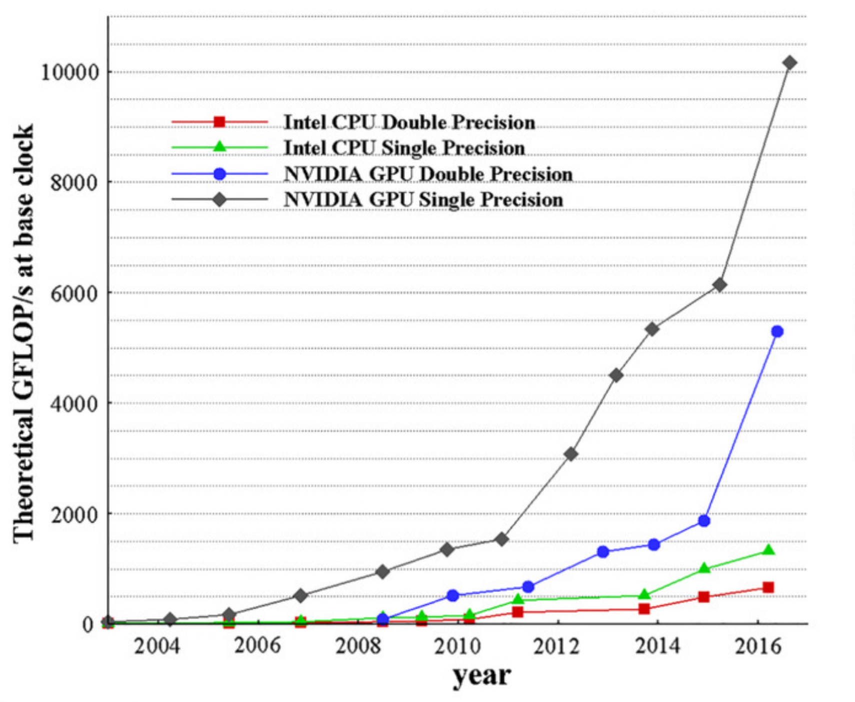
Comparison study for implementation efficiency of CUDA GPU parallel computation with the fast iterative shrinkage-thresholding algorithm

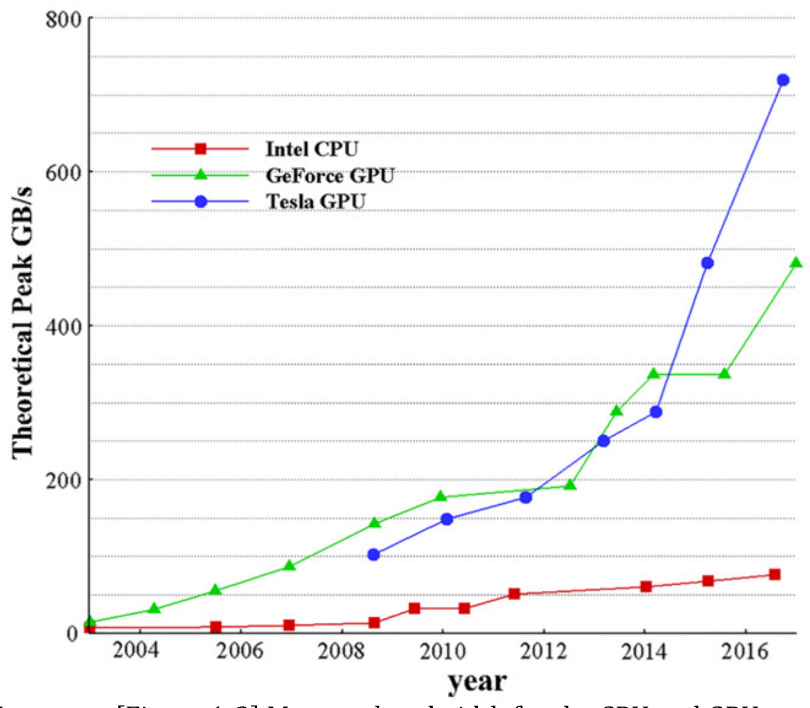
Younsang Cho, Donghyeon Yu
Department of Statistics, Inha university

