 7 comprises four parts. The first part is to load CUDA C d_MatMul() function into Python as mm_pycuda() by the JIT compilation using SourceMoudle() and get_function() in lines 2–5 of Code 7. The second part is to transfer the numpy array on the host memory into the device memory using

```
import math
with open("./matmul_pycuda.cu") as f:
    matmul_kernel = f.read()
mm_module = SourceModule(matmul_kernel)
mm_pycuda = mm_module.get_function("d_MatMul")

dA = gpuarray.to_gpu(A.astype(np.float32))
dB = gpuarray.to_gpu(B.astype(np.float32))
dC = gpuarray.to_gpu(C.astype(np.float32))
TPB = (32, 32, 1)
block = math.ceil(n/32)
BPG = (block, block, 1)
mm_pycuda(dA,dB,dC, np.int32(n), block = TPB,
    grid = BPG)
```

Code 7. Example of PyCUDA for the matrix-matrix multiplication.

`gpuarray.to_gpu()` function in lines 7–9 of Code 7. The third part is to specify the thread-block-grid hierarchy with `TPB` (thread per block) and `BPG` (block per grid) in lines 10–12 of Code 7. The final part is to call the `mm_pycuda` function at lines 13 of Code 7, where `block` and `grid` are the arguments for the thread dimension per block and the block dimension per grid, respectively. Note that the arguments `block` and `grid` should be three-dimensional tuple of integers. The `math.ceil()` function at line 11 of Code 7 is used to obtain a standard Python integer value.

PyCUDA offers several advantages. First, as provided in Code 7, we can directly call the CUDA C kernel function without writing a CUDA C host function that calls the CUDA C kernel function as in Code 2. Thus, we do not need to consider the host and device memory-related functions and `dim3` type. Second, we can skip the command line compilation as in Code 3. However, there is a limitation in using PyCUDA. The cuBLAS library is not available with the PyCUDA library. This issue is reported in the frequently asked questions in PyCUDA Wiki (https://wiki.tiker.net/PyCuda/FrequentlyAskedQuestions/).

D. NUMBA

In this section, we briefly introduce the Numba library [17], which provides the JIT compilation using low level virtual machine (LLVM), that translates a general Python function into a machine code. Although the Numba and PyCUDA provide the JIT compilation, the former is more versatile than the latter. First, the Numba library supports the JIT compilation for both Python functions running on CPUs and GPUs by `jit` and `cuda.jit` decorators, respectively. Second, the Numba library only requires a general Python function to generate the optimized machine code, whereas the PyCUDA needs a CUDA C kernel function, as in Code 1. The Numba library can easily be installed into the developer's system by the following this simple command.

```
pip install numba
```

If the developer's system has the anaconda environment [20], the installation can be performed with the `conda` command as follows.

```
conda install numba
```

Note that the CUDA Toolkit is required to use CUDA GPU computation via the Numba library.

Even though the Numba library provides various options for Python JIT compilation, we focus on the CUDA GPU computation via `cuda.jit` decorator in this study. To use the decorator `cuda.jit`, we first import the `cuda` module from `numba` library as follows.

```
from numba import cuda
```

With the `cuda.jit` decorator, we implement a CUDA C kernel function easily with a usual Python syntax. The following Code 8 provides an example of the matrix-matrix multiplication using the `cuda.jit` decorator, which is from the Numba documentation (http://numba.pydata.org/numba-doc/latest/cuda/examples.html).

```
import numpy as np
import math
from numba import cuda

@cuda.jit
def matmul_numba(A,B,C):
    i,j = cuda.grid(2)
    if i < C.shape[0] and j <C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i,k] * B[k,j]
        C[i,j] = tmp

np.random.seed(2022)
n = 50
A = np.random.randn(n,n).astype(np.float32)
B = np.random.randn(n,n).astype(np.float32)
C = np.zeros_like(A).astype(np.float32)

TPB = (32,32,1)
block = math.ceil(n/32)
BPG = (block, block,1)

matmul_numba[BPG,TPB](A,B,C)
```

Code 8. Example of numba for the matrix-matrix multiplication.

The function `cuda.grid(ndim)` returns the thread position in the thread-block-grid hierarchy, where `ndim` corresponds to the dimension of the thread structure declared when instantiating the kernel function. For example, the code in line 7 of Code 8 is equivalent to the following expression.

```
i = cuda.blockIdx.x * cuda.blockDim.x + cuda.
    threadIdx.x
j = cuda.blockIdx.y * cuda.blockDim.y + cuda.
    threadIdx.y
```

When invoking the kernel function, it requires the specification of the thread-block-grid hierarchy with `[blocks per grid, threads per block]` between the kernel function name and its arguments as used in line 24 of Code 8. The `cuda.jit` uses the lazy compilation so that the data types of the arguments are automatically recognized when the function is invoked.

E. TENSORFLOW

This section introduces the implementation approach for CUDA GPU computation using TensorFlow [12].

TensorFlow is one of the main open-source platforms for implementing deep neural network models and providing various functions that support the CUDA GPU parallel computation, including the basic linear algebraic operations. Although we focus on the implementation of the matrix-matrix multiplication on the CUDA GPU device in this brief introduction, TensorFlow is flexible and applicable for tackling common optimization problems. We provide an implementation example for the FISTA in Section IV.

TensorFlow can easily be installed using Python's `pip` package with the following command.

```
pip install tensorflow
```

It is also possible to build it from the source codes of TensorFlow. Additional information is provided in the TensorFlow installation guide (https://www.tensorflow.org/install). To use TensorFlow, we simply need to import the `tensorflow` library into the Python process using the following Python code.

```
import tensorflow as tf
```

Note that the alias `tf` is commonly used in the TensorFlow community when importing the `tensorflow` library.

To conduct the matrix-matrix multiplication on a CUDA GPU device, we provide an example of TensorFlow in Code 9. Here, we assume that the codes at line 14–18 in Code 8 are executed before running the codes in Code 9

```
1 with tf.device("/GPU:0"):
2     dA = tf.constant(A,dtype=tf.float32)
3     dB = tf.constant(B,dtype=tf.float32)
4     dC = tf.matmul(dA, dB)
5 C = dC.numpy()
```

Code 9. Example of TensorFlow for matrix-matrix multiplication.

In Code 9, we use `with tf.device("/GPU:0")` to manually assign the CUDA parallel computation on the GPU device whose device ID is `"/GPU:0"`. The device IDs can be found by running the following code.

```
tf.config.list_physical_devices()
```

Although we use the statement `with tf.device()` to explicitly indicate the chosen GPU device in Code 9, TensorFlow prioritizes the GPU device by default when it is available. TensorFlow automatically uses the GPU device when variables are defined as `tensorflow.Tensor` object in lines 2 and 3 of Code 9. The code `C = dC.numpy()` in line 5 of Code 9 is not necessary in general; however, we use the code to explicitly conduct the memory transfer from the device memory to the host memory for a fair comparison. Note that TensorFlow automatically handles memory transfer between the host and device memories. For example, the users can directly use `tf.matmul()` function without the explicit memory transfer for the multiplication of two matrices where one is stored in the device memory and the other is stored in the host memory.

F. PyTorch

This section introduces the implementation approach for CUDA GPU computation using PyTorch [13]. PyTorch is a Python library based on Torch library [22] in Lua programming language and is also one of the main open source platforms for learning deep neural network models. Similar to TensorFlow, PyTorch supports the CUDA GPU parallel computation and is flexible and applicable to solve general optimization problems. We provide an implementation example for the FISTA in Section IV as well.

PyTorch can be installed with either Python's `pip` package or Anaconda environment with the following commands.

```
pip install torch torchvision torchaudio
conda install pytorch torchvision torchaudio -c
    pytorch
```

The `torchvision` and `torchaudio` libraries are additional libraries for the computer vision and the audio and signal processing using PyTorch, respectively. It is also possible to build from the source codes of PyTorch. Additional information is provided in the PyTorch installation from the source code documentation (https://github.com/pytorch/pytorch#from-source). To use PyTorch, we simply have to import the `torch` library into Python process using the following Python code.

```
import torch
```

To proceed with the matrix-matrix multiplication by PyTorch on a CUDA GPU device, we need to run the following codes in Code 10. Similar to the case of TensorFlow, we assume that the codes in lines 14–18 of Code 8 are executed before running the codes in Code 10

```
1 dA = torch.tensor(A).to("cuda:0")
2 dB = torch.tensor(B).to("cuda:0")
3 dC = torch.matmul(dA,dB)
4 C = dC.to("cpu")
```

Code 10. Example of PyTorch for matrix-matrix multiplication.

In Code 10, we create three tensor objects with `torch.tensor()` to store three matrices and transfer these tensors into the device memory with the function `torch.tensor.to(device)`, where the `device` denotes the host memory if `device="cpu"` and the device memory if `device="cuda:0"`. We can explicitly specify the target device with the `index` in the name `"cuda:index"` if the system has multi-GPU devices. PyTorch only supports the explicit memory transfer between the host and device memory transfer unlike TensorFlow. For example, this code `torch.matmul(dA,C)` creates a runtime error that the function expects the object of device type `cuda` but the second argument has the device type `cpu`.

G. NUMERICAL COMPARISON FOR MATRIX-MATRIX MULTIPLICATION

We have explored the existing implementation approaches for the CUDA GPU parallel computation in Python. TensorFlow

and PyTorch are easy to implement because of their simplicity. The implementation with the Numba library is also easier than other implementation approaches using the low-level language syntax as the Numba library only requires Python-syntax to implement the CUDA kernel function. However, the implementation approach that offers the best computational efficiency for CUDA GPU computation remains uncertain.

In this section, we numerically investigate the computational efficiency of the implementation approaches with the matrix-matrix multiplication. The matrix-matrix multiplication is fairly simple, and then this comparison is still restricted to cover the comparison of the CUDA C implementation for general problems. Section IV investigates the computational efficiency of FISTA to deal with more general problems. To measure the computational efficiency of the implementation approaches, we use the execution time in seconds for each implementation approach on a workstation (Intel(R) Xeon(R) W-2175 CPU (base: 2.50 GHz, max-turbo: 4.30 GHz) and 128 GB RAM) with NVIDIA GeForce GTX 1080 Ti. We consider the matrix-matrix multiplication of two square matrices whose dimensions are denoted by n and $n = 1000, 1500, 2000, 2500, 3000$. Additionally, the CUDA GPU device had different computing powers for single-precision and double-precision operations. To address this difference, we have measured the execution times for both single-precision and double-precision operations. Since the execution time could vary from the resources used by the operating system, we repeat the matrix-matrix multiplication 11 times and average the execution times from the second to the last repetitions (i.e., average of 10 execution times). The first execution time is affected by the preparation procedures of the implementation approaches. Here, we measure the averages of the preparation time by the difference between the first execution time and the average of the execution times from the second to the last repetitions. As a result, the Python with a dynamic-link library using the kernel functions and cuBLAS library need 0.1190 and 0.2637 seconds for the preparation of the CUDA computation, respectively. The Numba, TensorFlow, and PyTorch libraries needed 0.4006, 0.5350, and 1.9513 seconds, respectively. However, PyCUDA has the first execution time similar to the other repetitions. This observation might be from the initialization of the PyCUDA library by `import pycuda.autoinit`. PyCUDA seems to initialize the CUDA context when the `pycuda.autoinit` is imported.

Figures 1 and 2 depict the averages of the execution times of the implementation approaches with single-precision and double-precision, respectively. We also report the average computation times in Table 1. The standard errors of the average computation times for all dimensions are less than 0.001. We omit the standard errors in Table 1.

From the comparison of the computational efficiency for the matrix-matrix multiplication, we observe several patterns. First, PyTorch is always faster than TensorFlow for both single-precision and double-precision even if PyTorch needs more preparation time than TensorFlow. From the documentation of Tensorflow and PyTorch, we found that the matrix-matrix multiplication of PyTorch uses the cuBLAS library and that of TensorFlow uses its own function.

Second, using Python, with a dynamic-link library using the cuBLAS library (DLL-cuBLAS), is faster than using the CUDA C kernel function (DLL-Kernel) for all cases except $n = 1000$ with double-precision. Especially, DLL-cuBLAS is much faster than DLL-Kernel with single-precision. Based on this observation, the `cuBLAS` library is found suitable for the implementation of Python with a dynamic-link library than the user-defined kernel function if the target problem is adequate with single-precision calculation. However, the difference between two implementations with double-precision was only 0.0581 for 3000×3000-dimensional matrix-matrix multiplication. Additionally, there remain several techniques to accelerate the implementation using the user-defined kernel function. For example, we can accelerate the user-defined kernel function by considering the memory hierarchy of the CUDA C programming model. From these points of view, we recommend choosing the implementation approach, between using the user-defined kernel functions and the cuBLAS library, based on the familiarity of the developer when the target problem requires double-precision calculation.

Third, there is no significant difference in the computational efficiency between PyCUDA and DLL-Kernel for single-precision operations; however, PyCUDA becomes more efficient than DLL-Kernel for operations requiring double-precision. When we need to solve a problem with double-precision, PyCUDA is a better option for CUDA GPU implementation than DLL-Kernel. For double-precision, PyCUDA is one of the best implementation approaches in terms of the computational efficiency.

Finally, the Numba library is better for ease of implementation but worse for computational efficiency than Python with a dynamic-link libraries (DLL-Kernel and DLL-cuBLAS) and PyCUDA. Furthermore, the Numba library is also slower than TensorFlow and PyTorch that provide simpler implementations than the Numba library. Further investigation reveals that this inefficiency is caused by the implicit memory transfer between the host and device memories. The comparison of the matrix-matrix multiplication is done with the naive implementation of CUDA C without consideration of the memory hierarchy of the CUDA GPU device for fair comparison and simplicity of description. The readers can accelerate the matrix-matrix multiplication by using the shared memory of the CUDA GPU device. We refer to the example using the shared memory for the Numba library provided in the Numba documentation (http://numba.pydata.org/numba-doc/latest/cuda/examples.html).

In summary, considering both the ease of implementation and the computational efficiency, PyTorch is the best implementation approach for the matrix-matrix multiplication. However, if the dimension of target problem is larger than 3000 and the computational efficiency is more important than ease of implementation, the Python with a dynamic-link

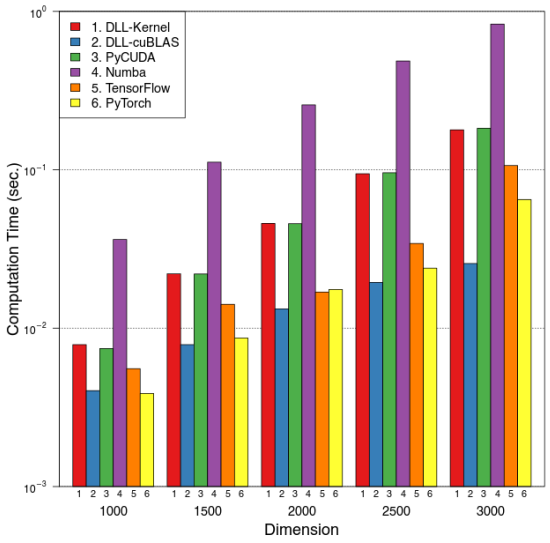

FIGURE 1. Average computation times of the matrix-matrix multiplication with single-precision.

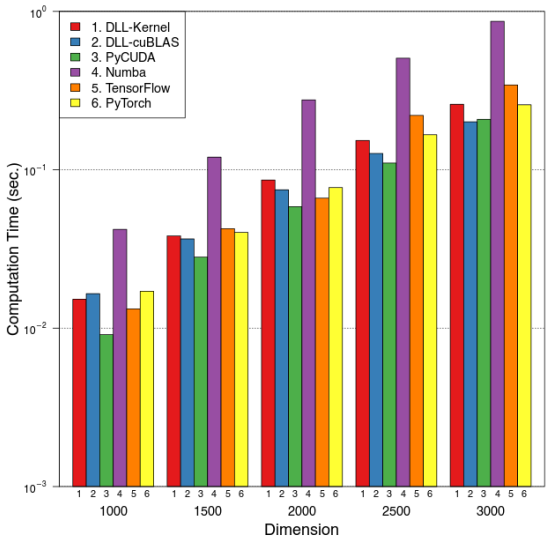

FIGURE 2. Average computation times of the matrix-matrix multiplication with double-precision.

library using the cuBLAS library emerges as the best option for carrying out the matrix-matrix multiplication with single-precision representation.

III. FAST ITERATIVE SHRINKAGE-THRESHOLDING ALGORITHM

The previous section demonstrated an implementation guideline for matrix-matrix multiplications as a suitable example of basic linear operations, as well as a comparison result for computational efficiency. As previously mentioned, the existing implementation approaches are adequate for common optimization problems. We now explore an optimization problem that exploits a general iterative algorithm for minimization problems and fully consists of the basic linear algebraic operations. In this section, we briefly introduce the FISTA and LASSO regression models, and provide the details of the FISTA for LASSO as well.

A. FISTA

The FISTA [15] considers the minimization of the following problem: for $\mathbf{x} \in \mathbf{R}^p$,

$$\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} f(\mathbf{x}) + g(\mathbf{x}), \tag{2}$$

where $f(\mathbf{x})$ is a differentiable convex function of $\mathbf{x}$ and $g(\mathbf{x})$ is convex but possibly non-differentiable function of $\mathbf{x}$. The FISTA, adopting Nesterov's acceleration [24], is an accelerated version of the iterative shrinkage-thresholding algorithm (ISTA) [23].

To describe the FISTA, consider a quadratic approximation $Q_L(\mathbf{x}|\mathbf{y})$ of $f(\mathbf{x})+g(\mathbf{x})$ at a given point $\mathbf{y}$ with $L > 0$ as follows:

$$Q_L(\mathbf{x}|\mathbf{y}) = f(\mathbf{y}) + \nabla f(\mathbf{y})^T(\mathbf{x}-\mathbf{y}) + \frac{L}{2}\|\mathbf{x}-\mathbf{y}\|_2^2 + g(\mathbf{x}), \tag{3}$$

where $\|\mathbf{x}\|_2^2 = \sum_{j=1}^{p} x_j^2$. Furthermore, it is easy to demonstrate that the minimizer of (3) depending on $\mathbf{y}$ is equivalent to the solution of the following problem:

$$q_L(\mathbf{y}) = \underset{\mathbf{x}}{\operatorname{argmin}} \left\{ g(\mathbf{x}) + \frac{L}{2}\left\|\mathbf{x} - \left(\mathbf{y} - L^{-1}\nabla f(\mathbf{y})\right)\right\|_2^2 \right\}. \tag{4}$$

The ISTA algorithm iterates $\mathbf{x}_k = q_L(\mathbf{x}_{k-1})$ with an initial point $\mathbf{x}_0$ and a constant step size L^{-1} with a Lipschitz constant L of ∇f. Note that the backtracking procedure is adopted for choosing step size L^{-1} by repeatedly checking the inequality (5) with $L_k = L_{k-1}\eta^{i_k}$ for $\eta > 1$ and $i_k \geq 0$ even when the Lipschitz constant L of ∇f is not available or it has enormous computational cost. The backtracking procedure finds a value L that satisfies the following inequality:

$$f(q_L(\mathbf{x}_{\text{cur}})) + g(q_L(\mathbf{x}_{\text{cur}})) \leq Q_L(q_L(\mathbf{x}_{\text{cur}})|\mathbf{x}_{\text{cur}}), \tag{5}$$

where $\mathbf{x}_{\text{cur}}$ is the iterative solution at the current iteration. After finding L in (5), the next iterative solution is defined as

$$\mathbf{x}_{\text{next}} = q_L(\mathbf{x}_{\text{cur}}). \tag{6}$$

By Theorem 3.1 in Beck and Teboulle [15], the rate of convergence to the optimal function value is no worse than $O(1/k)$, where k is the number of proceeded iterations.

Motivated by Nesterov's acceleration [24], the FISTA additionally considers a sequence of p-dimensional vectors $(\mathbf{z}_k)_{k\geq 1}$ and a sequence of scalars $(t_k)_{k\geq 1}$ such that for $k \geq 1$,

$$\begin{aligned} t_{k+1} &= \frac{1+\sqrt{1+4t_k^2}}{2}, \\ \mathbf{z}_{k+1} &= \mathbf{x}_k + \left(\frac{t_k - 1}{t_{k+1}}\right)(\mathbf{x}_k - \mathbf{x}_{k-1}), \end{aligned} \tag{7}$$

where $t_1 = 1$ and $\mathbf{z}_1 = \mathbf{x}_0$. The difference between the FISTA and the ISTA is that the next iterative solution $\mathbf{x}_{k+1}$ is updated by $\mathbf{x}_{k+1} = q_L(\mathbf{z}_{k+1})$ instead of $\mathbf{x}_{k+1} = q_L(\mathbf{x}_k)$ at the kth iteration, where L is either a Lipschitz constant of

TABLE 1. Summary of average computation times for the matrix-matrix multiplication.

Precision	n	DLL-Kernel	DLL-cuBLAS	PyCuda	Numba	TensorFlow	PyTorch
Single	1000	0.0079	0.0040	0.0074	0.0363	0.0055	0.0039
	1500	0.0221	0.0079	0.0220	0.1117	0.0141	0.0087
	2000	0.0458	0.0132	0.0457	0.2568	0.0169	0.0175
	2500	0.0944	0.0194	0.0958	0.4860	0.0343	0.0239
	3000	0.1787	0.0256	0.1828	0.8314	0.1064	0.0649
Double	1000	0.0152	0.0165	0.0091	0.0420	0.0132	0.0171
	1500	0.0383	0.0366	0.0281	0.1202	0.0425	0.0402
	2000	0.0862	0.0747	0.0585	0.2763	0.0662	0.0772
	2500	0.1531	0.1267	0.1103	0.5063	0.2204	0.1664
	3000	0.2589	0.2008	0.2082	0.8651	0.3430	0.2572

∇f or a value satisfying (5). By Theorem 4.4 in Beck and Teboulle [15], the rate of convergence to the optimal function value is no worse than $O(1/k^2)$, where k is the number of proceeded iterations.