Introduction

- Parallel computation using graphics processing units (GPUs) gets much attention and is efficient for single-instruction multiple-data (SIMD) processing.
- Theoretical computation capacity of the GPU device has been growing fast and is much higher than that of the CPU nowadays (Figure 1).
- There are several platforms for conducting parallel computation on GPUs using compute unified device architecture (CUDA) developed by NVIDIA. (Python, PyCUDA, Tensorflow, etc.)
- However, it is unclear what platform is the most efficient for CUDA.



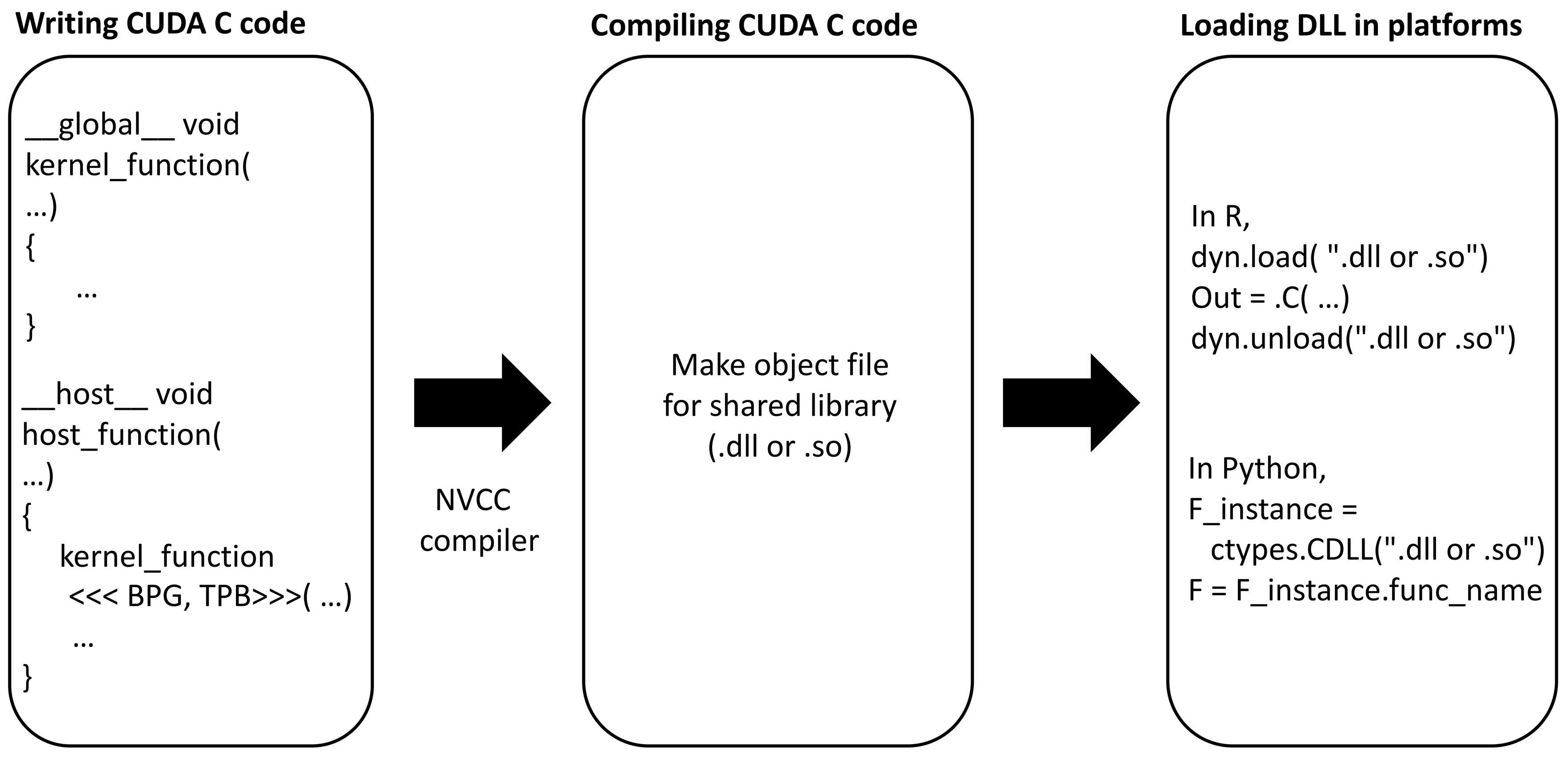
[Figure 1-1] Floating-point operations per second for the CPU and GPU