B. LASSO REGRESSION MODEL

The LASSO regression model [16] is one of the most popular penalized linear regression models. Especially, the LASSO regression model is mostly used for high-dimensional and low sample size (HDLSS) data for both the parameter estimation and the variable selection. To be more specific, let $\mathbf{y} \in \mathbf{R}^n$ and $\mathbf{X} \in \mathbf{R}^{n\times p}$ be a response vector and a design matrix, respectively. The LASSO regression model then considers the following minimization problem:

$$\hat{\boldsymbol{\beta}}(\lambda) = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \frac{1}{2}\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_1, \tag{8}$$

where $\|\boldsymbol{\beta}\|_1 = \sum_{j=1}^{p} |\beta_j|$ is the ℓ_1-norm of a vector $\boldsymbol{\beta} \in \mathbf{R}^p$.

Many algorithms have been developed to obtain the optimal solution of the LASSO regression model. The original paper on LASSO regression [16] considered quadratic programming with linear constraints. Wu and Lange [25] propose the coordinate descent algorithm that is efficient in both computation and memory loads. Furthermore, there are solution-path algorithms that provide whole solution path along tuning parameter λ. Efron *et al.* [26] propose the least angle regression (LARS) algorithm for LASSO regression and Tibshirani and Taylor [27] propose the solution path algorithm for the generalized lasso regression with a penalty term $\lambda\|D\boldsymbol{\beta}\|_1$ instead of $\lambda\|\boldsymbol{\beta}\|_1$, where the definition of a matrix D determines a target problem. For example, the generalized lasso becomes the original lasso problem when $D = I_p$, where I_p is p-dimensional identity matrix. However, theses algorithms are usually conducted in *a sequential manner* that is not adequate for CUDA GPU parallel computation. Computation scalability has become a challenging task for LASSO, which deals with HDLSS as the quantity of data in application domains increases exponentially. Parallel computing techniques can be a good solution for fulfilling the demand for these practical uses. Therefore, we consider applying the FISTA to obtain the solution of the LASSO regression in this study because it fully consists of the basic linear algebraic operations that are well suited for CUDA GPU parallel computation.

C. FISTA FOR LASSO REGRESSION

In this section, we provide details of the FISTA for LASSO. From the descriptions in Sections III-A and III-B, we first define the functions $f(\boldsymbol{\beta})$ and $g(\boldsymbol{\beta})$ in the FISTA as follows:

$$f(\boldsymbol{\beta}) = \frac{1}{2}\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2, \quad g(\boldsymbol{\beta}) = \lambda\|\beta\|_1.$$

Then, the local quadratic approximation $Q(\boldsymbol{\beta}|\boldsymbol{\beta}')$ of the LASSO objective function at a given point $\boldsymbol{\beta}'$ with $L > 0$ is defined as

$$\begin{aligned} Q(\boldsymbol{\beta}|\boldsymbol{\beta}') = &\frac{1}{2}\|\mathbf{r}(\boldsymbol{\beta}')\|_2^2 - \mathbf{r}(\boldsymbol{\beta}')^T\mathbf{X}\mathbf{d}(\boldsymbol{\beta}, \boldsymbol{\beta}') \\ &+ \frac{L}{2}\|\mathbf{d}(\boldsymbol{\beta}, \boldsymbol{\beta}')\|_2^2 + \lambda\|\boldsymbol{\beta}\|_1, \end{aligned} \tag{9}$$

where $\mathbf{r}(\boldsymbol{\beta}') = \mathbf{y} - \mathbf{X}\boldsymbol{\beta}'$ and $\mathbf{d}(\boldsymbol{\beta}, \boldsymbol{\beta}') = \boldsymbol{\beta} - \boldsymbol{\beta}'$. We can also represent $\mathbf{q}_L$ in (4) as

$$\mathbf{q}_L(\boldsymbol{\beta}') = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \left\{\lambda\|\boldsymbol{\beta}\|_1 + \frac{L}{2}\|\boldsymbol{\beta} - \mathbf{u}(\boldsymbol{\beta}', L)\|_2^2\right\}, \tag{10}$$

where $\mathbf{u}(\boldsymbol{\beta}', L) = \boldsymbol{\beta}' + L^{-1}\mathbf{X}^T\mathbf{r}(\boldsymbol{\beta}')$. The minimization problem in (10) is equivalent to the LASSO regression model with $\mathbf{X} = I_n$ and the tuning parameter λ/L. Thus, the optimal solution $\mathbf{q}_L(\boldsymbol{\beta}')$ in (10) has the following explicit form:

$$\mathbf{q}_L(\boldsymbol{\beta}') = S_{\lambda/L}\big(\mathbf{u}(\boldsymbol{\beta}', L)\big), \tag{11}$$

where $S_{\lambda/L}(\mathbf{u}) = \big(\mathrm{Soft}_{\lambda/L}(u_1), \ldots, \mathrm{Soft}_{\lambda/L}(u_p)\big)$ and $\mathrm{Soft}_{\lambda/L}(x) = \mathrm{sign}(x)\max\{|x| - \lambda/L, 0\}$ is the soft-thresholding operator. This soft-thresholding operator is not only simple but is also $S_{\lambda/L}(\mathbf{u})$ is only needed for independent element-wise operations, which are well suited for CUDA GPU parallel computation. Moreover, the inequality in (5) can be verified using the following conditions:

$$\|\mathbf{r}(\boldsymbol{\beta}^1)\|_2^2 - \|\mathbf{r}(\boldsymbol{\beta}^0)\|_2^2 \le L\|\mathbf{d}(\boldsymbol{\beta}^1, \boldsymbol{\beta}^0)\|_2^2 - 2\mathbf{r}(\boldsymbol{\beta}^0)^T\mathbf{X}\mathbf{d}(\boldsymbol{\beta}^1, \boldsymbol{\beta}^0),$$

where $\boldsymbol{\beta}^0$ and $\boldsymbol{\beta}^1$ denote the current and next iterative solutions, respectively. Thus, all operations in the FISTA for LASSO can be expressed by the basic linear algebra operations, and then the FISTA for LASSO is well suited to parallel CUDA GPU computation. Note that the smallest Lipschitz

constant L of the LASSO regression problem is the largest eigenvalue of $\mathbf{X}^T\mathbf{X}$. For moderate-sized issues, we employ $L = \lambda_{\max}(\mathbf{X}^T\mathbf{X})$ and calculate L by using the backtracking procedure when the computational cost for calculating the largest eigenvalue is high. We provide a summary of the FISTA for LASSO with the backtracking procedure in Algorithm 1.

Algorithm 1 FISTA for LASSO

Require: $\mathbf{y} \in \mathbf{R}^n$, $X \in \mathbf{R}^{n\times p}$, $\hat{\boldsymbol{\beta}}^{(0)} \in \mathbf{R}^p$, λ, L_0, η_0, `max_iter`, and $\delta_{\texttt{tol}}$

`t` $\leftarrow 1$, $\hat{\boldsymbol{\beta}}' \leftarrow \hat{\boldsymbol{\beta}}^{(0)}$, $\hat{\boldsymbol{\beta}}^{(prev)} \leftarrow \hat{\boldsymbol{\beta}}^{(0)}$ ▷ initialization
for $k = 1, 2, \ldots,$ `max_iter` **do**
  `ymXbp` $\leftarrow \mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}'$ ▷ $\mathbf{r}(\hat{\boldsymbol{\beta}}')$
  `rbp` $\leftarrow \|\mathbf{r}(\hat{\boldsymbol{\beta}}')\|_2^2$
  `XTrbp` $\leftarrow \mathbf{X}^T\mathbf{r}(\hat{\boldsymbol{\beta}}')$
  $i_k \leftarrow -1$
  repeat ▷ start backtracking procedure
    $i_k \leftarrow i_k + 1$
    $\eta_k = \eta_0^{i_k}$
    $L_k \leftarrow \eta_k L_{k-1}$
    $\hat{\boldsymbol{\beta}} \leftarrow \text{Soft}_{\lambda/L_k}(\hat{\boldsymbol{\beta}}' + L_k^{-1}\texttt{XTrbp})$
    `diff_beta` $\leftarrow \hat{\boldsymbol{\beta}} - \hat{\boldsymbol{\beta}}'$ ▷ $\mathbf{d}(\hat{\boldsymbol{\beta}}, \hat{\boldsymbol{\beta}}')$
    `diff_beta_sq` $\leftarrow \|\mathbf{d}(\hat{\boldsymbol{\beta}}, \hat{\boldsymbol{\beta}}')\|_2^2$
    `XTrbpd` $\leftarrow \mathbf{d}(\hat{\boldsymbol{\beta}}, \hat{\boldsymbol{\beta}}')^T$`XTrbp`
    `RHS` $\leftarrow L_k$`*diff_beta_sq -2.0*XTrbpd`
    `rb` $\leftarrow \|\mathbf{r}(\hat{\boldsymbol{\beta}})\|_2^2$
    `LHS` $\leftarrow$ `rb-rbp` ▷ $\|\mathbf{r}(\hat{\boldsymbol{\beta}})\|_2^2 - \|\mathbf{r}(\hat{\boldsymbol{\beta}}')\|_2^2$
  until `LHS` $\leq$ `RHS` ▷ end backtracking procedure
  `tnext` $\leftarrow (1 + \sqrt{1 + 4\texttt{t}^2})/2$
  `diff_beta` $\leftarrow \hat{\boldsymbol{\beta}} - \hat{\boldsymbol{\beta}}^{(prev)}$
  $\hat{\boldsymbol{\beta}}' \leftarrow \hat{\boldsymbol{\beta}} + (\texttt{t} - 1)/\texttt{tnext}*\texttt{diff\_beta}$
  `diff_beta_L2` $\leftarrow \left\|\mathbf{d}(\hat{\boldsymbol{\beta}}, \hat{\boldsymbol{\beta}}^{(prev)})\right\|_2$
  if `diff_beta_L2` $< \delta_{\text{tol}}$ **then**
    break
  end if
  `t` $\leftarrow$ `tnext`
  $\hat{\boldsymbol{\beta}}^{(prev)} \leftarrow \hat{\boldsymbol{\beta}}$
end for

IV. COMPARATIVE STUDY OF CUDA GPU IMPLEMENTATION OF FISTA FOR LASSO

This section provides the implementation examples of the FISTA for LASSO described in Section III-C along with the implementation approaches introduced in Section II. After exploring all the implementations, we compare their computational efficiency with the samples generated from the model

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{12}$$

where $\mathbf{X} \in \mathbf{R}^{n\times p}$ and $\epsilon \in \mathbf{R}^n$ are generated from the standard normal distribution and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_p)^T$ is defined as

$$\beta_j = \begin{cases} 1.0 & \text{for } j = 1, \ldots, 0.05p \\ 0 & \text{for } j = 0.05p + 1, \ldots, p. \end{cases}$$

We consider solving the minimization problem in (8) with the given observations $\mathbf{y}$ and $\mathbf{X}$. Practically, we need to search for the optimal tuning parameter λ^* for the LASSO regression problem. In this study, however, we just set $\lambda = \sqrt{2\log(p)/n}$ that controls the strength of the penalty term in order to focus on investigating the computational efficiency of the CUDA GPU implementations. Thus, we assume that the following codes are executed before the implementation of the FISTA for LASSO with the given n and p.

```
import numpy as np
np.random.seed(2022)
X = np.random.randn(n,p)
beta_tr = np.zeros(p)
beta_tr[:int(0.05*p)] = 1.0
y = np.dot(X, beta_tr) + np.random.randn(n)
```

Note that we omit the data type conversion here. If users require the single-precision (double-precision) representation, use the `numpy.ndarray.astype()` function with the argument `np.float32` (`np.float64`). The implementation guide is provided in the following subsections.

A. PYTHON WITH A DYNAMIC-LINK LIBRARY BY CUDA C USING KERNEL FUNCTION

Here, we first provide CUDA C kernel functions required in the FISTA for LASSO in Code 11.

The codes in line 1–9 of Code 11 are handled by C preprocessor that includes the header files and substitutes the preprocessor macros defined by the keyword `#define`. The remainder of the Code 11 includes five CUDA C kernel functions. First, the `Sgemv()` function at lines 11–31 in Code 11 conducts the matrix-vector multiplication of either $\mathbf{y} = \alpha A\mathbf{x} + \mathbf{y}$ or $\mathbf{y} = \alpha A^T\mathbf{x} + \mathbf{y}$ on CUDA GPU device. In the `Sgemv()` function, the argument `Bool` determines the type of the matrix-vector multiplication, where $\mathbf{y} = \alpha A\mathbf{x} + \mathbf{y}$ and $\mathbf{y} = \alpha A^T\mathbf{x} + \mathbf{y}$ are conducted if `Bool=false` and `Bool=true`, respectively. Second, the function `soft_thr()` in lines 33–40 of Code 11 provides the soft-thresholding operation $S_\alpha(\mathbf{x})$ defined in (11). Third, the function `Saxpy()` provides the vector addition of two vectors having the form $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$ like `cublasSaxpy()` in the cuBLAS library. Fourth, the function `vec_prod()` conducts the element-wise multiplication between two vectors having the form $z = \mathbf{x} \circ \mathbf{y}$, where $\mathbf{x} \circ \mathbf{y}$ denote the Hadamard product of $\mathbf{x}$ and $\mathbf{y}$. Finally, the function `reduce_sum()` conducts additional demands of every two elements that are apart with the distance `mid_length` in parallel on CUDA GPU device, which is repeatedly called to sum all elements of a given vector in the host function of the FISTA for LASSO.

Here's an implementation example of the host function of the FISTA for LASSO in Code 12.

In Code 12, the codes in lines 3–64 contains the declaration of variables and preparation of CUDA C device variables using the dynamic allocation with the `cudaMalloc()` function. The variable `d_ymXbp` stores the operation $\mathbf{r}(\boldsymbol{\beta}') = \mathbf{y} - \mathbf{X}\boldsymbol{\beta}'$ and variable `d_ymXbp2[0]` stores $\|\mathbf{r}(\boldsymbol{\beta}')\|_2^2$ after the

```
#include<stdio.h>
#include<cuda.h>
#include<cuda_runtime.h>
#include<stdbool.h>
#include<math.h>
#define HTD cudaMemcpyHostToDevice
#define DTH cudaMemcpyDeviceToHost
#define DTD cudaMemcpyDeviceToDevice
#define sign(i) ((i> 0) ? (1) : (-1))

__global__ void Sgemv(float alpha, float *A,
  float *x, float *y, int N, int P, bool Bool) {
  int idx = blockDim.x*blockIdx.x+threadIdx.x;
  float value = 0.0;
  if (Bool == false){
    if (idx < N){
      for (int j = 0; j < P; j++){
        value += A[idx*P+j] * x[j];
      }
      y[idx] += (alpha * value);
    }
  }
  else{
    if (idx < P){
      for (int i = 0; i < N; i++){
        value += A[i*P+idx] * x[i];
      }
      y[idx] += (alpha * value);
    }
  }
}

__global__ void soft_thr(float *x, float alpha,
    float *S, int length) {
  float thr = 0;
  int idx = blockIdx.x*blockDim.x+threadIdx.x;
  if (idx < length){
    thr = fabs(x[idx]) - alpha;
    S[idx] = (thr>0)?(sign(x[idx])*thr):0.0;
  }
}

__global__ void Saxpy(float alpha, float *x,
    float *y, int length) {
  int idx = blockIdx.x*blockDim.x+threadIdx.x;
  if (idx < length) {
    y[idx] += (alpha*x[idx]);
  }
}

__global__ void vec_prod(float *x, float *y,
    float *z, int length) {
  int idx = blockIdx.x*blockDim.x+threadIdx.x;
  if (idx < length) {
    z[idx] = x[idx]*y[idx];
  }
}

__global__ void reduce_sum(float *res, int length
    , int mid_length) {
  int idx = blockIdx.x*blockDim.x+threadIdx.x;
  if (idx + mid_length < length) {
    res[idx] = res[idx]+res[idx+mid_length];
  }
}
```

Code 11. CUDA C kernel functions for mathematical operations in FISTA for LASSO.

```
extern "C"
__host__ void FISTA(float *beta, float *X, float
    *y, float *lambda, float *L, float *Eta,
    float *tolerance, float *Loss, int *
    max_iteration, int *N, int *P, int *steps) {
  int max_iter = *max_iteration;
  int n=*N;
  int p=*P;
  int i, i_k, k;

  float lam=*lambda;
  float tol=*tolerance;
  float eta=*Eta;
  float L_prev=*L;
  float One=1.0, MOne = -1.0;

  float *d_y, *d_ymXbp, *d_ymXbp2, *d_X;
  float *d_beta_p, *d_XTrbp, *d_bstar, *d_beta;
  float *d_diff_beta, *d_diff_beta2;
  float *d_ymXb, *d_ymXb2, *d_beta_prev;
  float *d_XTrbpd, *crit, t1, tnext;
  float eta_ik, L_cur, pL_cur, RHS, LHS;
  float h_rbp=0., h_RHS_1st=0., h_RHS_2nd=0.;
  float h_rb=0., h_crit=0., t = 1.0;

  float log_p=(float)log2(p);
  float log_n=(float)log2(n);
  int step_p=ceil(log_p);
  int step_n=ceil(log_n);
  int mid_len_n=(int)pow(2,step_n-1);
  int mid_len_p=(int)pow(2,step_p-1);
  int mid_tmp_n, mid_tmp_p;
  int nbnp, len_p, len_n;

  bool cond;

  size_t sz_1=sizeof(float);
  size_t sz_np=n*p*sizeof(float);
  size_t sz_n=n*sizeof(float);
  size_t sz_p=p*sizeof(float);

  dim3 TPB(32,1);
  dim3 BPG(1,1);

  unsigned int bpg_p=ceil(((float)p)/TPB.x);
  unsigned int bpg_n=ceil(((float)n)/TPB.x);

  cudaMalloc(&d_y, sz_n);
  cudaMalloc(&d_ymXbp, sz_n);
  cudaMalloc(&d_ymXbp2, sz_n);
  cudaMalloc(&d_X, sz_np);
  cudaMalloc(&d_beta_p, sz_p);
  cudaMalloc(&d_XTrbp, sz_p);
  cudaMalloc(&d_bstar, sz_p);
  cudaMalloc(&d_beta, sz_p);
  cudaMalloc(&d_diff_beta, sz_p);
  cudaMalloc(&d_diff_beta2, sz_p);
  cudaMalloc(&d_ymXb, sz_n);
  cudaMalloc(&d_ymXb2, sz_n);
  cudaMalloc(&d_beta_prev, sz_p);
  cudaMalloc(&d_XTrbpd, sz_p);

  cudaMemcpy(d_y, y, sz_n, HTD);
  cudaMemcpy(d_X, X, sz_np, HTD);
  cudaMemcpy(d_beta_p, beta, sz_p, HTD);
  cudaMemcpy(d_beta_prev, beta, sz_p, HTD);
  crit=(float *)malloc(sz_1*max_iter);

  for(k = 0; k < max_iter; k++) {
    BPG.x = bpg_n;
    cudaMemcpy(d_ymXbp, d_y, sz_n, DTD);
    Sgemv<<<BPG,TPB>>>(MOne, d_X, d_beta_p,
                       d_ymXbp, n, p, false);
```

Code 12. Host function of the FISTA for LASSO using the user-defined CUDA C kernel function.

reduced summation by the codes in lines 66–83 of Code 12. The variable `d_XTrbp` also stores $\mathbf{X}^T\mathbf{r}(\boldsymbol{\beta}')$ in lines 85–88 of Code 12. The codes in lines 89–147 conducts the backtracking procedure described in lines 7–18 in Algorithm 1. The codes in lines 149–157 correspond to the updating equations in (7). To check the convergence of the FISTA, we use the ℓ_2-norm of the difference between the previous and current

```
    vec_prod<<<BPG,TPB>>>(d_ymXbp, d_ymXbp,
                          d_ymXbp2, n);
    mid_tmp_n = mid_len_n;
    len_n = n;
    for(i = 0; i < step_n; i++) {
      nbnp = ceil(((float)mid_tmp_n)/TPB.x);
      reduce_sum<<<nbnp,TPB>>>(d_ymXbp2,len_n,
                               mid_tmp_n);
      len_n = mid_tmp_n;
      mid_tmp_n /= 2;
    }
    cudaMemcpy(&h_rbp, &d_ymXbp2[0],
               sizeof(float), DTH);

    cudaMemset(d_XTrbp, 0, sz_p);
    BPG.x = bpg_p;
    Sgemv<<<BPG,TPB>>>(One, d_X, d_ymXbp,
                       d_XTrbp, n, p, true);
    i_k = -1;
    cond = true;
    while (cond) {
      i_k += 1;
      eta_ik = pow(eta, i_k);
      L_cur = L_prev * eta_ik;
      pL_cur = float(1.0 / L_cur);
      BPG.x = bpg_p;
      cudaMemcpy(d_bstar, d_beta_p, sz_p, DTD);
      Saxpy<<<BPG,TPB>>>(pL_cur, d_XTrbp,
                         d_bstar, p);
      soft_thr<<<BPG,TPB>>>(d_bstar,lam/L_cur,
                            d_beta,p);
      cudaMemcpy(d_diff_beta,d_beta,sz_p,DTD);
      Saxpy<<<BPG, TPB>>>(MOne, d_beta_p,
                          d_diff_beta, p);

      vec_prod<<<BPG,TPB>>>(d_diff_beta,
                d_diff_beta, d_diff_beta2, p);
      vec_prod<<<BPG,TPB>>>(d_XTrbp,
                d_diff_beta, d_XTrbpd, p);
      mid_tmp_p = mid_len_p;
      len_p = p;
      for(i = 0; i < step_p; i++) {
        nbnp = ceil(((float)mid_tmp_p)/TPB.x);
        reduce_sum<<<nbnp,TPB>>>(d_diff_beta2,
                                 len_p, mid_tmp_p);
        reduce_sum<<<nbnp,TPB>>>(d_XTrbpd,
                                 len_p, mid_tmp_p);
        len_p = mid_tmp_p;
        mid_tmp_p /= 2;
      }
      cudaMemcpy(&h_RHS_1st, &d_diff_beta2[0],
                 sizeof(float), DTH);
      cudaMemcpy(&h_RHS_2nd, &d_XTrbpd[0],
                 sizeof(float), DTH);
      RHS = L_cur*h_RHS_1st-2.0*h_RHS_2nd;

      mid_tmp_n = mid_len_n;
      len_n = n;
      BPG.x = bpg_n;
      cudaMemcpy(d_ymXb, d_y, sz_n, DTD);
      Sgemv<<<BPG, TPB>>>(MOne, d_X, d_beta,
                          d_ymXb, n, p, false);
      vec_prod<<<BPG, TPB>>>(d_ymXb, d_ymXb,
                             d_ymXb2, n);
      for(i = 0; i < step_n; i++) {
        nbnp = ceil(((float)mid_tmp_n)/TPB.x);
        reduce_sum<<<nbnp,TPB>>>(d_ymXb2,
                                 len_n, mid_tmp_n);
        len_n = mid_tmp_n;
        mid_tmp_n /= 2;
      }
      cudaMemcpy(&h_rb, &d_ymXb2[0],
                 sizeof(float), DTH);
```