[Figure 1-2] Memory bandwidth for the CPU and GPU

Basic Implementation Procedure for Kernel function with CUDA C

We introduce a procedure for using CUDA with R and Python (Figure 2).



* BPG: Blocks Per Grid; TPB: Threads Per Block

[Figure 2] Procedure for using CUDA C extensions in platforms R and Python

1) Writing CUDA C code
We implement algorithms to C extensions which are used for CUDA kernel functions.

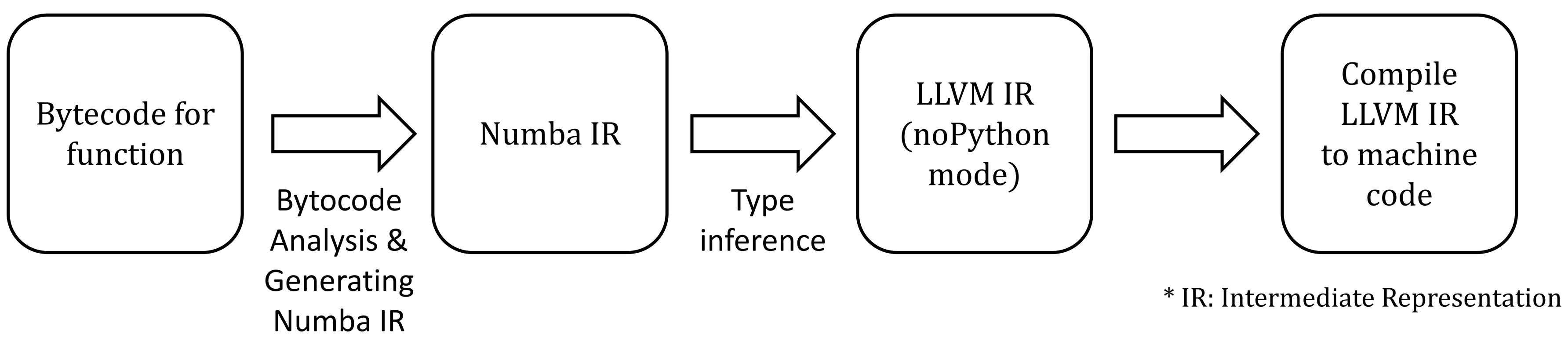
2) Compiling CUDA C code
We compile CUDA C code with NVCC compiler to object file, which can be loaded in platforms.
ex) nvcc add.C –o add.so --shared -Xcompiler –fPIC –lcublas
–gencode arch=compute_75,code=sm_75

3) Loading DLL (Dynamic Link Library) in platforms (R and Python)
After compiling CUDA C code, we call the functions written in C language by using the function dyn.load and .C in R or ctypes.CDLL in Python.

Platforms

We compared seven implementation methods for CUDA kernel functions. Among those, two implementation methods are used for just-in-time (JIT) compilation and neural network, not CUDA kernel functions.

1. Numba on CPU in Python (Numba-CPU)
Numba is used by compiling with just-in-time. Hence it is much faster than general functions defined by users in Python. In Figure 3, we represent how to work Numba for our algorithm internally.