Code 12. ***(Continued.)*** **Host function of the FISTA for LASSO using the user-defined CUDA C kernel function.**

iterative solutions, which correspond to the codes in lines 159–176 of Code 12. Finally, all the dynamically allocated

```
      LHS = h_rb - h_rbp;
      cond = (LHS > RHS);
    }
    L_prev = L_cur;

    tnext =  (1.0+sqrtf(1+4*t*t))/2.0;
    BPG.x = bpg_p;
    cudaMemcpy(d_diff_beta, d_beta, sz_p, DTD);
    Saxpy<<<BPG,TPB>>>(MOne, d_beta_prev,
                       d_diff_beta, p);
    t1 = (t - 1.0)/tnext;
    cudaMemcpy(d_beta_p, d_beta, sz_p, DTD);
    Saxpy<<<BPG,TPB>>>(t1, d_diff_beta,
                       d_beta_p, p);

    vec_prod<<<BPG,TPB>>>(d_diff_beta,
                d_diff_beta, d_diff_beta2, p);
    len_p = p;
    mid_tmp_p = mid_len_p;
    for(i = 0; i < step_p; i++) {
      nbnp = ceil(((float) mid_tmp_p) / TPB.x);
      reduce_sum<<<nbnp, TPB>>>(d_diff_beta2,
                                len_p, mid_tmp_p);
      len_p = mid_tmp_p;
      mid_tmp_p /= 2;
    }
    cudaMemcpy(&h_crit, &d_diff_beta2[0],
               sizeof(float), DTH);
    crit[k] = sqrtf(h_crit);
    if (crit[k] < tol)
      break;
    t = tnext;
    cudaMemcpy(d_beta_prev, d_beta, sz_p, DTD);
  }

  *steps = k;
  memcpy(Loss, crit, sizeof(float) * max_iter);
  cudaMemcpy(beta, d_beta, sz_p, DTH);

  free(crit);
  cudaFree(d_y);
  cudaFree(d_ymXbp);
  cudaFree(d_ymXbp2);
  cudaFree(d_X);
  cudaFree(d_beta_p);
  cudaFree(d_XTrbp);
  cudaFree(d_bstar);
  cudaFree(d_beta);
  cudaFree(d_diff_beta);
  cudaFree(d_diff_beta2);
  cudaFree(d_ymXb);
  cudaFree(d_ymXb2);
  cudaFree(d_beta_prev);
  cudaFree(d_XTrbpd);
}
```

Code 12. ***(Continued.)*** **Host function of the FISTA for LASSO using the user-defined CUDA C kernel function.**

memory spaces are freed by the codes in lines 183–197 of Code 12.

To employ the implemented functions, we first need to compile the codes in Codes 11 and 12 as given in Code 3, where we use the name of the dynamic-link library as `fista_kernel.so`. After compilation, we can conduct the FISTA for LASSO on CUDA GPU device in Python with the codes shown in Code 13.

The codes in Code 13 comprises three parts. The first part is to load the dynamic-link library and define the types of the arguments in lines 1–11. The second part involves setting the parameters required in the FISTA for LASSO in lines 13–28.

```
libc_FISTA_dll = CDLL("./fista_kernel.so")
FISTA_DLL = libc_FISTA_dll.FISTA

FISTA_DLL.restype = None
FISTA_DLL.argtypes = (POINTER(c_float),
          POINTER(c_float), POINTER(c_float),
          POINTER(c_float), POINTER(c_float),
          POINTER(c_float), POINTER(c_float),
          POINTER(c_float), POINTER(c_int),
          POINTER(c_int), POINTER(c_int),
          POINTER(c_int))

beta = np.zeros(p).astype(np.float32)
beta_c = beta.ctypes.data_as(POINTER(c_float))
X_ca =  X.ctypes.data_as(POINTER(c_float))
y_c = y.ctypes.data_as(POINTER(c_float))
max_iter = 5000
loss = np.zeros(max_iter).astype(np.float32)
loss_c = loss.ctypes.data_as(POINTER(c_float))
lam =  np.sqrt(2*np.log(p)/n)
lam_c = c_float(lam)
L_c = c_float(10)
eta_c = c_float(2)
tol_c = c_float(1e-3)
iter_c = c_int(max_iter)
n_c = c_int(n)
p_c = c_int(p)
step_c = c_int(0)

FISTA_DLL(beta_c,X_ca,y_c,lam_c,L_c,eta_c,
          tol_c,loss_c,iter_c,n_c,p_c,step_c)
```

Code 13. Python code for FISTA for LASSO using the user-defined kernel function.

Here, we set the convergence tolerance as 10^{-3} and $\eta = 2$ and $L_0 = 10$ for $L_k = L_{k-1}\eta^{i_k}$ with $i_k \geq 0$ in the backtracking procedure of the FISTA. The variable `loss` stores the convergence criterion $\|\boldsymbol{\beta}^{(k)} - \boldsymbol{\beta}^{(k-1)}\|_2$ for $k \geq 1$ and the variable `step_c` stores the number of iterations needed to satisfy the convergence criterion. Finally, we obtain the solution of the LASSO regression with `FISTA_Kernel()` function in line 30 in Code 13. Our solution is stored in the variable `beta` after running the function `FISTA_Kernel()`.

B. PYTHON WITH A DYNAMIC-LINK LIBRARY BY CUDA C USING cuBLAS LIBRARY

To use the cuBLAS library, we need to replace the codes in lines 67–197 in Code 12 with the codes in Code 14. In the implementation with the cuBLAS library, we only keep the CUDA C kernel function `soft_thr()` in Code 11.

The roles of the variables in Code 14 are identical to those in Code 12. As demonstrated in Code 14, we can simplify the codes in 12 using the functions provided by the cuBLAS library. For example, the codes in lines 71–83 of Code 12 are comparable to the code in lines 56 and 57 in Code 14. In addition, the functions in the cuBLAS library are usually well-optimized; thereby helping us achieve not only the concise expressions but also the computational efficiency with the cuBLAS library. A numerical comparison is provided in Section IV-G.

To execute the implemented function with the cuBLAS library, we have to apply the same procedure in the Python with a dynamic-link library using the user-defined kernel functions. In the compilation step, we need to add

```
BPG.x = bpg_p;

cublasHandle_t handle;
cublasOperation_t tran = CUBLAS_OP_T;
cublasOperation_t ntran = CUBLAS_OP_N;

float *d_y, *d_ymXbp, *d_X, *d_beta_p, *d_XTrbp;
float *d_bstar, *d_beta, *d_diff_beta;
float *d_ymXb, *d_beta_prev, *crit;
float *d_One, *d_MOne, *d_Zero;
float eta_ik, L_cur, pL_cur, RHS, LHS, tnext, t1;
float h_rbp, h_RHS_1st, h_RHS_2nd, h_rb, h_crit;
float *d_rbp, *d_RHS_1st, *d_RHS_2nd, *d_rb;
float t = 1.0, *d_t1, *d_pL_cur, *d_crit;
bool cond;

cudaMalloc(&d_y, sz_n);
cudaMalloc(&d_ymXbp, sz_n);
cudaMalloc(&d_X, sz_np);
cudaMalloc(&d_beta_p, sz_p);
cudaMalloc(&d_XTrbp, sz_p);
cudaMalloc(&d_bstar, sz_p);
cudaMalloc(&d_beta, sz_p);
cudaMalloc(&d_diff_beta, sz_p);
cudaMalloc(&d_ymXb, sz_n);
cudaMalloc(&d_beta_prev, sz_p);
cudaMalloc(&d_One, sz_1);
cudaMalloc(&d_Zero, sz_1);
cudaMalloc(&d_MOne, sz_1);
cudaMalloc(&d_rbp, sz_1);
cudaMalloc(&d_rb, sz_1);
cudaMalloc(&d_RHS_1st, sz_1);
cudaMalloc(&d_RHS_2nd, sz_1);
cudaMalloc(&d_crit, sz_1);
cudaMalloc(&d_t1, sz_1);
cudaMalloc(&d_pL_cur, sz_1);

cudaMemcpy(d_One, &One, sz_1, HTD);
cudaMemcpy(d_MOne, &MOne, sz_1, HTD);
cudaMemcpy(d_Zero, &Zero, sz_1, HTD);

cudaMemcpy(d_y, y, sz_n, HTD);
cudaMemcpy(d_X, X, sz_np, HTD);
cudaMemcpy(d_beta_p, beta, sz_p, HTD);
cudaMemcpy(d_beta_prev, beta, sz_p, HTD);
crit = (float *)malloc(sz_1 * max_iter);

cublasCreate(&handle);
cublasSetPointerMode(handle,
                     CUBLAS_POINTER_MODE_DEVICE);
for(k=0;k<max_iter;k++)
{
  cudaMemcpy(d_ymXbp,d_y,sz_n,DTD);
  cublasSgemv(handle,ntran,n,p,d_MOne,d_X,n,
              d_beta_p,1,d_One,d_ymXbp,1);
  cublasSdot(handle,n,d_ymXbp,1,d_ymXbp,1,d_rbp);
  cudaMemcpy(&h_rbp, d_rbp, sz_1, DTH);
  cublasSgemv(handle, tran, n, p, d_One, d_X, n,
              d_ymXbp, 1, d_Zero, d_XTrbp, 1);
  i_k = -1;
  cond = true;
  while(cond)
  {
    i_k += 1;
    eta_ik = pow(eta, i_k);
    L_cur = L_prev * eta_ik;
    pL_cur = 1.0 / L_cur;
    cudaMemcpy(d_pL_cur,&pL_cur,sz_1, HTD);
    cudaMemcpy(d_bstar, d_beta_p, sz_p, DTD);
    cublasSaxpy(handle,p,d_pL_cur,
                d_XTrbp,1,d_bstar,1);
```

Code 14. Host function of the FISTA for LASSO using the cuBLAS library.