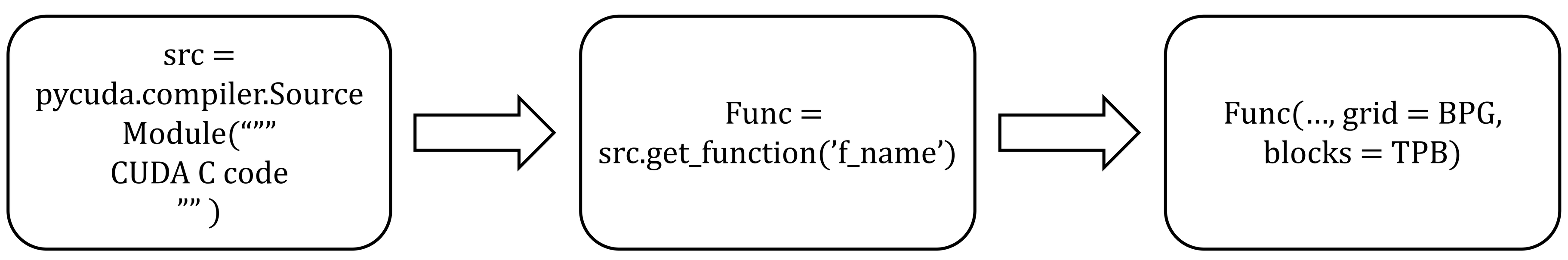


* IR: Intermediate Representation

[Figure 3] Compilation process in Numba

2. Numba on GPU in Python (Numba-GPU)
Numba can be easily able to define the CUDA kernel function by "@cuda.jit" decorator or functions executed on the GPU using CUDA by "@vectorize" or "@guvectorize" decorator with a "target = 'cuda' " argument.

3. PyCUDA in Python (PyCUDA)
In PyCUDA, it is still needed to write a C code but not to compile in command lines. CUDA C code is compiled in Python with PyCUDA compiler, and the kernel function in CUDA C code can be called as function's name (Figure 4).



[Figure 4] Procedure for using PyCUDA in Python

4. TensorFlow functions in Python (TF-F)
There are some functions executed on GPU in TensorFlow. So, we implemented our algorithm just using that functions.

5. Neural network with TensorFlow in Python (TF-NN)
Neural network model is flexible, and the LASSO problem can be represented as a simple neural network with an ℓ_1-regularized loss function

6. Using dynamic link library in Python (P-DLL)
As mentioned before, we can load DLL files, which are written in CUDA C, using "ctypes.CDLL" that is a built-in function in Python.

7. Using dynamic link library in R (R-DLL)
We can also load DLL files, which are written in CUDA C, using "dyn.load" in R.

FISTA (Fast Iterative Shrinkage-Thresholding Algorithm)

We consider FISTA (Beck and Teboulle, 2009) with backtracking as the following:

Step 0. Take $L_0 > 0$, some $\eta > 1$, and $\boldsymbol{x}_0 \in \mathbb{R}^n$. Set $\boldsymbol{y}_1 = \boldsymbol{x}_0, t_1 = 1$.

Step k. $(k \geq 1)$ Find the smallest nonnegative integers i_k such that with $\bar{L} = \eta^{i_k} L_{k-1}$

$$F\left(p_{\bar{L}}(\boldsymbol{y}_k)\right) \leq Q_{\bar{L}}(p_{\bar{L}}(\boldsymbol{y}_k), \boldsymbol{y}_k).$$

Set $L_k = \eta^{i_k} L_{k-1}$ and compute

$$\boldsymbol{x}_k = p_{L_k}(\boldsymbol{y}_k),$$
$$t_{k+1} = \frac{1 + \sqrt{1 + 4t_k^2}}{2},$$
$$\boldsymbol{y}_{k+1} = \boldsymbol{x}_k + \left(\frac{t_k - 1}{t_{k+1}}\right)(\boldsymbol{x}_k - \boldsymbol{x}_{k-1}).$$