the option `-lcublas` in the command of `nvcc` as provided in Section II. The name of the compiled library

```
    soft_thr<<<BPG, TPB>>>(d_bstar,lam/L_cur,
                           d_beta, p);
    cudaMemcpy(d_diff_beta, d_beta, sz_p, DTD);
    cublasSaxpy(handle, p, d_MOne,d_beta_p,1,
                d_diff_beta, 1);
    cublasSdot(handle, p, d_diff_beta, 1,
               d_diff_beta, 1, d_RHS_1st);
    cublasSdot(handle, p, d_diff_beta, 1,
               d_XTrbp, 1, d_RHS_2nd);
    cudaMemcpy(&h_RHS_1st, d_RHS_1st, sz_1, DTH);
    cudaMemcpy(&h_RHS_2nd, d_RHS_2nd, sz_1, DTH);
    RHS = L_cur * h_RHS_1st - 2.0 * h_RHS_2nd;

    cudaMemcpy(d_ymXb, d_y, sz_n, DTD);
    cublasSgemv(handle,ntran,n,p,d_MOne,d_X,n,
                d_beta,1,d_One,d_ymXb,1);
    cublasSdot(handle,n,d_ymXb,1,d_ymXb,1,d_rb);
    cudaMemcpy(&h_rb, d_rb, sz_1, DTH);
    LHS = h_rb - h_rbp;
    cond = (LHS > RHS);
  }
  tnext = (1.0+sqrtf(1+4*t*t))/2.0;
  cudaMemcpy(d_diff_beta, d_beta, sz_p, DTD);
  cublasSaxpy(handle,p,d_MOne,d_beta_prev,1,
              d_diff_beta,1);
  t1 = (t-1.0)/tnext;
  cudaMemcpy(d_t1, &t1, sz_1, HTD);
  cudaMemcpy(d_beta_p, d_beta, sz_p, DTD);
  cublasSaxpy(handle,p,d_t1,d_diff_beta,1,
              d_beta_p,1);
  cublasSdot(handle,p,d_diff_beta,1,
             d_diff_beta,1,d_crit);
  cudaMemcpy(&h_crit, d_crit, sz_1, DTH);
  crit[k] = sqrtf(h_crit);
  if (crit[k] < tol)
    break;
  t = tnext;
  L_prev = L_cur;
  cudaMemcpy(d_beta_prev, d_beta, sz_p, DTD);
}
*steps = k;
memcpy(Loss, crit, sz_1 * max_iter);
cudaMemcpy(beta, d_beta, sz_p, DTH);
cublasDestroy(handle);
free(crit);
cudaFree(d_y);
cudaFree(d_ymXbp);
cudaFree(d_X);
cudaFree(d_beta_p);
cudaFree(d_XTrbp);
cudaFree(d_bstar);
cudaFree(d_beta);
cudaFree(d_diff_beta);
cudaFree(d_ymXb);
cudaFree(d_beta_prev);
cudaFree(d_One);
cudaFree(d_Zero);
cudaFree(d_MOne);
cudaFree(d_rbp);
cudaFree(d_rb);
cudaFree(d_RHS_1st);
cudaFree(d_RHS_2nd);
cudaFree(d_crit);
cudaFree(d_t1);
cudaFree(d_pL_cur);
```

Code 14. ***(Continued.)*** **Host function of the FISTA for LASSO using the cuBLAS library.**

is `fista_blas.so`. After compilation, we can conduct the FISTA for LASSO with the cuBLAS library in Python by changing the file name `fista_kernel.so` to `fista_blas.so` in line 1 and changing the code in line 15 to

```
X_c =  X.ravel(order = "F")
X_ca = X_c.ctypes.data_as(POINTER(c_float))
```

in Code 13.

C. PyCUDA

This section provides an implementation example of the FISTA for LASSO with the PyCUDA library. First, we suppose that the codes of the CUDA C kernel functions in Code 11 are stored in a file `FISTA_kernel.cu`. Then, we define a Python function for FISTA using the PyCUDA library, as shown in Code 15.

In Code 15, we first load the PyCUDA and Numpy libraries and compile the user-defined kernel functions stored in `FISTA_kernel.cu` using `SourceModule()` function of the PyCUDA library in lines 1–8. The function `Module.get_function()` of the PyCUDA used in lines 9 and 10 makes the CUDA C kernel function callable in Python platform, where `Module` is a Python class provided by the PyCUDA library. In the PyCUDA example, we use the `Sgemv()` and `soft_thr()` kernel functions. The codes in lines 12–33 in Code 15 are used to declare the required variables in the host and device memories, where the `gpuarray` is a Python class provided by the PyCUDA library to handle the device variables and several linear algebraic operations. The roles of the variables in Code 15 are also exactly same as those in Code 11. We omit the explanation of the remaining codes in Code 15 to avoid duplication.

To conduct the `FISTA()` function by the PyCUDA library, we simply write the Python codes as shown in Code 16.
The advantage of the PyCUDA library is that we can write the host function to call the CUDA C kernel functions with Python-syntax and avoid the direct compilation of the CUDA C source code by `nvcc` on the command line.

D. NUMBA

This section provides an implementation example of the FISTA for LASSO using the Numba library. As shown in Code 11, we first write the CUDA C kernel function using the decorator `numba.cuda.jit` as given in Code 17, where `cuda.jit` is used by `from numba import cuda` for ease of use. With the `numba.cuda.jit` decorator, we can write the CUDA C kernel function with Python-syntax. The roles of the kernel functions in Code 17 are exactly same as of those in Code 11, except the `reduce()` function. The `reduce()` function is defined for the sum of all elements in a given vector with the decorator `numba.cuda.reduce` that supports a reduction algorithm running on a GPU device.

Utilizing the defined kernel functions, we write the Python function for the FISTA for LASSO as given in Code 18.

In Code 18, we additionally consider an argument `dtype` to cover the single-precision and double-precision representations. Note that the Numba library supports the Numpy data types and we simply use `numpy.float32` and `numpy.float64` for single-precision and double-precision, respectively. The roles of the variables in Code 18 are exactly same as of those in Code 12. We omit the

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
from pycuda import gpuarray, tools, cumath
import numpy as np
with open("./FISTA_kernel.cu") as f:
  FISTA_kernel = f.read()
FISTA_md = SourceModule(FISTA_kernel)
Sgemv = FISTA_md.get_function("Sgemv")
soft_thr= FISTA_md.get_function("soft_thr")

def FISTA(beta, X, y, lam, L, eta, tol=1e-4,
    max_iter=5000):
  n = np.int32(X.shape[0])
  p = np.int32(X.shape[1])
  t = np.float32(1.0)
  One = np.float32(1.0)
  MOne = np.float32(-1.0)
  IntOne = np.int32(1)
  IntZero = np.int32(0)
  crit = np.zeros(max_iter, dtype=np.float32)
  temp = np.zeros(p, dtype=np.float32)
  d_beta_p = gpuarray.to_gpu(temp)
  d_beta_prev = gpuarray.to_gpu(temp)
  d_X = gpuarray.to_gpu(X)
  d_y = gpuarray.to_gpu(y)
  d_ymXbp = gpuarray.empty(n, dtype = np.float32)
  d_beta = gpuarray.empty(p, dtype = np.float32)
  TPB = (32, 1, 1)
  bpg_p = math.ceil(np.float32(p)/TPB[0])
  bpg_n = math.ceil(np.float32(n)/TPB[0])
  BPG_p = (bpg_p, 1, 1)
  BPG_n = (bpg_n, 1, 1)
  L_prev = L

  for k in range(max_iter):
    d_ymXbp = d_y.copy()
    Sgemv(MOne, d_X, d_beta_p,
          d_ymXbp, n, p, IntZero,
          grid = BPG_n, block = TPB)
    h_rbp = gpuarray.dot(d_ymXbp, d_ymXbp)
    d_XTrbp = gpuarray.zeros(p, np.float32)
    Sgemv(One, d_X, d_ymXbp, d_XTrbp, n, p,
          IntOne, grid = BPG_p, block = TPB)

    i_k = -1
    cond = True
    while cond:
      i_k += 1
      eta_ik = eta ** i_k
      L_cur = L_prev * eta_ik
      alpha = np.float32(1.0/L_cur)
      d_bstar = d_beta_p + alpha*d_XTrbp
      alpha = np.float32(lam/L_cur)
      soft_thr(d_bstar, alpha, d_beta, p,
               grid = BPG_p, block = TPB)
      d_diff_beta = d_beta - d_beta_p
      h_RHS_1st = gpuarray.dot(d_diff_beta,
                               d_diff_beta)
      h_RHS_2nd = gpuarray.dot(d_diff_beta,
                               d_XTrbp)

      RHS1 = L_cur * h_RHS_1st.get()
      RHS2 = np.float32(2.0)*h_RHS_2nd.get()
      RHS =  RHS1 - RHS2

      d_ymXb = d_y.copy()
      Sgemv(MOne, d_X, d_beta, d_ymXb, n, p,
            IntZero, grid = BPG_n, block = TPB)
      d_ymXb2 = gpuarray.dot(d_ymXb, d_ymXb)
      LHS = d_ymXb2.get() - h_rbp.get()
      cond = (LHS > RHS)
```

Code 15. **Python function for FISTA for LASSO using the PyCUDA library.**

explanation of remaining codes in Code 18 to avoid duplication. To conduct the `FISTA()` function by the Numba library, we use the same codes in Code 16 with the additional

```

    L_prev = L_cur
    tnext = np.float32((1.0+np.sqrt(1+4*t*t))/2)
    d_diff_beta = d_beta - d_beta_prev
    alpha = np.float32((t - 1.0)/tnext)
    d_beta_p = d_beta + alpha * d_diff_beta
    d_diff_b_sq = gpuarray.dot(d_diff_beta,
                               d_diff_beta)
    crit[k] = np.sqrt(d_diff_b_sq.get())
    if crit[k] < tol:
        break
    t = tnext
    d_beta_prev = d_beta.copy()
  return d_beta.get(), crit, k
```

Code 15. ***(Continued.)*** **Python function for FISTA for LASSO using the PyCUDA library.**

```
beta = np.zeros(p, dtype = np.float32)
lam = np.sqrt(2*np.log(p)/n, dtype = np.float32)
L = np.float32(10)
eta = np.float32(2)
tol = np.float32(1e-03)
out, crit, step = FISTA(beta,X,y,lam,L,eta,tol)
```

Code 16. **Python code for conducting the FISTA for LASSO implemented by the PyCUDA library.**

argument `dtype`. The Numba library offers all the implementation advantages described in the PyCUDA library, and provides the additional advantage of writing CUDA C kernel functions with Python-syntax using the decorator `numba.cuda.jit`.

E. TENSORFLOW

This section demonstrates an implementation example of the FISTA for LASSO using the TensorFlow library. As the TensorFlow library includes the functions for the basic linear algebraic operations on a GPU device, we only write a function `soft_thr()` for the soft-thresholding operation in the FISTA for LASSO as given in Code 19 with the decorator `tf.function`, which is required to conduct operations with `tf.Tensor` object.

In Code 19, we use a statement `with tf.device()` to explicitly select the target GPU device. As the TensorFlow library automatically selects the GPU device depending on its availability, the statement related to `with` keywords can be removed for conciseness of the implementation. We use the `tf.linalg.matvec()` and `tf.tensordot()` for the matrix-vector multiplication and the inner production of two vectors, respectively. We omit the explanation of remaining codes in Code 19 to avoid duplication. To conduct the `FISTA()` function by the TensorFlow library, we use the same codes in Code 16 with the additional argument `dtype` as well. The TensorFlow library has the all implementation advantages described in the Numba library, and provides an additional advantage where we do not have to write any CUDA C kernel function for the basic linear algebraic operations.

F. PYTORCH

This section provides an implementation example of the FISTA for LASSO using the PyTorch library. The PyTorch

```python
import math
from numba import cuda

@cuda.jit
def gemv(alpha, A, x, y, tran):
  Row = cuda.grid(1)
  n = A.shape[0]
  p = A.shape[1]
  value = 0.
  if tran:
    if Row < p:
      for i in range(n):
          value += A[i, Row] * x[i]
      y[Row] += alpha * value
  else:
    if Row < n:
      for j in range(p):
          value += A[Row, j] * x[j]
        y[Row] += alpha * value

@cuda.jit
def softthr(x, alpha, S):
  idx = cuda.grid(1)
  if idx < x.shape[0]:
    thr = math.fabs(x[idx]) - alpha
    if thr > 0:
      S[idx] = math.copysign(thr, x[idx])
    else:
      S[idx] = 0

@cuda.jit
def axpy(alpha, x, y):
  idx = cuda.grid(1)
  if idx < x.shape[0]:
    y[idx] += alpha * x[idx]

@cuda.jit
def vec_prod(x, y, res):
  idx = cuda.grid(1)
  if idx < x.shape[0]:
    res[idx] = x[idx] * y[idx]

@cuda.reduce
def reduce(x, y):
  return x+y
```

Code 17. CUDA kernel functions for mathematical operations in FISTA for LASSO using the numba library.

library provides the functions for the basic linear algebraic operations on a GPU device similar to the TensorFlow library. Thus, we write a function `soft_thr()` for the soft-thresholding operation in the FISTA for LASSO as given in Code 20 using the function `torch.maximum()`, which conducts the element-wise maximum of given `torch.Tensor` objects and is available from `PyTorch 1.7.0`.

In Code 20, we used `torch.Tensor.to()` function to transfer data between the host and device memories, where `"cpu"` and `"cuda:0"` denote the host and device memories, respectively. Unlike the TensorFlow library, we should use the explicit memory transfer from the host to the device memory to conduct the operations on a GPU device. We use the `torch.matmul()` and `torch.dot()` for the matrix-vector multiplication, and the inner production of two vectors, respectively. We omit the explanation of the remaining codes in Code 20 to avoid redundancy as well. To conduct the `FISTA()` function by the PyTorch library, we use the same codes in Code 16 with the additional argument `dtype`. The

```python
def FISTA(beta,X,y,lam,L,eta,tol=1e-4,
          max_iter=5000, dtype=np.float32):
  n = X.shape[0]
  p = X.shape[1]
  t = dtype(1.0)
  crit = np.zeros(max_iter,dtype=dtype)
  d_bstar = np.zeros(p,dtype=dtype)
  d_beta = np.zeros(p,dtype=dtype)
  d_diff_beta2 = np.zeros(p,dtype=dtype)
  d_XTrbpd = np.zeros(p,dtype=dtype)
  d_beta_prev = np.zeros(p,dtype=dtype)
  d_beta_p = np.zeros(p,dtype=dtype)
  d_ymXbp2 = np.zeros(n,dtype=dtype)
  d_ymXb2 = np.zeros(n,dtype=dtype)
  L_prev = L
  TPB = (32,1)
  BPGp = (math.ceil(p/32),1)
  BPGn = (math.ceil(n/32),1)

  for k in range(max_iter):
    d_ymXbp = y.copy()
    gemv[BPGn,TPB](-1.0,X,d_beta_p,d_ymXbp,False)
    vec_prod[BPGn,TPB](d_ymXbp,d_ymXbp,d_ymXbp2)
    h_rbp = reduce(d_ymXbp2)
    d_XTrbp = np.zeros(p,dtype=dtype)
    gemv[BPGp,TPB](1.0,X,d_ymXbp,d_XTrbp,True)

    i_k = -1
    cond = True
    while cond:
      i_k += 1
      L_cur = L_prev * (eta ** i_k)
      d_bstar = d_beta_p.copy()
      axpy[BPGp,TPB]((1.0/L_cur),d_XTrbp,d_bstar)
      softthr[BPGp,TPB](d_bstar,lam/L_cur,d_beta)
      d_diff_beta = d_beta.copy()
      axpy[BPGp,TPB](-1.0,d_beta_p,d_diff_beta)
      vec_prod[BPGp,TPB](d_diff_beta,d_diff_beta,
                         d_diff_beta2)
      vec_prod[BPGp,TPB](d_XTrbp,d_diff_beta,
                         d_XTrbpd)
      RHS = L_cur*reduce(d_diff_beta2)-
              2.0*reduce(d_XTrbpd)
      d_ymXb = y.copy()
      gemv[BPGn,TPB](-1.0,X,d_beta,d_ymXb,False)
      vec_prod[BPGn,TPB](d_ymXb,d_ymXb,d_ymXb2)
      h_rb = reduce(d_ymXb2)
      LHS =  h_rb-h_rbp
      cond = (LHS>RHS)

    L_prev = L_cur
    tnext = (1.0+np.sqrt(1 + 4 * t**2))/2.0
    d_diff_beta = d_beta.copy()
    axpy[BPGp,TPB](-1.0,d_beta_prev,d_diff_beta)
    t1 = (t-1.0)/tnext
    d_beta_p = d_beta.copy()
    axpy[BPGp,TPB](t1,d_diff_beta,d_beta_p)
    vec_prod[BPGp,TPB](d_diff_beta,d_diff_beta,
                       d_diff_beta2)
    crit[k] = np.sqrt(reduce(d_diff_beta2))
    if crit[k] < tol:
      break
    t = tnext
    d_beta_prev = d_beta.copy()
  return d_beta, crit, k
```

Code 18. Python function of the FISTA for LASSO using the numba library.

PyTorch library offers all the same implementation advantages as those offered by the TensorFlow library.

G. NUMERICAL COMPARISON OF FISTA FOR LASSO

This section numerically compares the computational efficiency of the implementation approaches provided in

```python
import tensorflow as tf
import numpy as np

@tf.function
def soft_thr(x,alpha):
  n = x.shape[0]
  S = tf.math.maximum(tf.abs(x)-alpha,0)*tf.math.
    sign(x)
  return S

def FISTA(beta, X, y, lamb, L, eta, tol=1e-4,
    max_iter=5000, dtype=tf.float32):
  n = X.shape[0]
  p = X.shape[1]
  crit = np.zeros(max_iter)

  with tf.device("/GPU:0"):
    dbeta = tf.constant(beta)
    dX = tf.constant(X)
    dy = tf.constant(y)
    t = tf.constant(1.0, dtype=dtype)
    dbeta_p = tf.constant(beta)
    dbeta_prev = tf.constant(beta)
    L_prev = tf.constant(L,dtype=dtype)

    for k in range(max_iter):
      dymXbp = dy-tf.linalg.matvec(dX,dbeta_p)
      drbp = tf.tensordot(dymXbp,dymXbp,axes=1)
      dXTrbp = tf.linalg.matvec(dX,dymXbp,
                                transpose_a=True)
      i_k = -1
      cond = True
      while cond:
        i_k += 1
        L_cur = L_prev*(eta**i_k)
        dbstar = dbeta_p + dXTrbp/L_cur
        dbeta = soft_thr(dbstar, lamb/L_cur)
        diff_beta = dbeta - dbeta_p
        RHS_1st = tf.tensordot(diff_beta,
                               diff_beta,axes=1)
        RHS_2nd = tf.tensordot(diff_beta, dXTrbp,
                               axes=1)
        RHS = L_cur * RHS_1st.numpy() -
              2.0 * RHS_2nd.numpy()
        dymXb = dy - tf.linalg.matvec(dX,dbeta)
        LHS = (tf.tensordot(dymXb,dymXb,
               axes=1)).numpy()-drbp.numpy()
        cond = (LHS>RHS)
      L_prev = L_cur
      tnext =  (1.0+tf.sqrt(1+4*t**2))/2.0
      diff_beta = dbeta-dbeta_prev
      t1 = (t - 1.0)/tnext
      dbeta_p = dbeta+t1*diff_beta
      crit[k] = tf.norm(diff_beta)
      if crit[k] < tol:
          break
      t = tnext
      dbeta_prev = dbeta
  return dbeta.numpy(), crit, k
```

Code 19. Python function of the FISTA for LASSO using the tensorflow library.

```python
def soft_thr(x, alpha):
  n=x.shape[0]
  S=torch.maximum(torch.abs(x)-alpha,
    torch.zeros(n,device="cuda:0"))*torch.sign(x)
  return S

def FISTA(beta, X, y, lam, L, eta , tol = 1e-04,
    max_iter = 5000, dtype = torch.float32):
  if(dtype == torch.float32):
    torch.set_default_tensor_type(torch.
    FloatTensor)
  else:
    torch.set_default_tensor_type(torch.
    DoubleTensor)
  device = torch.device("cuda:0")
  n = X.shape[0]
  p = X.shape[1]
  dbeta = torch.Tensor(beta).to(device)
  dX = torch.Tensor(X).to(device)
  dy = torch.Tensor(y).to(device)
  t = torch.ones(1,dtype=dtype, device=device)
  crit = np.zeros(max_iter)
  dbeta_p = torch.Tensor(beta).to(device)
  dbeta_prev = torch.Tensor(beta).to(device)
  L_prev = L
  for k in range(max_iter):
    dymXbp = dy-torch.matmul(dX,dbeta_p)
    drbp = torch.dot(dymXbp, dymXbp)
    dXTrbp = torch.matmul(torch.t(dX),dymXbp)