Numerical study

- Problem
FISTA algorithm finds the solution x for the following minimization with ℓ_1-norm penalty:

$$\min \frac{1}{2} \| Ax - b \|_2^2 + \lambda \| x \|_1$$

- Simulation setting
  - Dimensions: $n = \{500, 1000, 2500\}$, $p = \{2500, 5000, 10000\}$
  - Lambda: $\lambda = 0.5 \cdot \sqrt{\frac{2\log(p)}{n}}$ for each dimensions
  - Precision: single precision, double precision

[Table 1] Summary of computation times (sec.) for FISTA algorithm in single precision. Numbers in parenthesis denote the standard errors.

n	p	NB-CPU	NB-GPU	TF-F	PyCUDA	P-DLL	P-DLL-32	TF-NN
500	2500	1.4335 (0.0824)	189.4049 (8.0439)	1.2041 (0.0979)	1.0607 (0.0791)	0.4627 (0.0318)	**0.2607** (0.0180)	9.1716 (1.4541)
500	5000	3.4089 (0.1476)	453.4099 (16.6947)	1.3547 (0.1040)	1.4388 (0.0641)	0.9801 (0.0323)	**0.4903** (0.0168)	0.9559 (0.0336)
500	10000	9.4199 (0.2921)	1221.0449 (47.0390)	1.7114 (0.1163)	1.9640 (0.1046)	2.4081 (0.1125)	**0.9838** (0.0454)	1.0620 (0.0271)
1000	2500	3.3298 (0.1646)	487.8102 (28.2207)	1.4256 (0.0954)	1.2855 (0.0688)	0.5747 (0.0300)	**0.3682** (0.0199)	11.5588 (3.4998)
1000	5000	8.4049 (0.3076)	1127.5282 (50.9946)	1.5681 (0.1405)	1.6981 (0.0969)	1.1080 (0.0469)	**0.5904** (0.0250)	1.4385 (0.0300)
1000	10000	18.5007 (5.9672)	2547.7037 (818.9664)	1.7711 (0.0838)	2.3873 (0.0976)	2.4160 (0.7544)	**1.0615** (0.3329)	1.6393 (0.0325)
2500	2500	13.7332 (2.0433)	1756.7701 (246.6578)	2.0224 (0.2719)	2.0401 (0.2414)	0.8658 (0.1200)	**0.6953** (0.0952)	70.7893 (16.7180)
2500	5000	25.2819 (1.0731)	3251.5472 (189.6372)	1.6998 (0.5224)	2.0548 (0.6243)	1.4155 (0.0511)	**1.0136** (0.0412)	2.8745 (0.0473)
2500	10000	62.0108 (2.4768)	8388.9189 (252.8839)	2.2380 (0.1086)	3.8747 (0.1435)	3.3655 (0.1196)	**2.0873** (0.0764)	3.3711 (0.0171)

[Table 2] Summary of computation times (sec.) for FISTA algorithm in double precision. Numbers in parenthesis denote the standard errors.

n	p	NB-CPU	NB-GPU	TF-F	PyCUDA	P-DLL	P-DLL-32	TF-NN	R-DLL	R-DLL-32
500	2500	1.5076 (0.0729)	154.4324 (7.3262)	1.1854 (0.0946)	1.2187 (0.0655)	0.5800 (0.0274)	**0.3797** (0.0182)	10.2749 (1.6698)	0.4975 (0.0183)	0.4288 (0.0183)
500	5000	3.7415 (0.1651)	363.5905 (13.5200)	1.4027 (0.0972)	1.6198 (0.0785)	1.1849 (0.0428)	0.6627 (0.0248)	1.0678 (0.0288)	0.8035 (0.0305)	**0.6557** (0.0305)
500	10000	9.7471 (0.5415)	942.6866 (38.3124)	1.5280 (0.0905)	2.4830 (0.1643)	3.2228 (0.1312)	**1.4536** (0.0607)	1.2341 (0.0332)	1.6894 (0.0510)	1.3028 (0.0510)
1000	2500	3.6231 (0.1949)	370.2242 (27.8036)	1.3661 (0.0881)	1.4257 (0.0917)	0.7021 (0.0521)	**0.4793** (0.0350)	13.0257 (2.9117)	0.6539 (0.0219)	0.5296 (0.0219)
1000	5000	8.6451 (0.4510)	827.2198 (29.1598)	1.6186 (0.1117)	1.9771 (0.1233)	1.3521 (0.0463)	**0.8005** (0.0280)	1.6580 (0.0257)	1.0519 (0.0433)	0.8314 (0.0433)
1000	10000	19.0893 (6.1287)	1865.1549 (597.4873)	1.8941 (0.0932)	2.8716 (0.1087)	3.2657 (1.0222)	**1.5667** (0.4888)	1.9429 (0.0126)	2.0693 (0.0451)	1.6067 (0.0451)
2500	2500	16.0259 (2.2434)	1300.2521 (181.5767)	1.9295 (0.2802)	2.2760 (0.3244)	1.1369 (0.1589)	0.8461 (0.1161)	102.4257 (26.4692)	0.9879 (0.0754)	**0.8194** (0.0754)
2500	5000	27.9770 (1.3787)	2496.0400 (48.1004)	1.6181 (0.4924)	2.2523 (0.6907)	1.8553 (0.0363)	**1.2462** (0.0242)	3.4139 (0.0437)	1.4977 (0.0444)	1.3005 (0.0444)
2500	10000	73.0963 (3.1649)	6329.5722 (133.6182)	**2.5480** (0.0751)	4.4759 (0.1007)	4.6840 (0.0990)	2.7530 (0.0573)	4.2218 (0.0482)	3.1150 (0.0811)	2.7229 (0.0811)

- "P-DLL-32" denotes the Python with a dynamic link library using 32 threads per block in CUDA C and "R-DLL-32" denotes that is similar to "P-DLL-32" in R.
- "P-DLL-32" is the fastest platform on overall simulation settings.
- In high dimensional setting, the Numba platforms are not efficient for FISTA.

Conclusions

- In general, the python with a dynamic link library which is written in CUDA C is the most efficient platform on GPU parallel computing.
- In the high dimensional case, there is not much difference between "P-DLL-32" and "TF-F" platform.
- We recommend python or R with a dynamic link library for GPU parallel computation if researchers are familiar with python or R and CUDA C.
- If researchers are only familiar with python, the python with TensorFlow functions is an alternative for the efficient implementation of GPU parallel computation

References

[1] Abadi, M., Barham. P, Chen. J, et al. (2016). Tensorflow: A system for large-scale machine learning, *12th USENIX Symposium on Operating Systems Design and Implementation(OSID), USENIX Association*, 265--283
[2] Beck, A. and Teboulle, M. (2009). A fast iterative shrinkage-thresholding algorithm for linear inverse problems, *SIAM journal on imaging sciences*, **2**, 183—202.
[3] Cho, Y., Yu, D., Son, W., Park, S. (2020). Introduction to Numba library in Python for efficient statistical computing . *The Korean Journal of Applied statistics*, **33**(6), 665—682.
[4] Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A. (2012). PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation, *Parallel Computing*, **38**(3), 157--174
[5] Lei, J. Li, Dl., Zhou, Yl., et al (2019). Optimization and acceleration of flow simulations for CFD on CPU/GPU architecture. *J Braz. Soc. Mech. Sci. Eng*. **41**, 290 . https://doi.org/10.1007/s40430-019-1793-9
[6] S.K. Lam, A. Pitrou, and S.Seibert. (2015). Numba: A LLVM-based Python JIT compiler, *Proc. 2nd Workshop LLVM Compiler Infrastructure HPC*, **7**, 1—6