    i_k = -1
    cond = True
    while cond:
      i_k += 1
      L_cur = L_prev*(eta**i_k)
      dbstar = dbeta_p + dXTrbp/L_cur
      dbeta = soft_thr(dbstar, lam/L_cur)
      diff_beta = dbeta - dbeta_p
      RHS_1st = torch.dot(diff_beta,diff_beta)
      RHS_2nd = torch.dot(diff_beta,dXTrbp)
      RHS = L_cur*RHS_1st-2.0*RHS_2nd
      dymXb = dy-torch.matmul(dX,dbeta)
      LHS = torch.dot(dymXb,dymXb)-drbp
      cond = (LHS>RHS)
    L_prev = L_cur
    tnext = (1.0+torch.sqrt(1+4*t**2))/2.0
    diff_beta = dbeta-dbeta_prev
    t1 = (t-1.0)/tnext
    dbeta_p = dbeta+t1*diff_beta
    crit[k] = torch.norm(diff_beta)
    if crit[k] < tol:
      break
    t = tnext
    dbeta_prev = dbeta
  out = dbeta.to('cpu')
  return out.numpy(), crit, k
```

Code 20. Python function of the FISTA for LASSO using the PyTorch library.

Section IV. As described in Section II-G, we measure the execution time in seconds for each implementation approach on the same work station. We consider the number of variables p as 1000, 1500, 2000, 2500, and 3000 with a sample size $n = p/2$. We conduct all the implementation approaches to both the single-precision and double-precision representations to cover the difference of the computing powers between the single-precision and double-precision operations on a GPU device. With the same reason described in Section II-G, we conduct the implementation approaches 11 times for each case of p, and average the execution times from the second to the last repetition. The preparation time of the implementation approaches are estimated by the difference between the first execution time and the average of the execution time from the second to the last repetition. The result of the preparation time is given below.

DLL-Ker	DLL-cuBLAS	PyCUDA	Numba	TensorFlow	PyTorch
0.2060	0.3081	0.5767	1.4245	1.3571	1.9152

The Python, with a dynamic-link library, needs the shortest preparation time among all implementation approaches, and the Numba, TensorFlow, and PyTorch relatively needs much

TABLE 2. Summary of average computation times for the FISTA for LASSO with $n = p/2$ and $\lambda = \sqrt{2\log(p)/n}$.

Precision	p	DLL-Kernel	DLL-cuBLAS	PyCUDA	Numba	TensorFlow	PyTorch
Single	1000	0.1688	0.1101	1.0769	20.4607	3.4939	0.4674
	1500	0.3912	0.1615	1.5002	31.9469	4.2838	0.6071
	2000	0.5477	0.2292	1.7451	38.7159	5.0831	0.6008
	2500	1.2687	0.3874	3.1600	64.6504	7.0468	0.8858
	3000	1.6278	0.5020	3.6284	78.9376	7.5478	0.9651
Double	1000	0.3044	0.1344	1.1761	23.3196	3.4134	0.4375
	1500	0.6314	0.2175	1.6915	37.0458	4.2190	0.5856
	2000	0.8464	0.3035	1.8962	48.4946	4.6181	0.6048
	2500	2.0021	0.5916	3.7435	86.6629	6.8246	0.9334
	3000	2.8138	0.7992	4.7909	112.6555	6.8384	1.0992

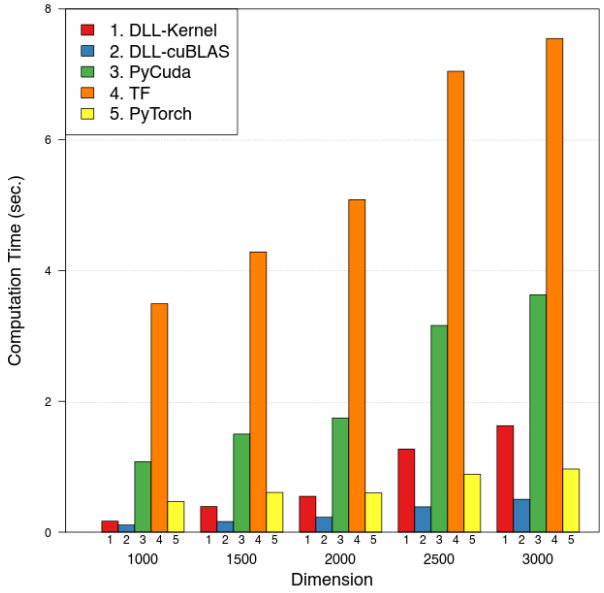

FIGURE 3. Average computation times of the FISTA for LASSO with single-precision.

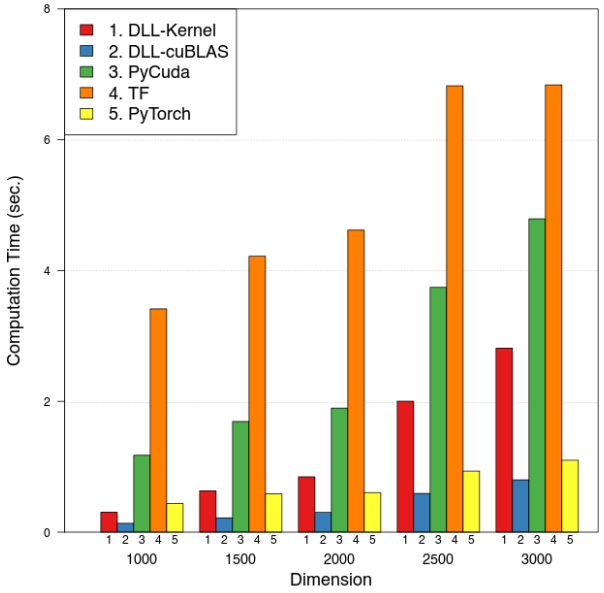

FIGURE 4. Average computation times of the FISTA for LASSO with double-precision.

more preparation time. These findings are based on the fact that the Numba library requires the JIT compilation at the first execution, and the TensorFlow and PyTorch require the creation of computational graphs for operations.

Figures 3 and 4 depict the average computation times of the implementation approaches except the Numba library with single-precision and double-precision representations. We omit the computation times of the Numba library in Figures 3 and 4 as it requires significantly more computation time than the others. To provide more precise comparison, we report the average computation times of all implementation approaches in Table 2. As used in Section II-G, we denote the Python with a dynamic-link library using the user-defined kernel functions and the cuBLAS library as DLL-Kernel and DLL-cuBLAS, respectively. For both single-precision and double-precision operations, the DLL-cuBLAS outperforms the others. This is because the DLL-based implementations conduct the main iterations with the compiled codes, and those defined by the Python-syntax conduct the main loop on Python platform. The DLL-Kernel is also more efficient and faster than the PyCUDA, Numba, and TensorFlow libraries. Despite being slower than DLL-cuBLAS, the PyTorch is the second or third best for the computation times and has an edge in terms of simplicity of implementation. If the computation time is the main issue of the target problem, the DLL-cuBLAS is the best option for implementation. Otherwise, the PyTorch is easy to implement while providing enough computational efficiency.

V. CONCLUSION

In this study, we investigated various implementation approaches for CUDA GPU parallel computation in Python with implementation guidelines and comparison results. For the ease of implementation, the existing GPU computation libraries, such as PyCUDA and PyTorch, are found be more convenient in comparison to implementing the dynamic-link library. However, for implementing the iterative algorithms, the Python with a dynamic-link library using the cuBLAS library is the most efficient in terms of the computation time. Especially, Numba is relatively restricted to implementing the iterative algorithms compared to TensorFlow and PyTorch. The implementation with PyTorch is comparable to the Python with a dynamic-link library for computational efficiency and better for ease of implementation. Although

our numerical studies have been conducted only for the FISTA for LASSO, and the comparison results could vary from the target problems, we would recommend considering either Python with a dynamic-link library using the cuBLAS library or PyTorch for CUDA GPU implementation in Python.

REFERENCES

[1] (2022). *NVIDIA, CUDA C++ Programming Guide*. Accessed: Mar. 15, 2022. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[2] C.-H. Lin, C.-H. Liu, L.-S. Chien, and S.-C. Chang, "Accelerating pattern matching using a novel parallel algorithm on GPUs," *IEEE Trans. Comput.*, vol. 62, no. 10, pp. 1906–1916, Oct. 2013.

[3] R.-B. Chen, Y. M. Tsai, and W. Wang, "Adaptive block size for dense QR factorization in hybrid CPU–GPU systems via statistical modeling," *Parallel Comput.*, vol. 40, nos. 5–6, pp. 70–85, May 2014.

[4] C.-L. Hung, Y.-S. Lin, C.-Y. Lin, Y.-C. Chung, and Y.-F. Chung, "CUDA ClustalW: An efficient parallel algorithm for progressive multiple sequence alignment on multi-GPUs," *Comput. Biol. Chem.*, vol. 58, pp. 62–68, Oct. 2015.

[5] B. Sofranac, A. Gleixner, and S. Pokutta, "Accelerating domain propagation: An efficient GPU-parallel algorithm over sparse matrices," *Parallel Comput.*, vol. 109, Mar. 2022, Art. no. 102874.

[6] Y.-G. Choi, S. Lee, and D. Yu, "An efficient parallel block coordinate descent algorithm for large-scale precision matrix estimation using graphics processing units," *Comput. Statist.*, vol. 37, no. 1, pp. 419–443, Mar. 2022.

[7] J. Gao, Y. Zhou, G. He, and Y. Xia, "A multi-GPU parallel optimization model for the preconditioned conjugate gradient algorithm," *Parallel Comput.*, vol. 63, pp. 1–16, Apr. 2017.

[8] J. Gao, Q. Chen, and G. He, "A thread-adaptive sparse approximate inverse preconditioning algorithm on multi-GPUs," *Parallel Comput.*, vol. 101, Apr. 2021, Art. no. 102724.

[9] I. Bogle, G. M. Slota, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring algorithms for distributed GPU environments," *Parallel Comput.*, vol. 110, May 2022, Art. no. 102896.

[10] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66–73, 2010.

[11] (2022). *NVIDIA, NVIDIA cuDNN*. Accessed: Mar. 15, 2022. [Online]. Available https://docs.nvidia.com/deeplearning/cudnn/pdf/cuDNN-Developer-Guide.pdf

[12] M. Abadi. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Accessed: Mar. 15, 2022. [Online]. Available: https://www.tensorflow.org

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, and A. Desmaison, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 8024–8035. Accessed: Mar. 15, 2022. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[14] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation," *Parallel Comput.*, vol. 38, no. 3, pp. 157–174, Mar. 2012.

[15] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM J. Imag. Sci.*, vol. 2, no. 1, pp. 183–202, 2009.

[16] R. Tibshirani, "Regression shrinkage and selection via the lasso," *J. Roy. Statist. Soc., B, Methodolog.*, vol. 58, no. 1, pp. 267–288, 1996.

[17] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT compiler," in *Proc. 2nd Workshop LLVM Compiler Infrastruct. (HPC–LLVM)*, 2015, pp. 1–6.

[18] Python Software Foundation. (2022). *Python 3.10.3 Documentation*. Accessed: Mar. 15, 2022. [Online]. Available: https://docs.python.org/3/

[19] (2022). *NVIDIA, cuBLAS Library*. Accessed: Mar. 15, 2022. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf

[20] (2020). *Anaconda Software Distribution, Anaconda Documentation*. Accessed: Mar. 15, 2022. [Online]. Available: https://docs.anaconda.com/

[21] A. Klöckner. (2021). *Pycuda 2021.1 Documentation*. Accessed: Mar. 15, 2022. [Online]. Available: https://documen.tician.de/pycuda/

[22] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A MATLAB-like environment for machine learning," in *Proc. BigLearn, NIPS Workshop*, 2011, pp. 1–6.

[23] I. Daubechies, M. Defrise, and C. De Mol, "An iterative thresholding algorithm for linear inverse problems with a sparsity constraint," *Commun. Pure Appl. Math.*, vol. 57, no. 11, pp. 1413–1457, Aug. 2004.

[24] Y. E. Nesterov, "A method for solving the convex programming problem with convergence rate $O(1/k^2)$), *Dokl. Akad. Nauk SSSR*, vol. 269, no. 3, pp. 543–547, 1983.

[25] T. T. Wu and K. Lange, "Coordinate descent algorithms for lasso penalized regression," *Ann. Appl. Statist.*, vol. 2, no. 1, pp. 224–244, Mar. 2008.

[26] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, "Least angle regression," *Ann. Statist.*, vol. 32, no. 2, pp. 407–499, 2004.

[27] R. J. Tibshirani and J. Taylor, "The solution path of the generalized lasso," *Ann. Statist.*, vol. 39, no. 3, pp. 1335–1371, 2011.

[28] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[29] D. Eddelbuettel and R. Francosis, "RCPP: Seamless R and C++ integration," *J. Stat. Softw.*, vol. 40, pp. 1–18, Apr. 2011.

[30] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 31–39, Mar. 2011.