尾道市立大学経済情報論集　第17巻第１号　2017年７月　抜刷

GPU による汎用計算について

Getting Started with GPGPU

菅　　準　一

研究ノート

GPU による汎用計算について

菅　　準一

CPU (Central Processing Unit) の処理能力は飛躍的に伸びたが，解きたい問題によっては，その処理時間のさらなる短縮を望む事態が生じる。場合によっては，プログラムの一部に並列処理を用いることで処理時間を短縮できる。しかしクロック数が高くてコア数とスレッド数の多い CPU の導入には，多額の費用がかかり実現が困難な状況がある。そこで，数値計算の用途に BTO (Build To Order) したものではないが，マルチコア CPU のパソコンに，CPU と比較すると安価な (GPGPU 専用の Tesla ではない）GPU (Graphics Processing Units) を導入することで，処理時間の短縮を図る GPGPU (General-Purpose computing on Graphics Processing Units) つまり GPU による汎用計算について考えてみる。

キーワード：GPGPU, CUDA, Python, Ubuntu, Docker

目次

1　はじめに

最近まで CPU の処理能力は飛躍的に伸びてきたが，解きたい問題によっては，その処理時間のさらなる短縮を望む事態が生じている。場合によっては，プログラムの一部に並列処理を用いることで処理時間を短縮できる。しかしクロック数が高くてコア数とスレッド数の多い CPU の導入には，多額の費用がかかり実現が困難な状況がある。そこで，数値計算の

用途に BTO したものではないが，マルチコア CPU のパソコンに，CPU と比較すると安価な (GPGPU 専用の Tesla ではない）GPU を導入することで，処理時間の短縮を図る GPGPU つまり GPU による汎用計算について考えてみる。

GPGPU を実現するためには，GPGPU に対応する GPU が必要である。GPU を積んだグラフィックカードの２大メーカーである AMD 社と NVIDIA 社の製品であれば対応している。ここでは比較的安価なグラフィックカードである NVIDIA 社の GeForce 1060 (3G) を使って Ubuntu 16.10 上で GPGPU を体験してみる。今回用いた PC は Windows 10 Home がプレインストールされ販売されているもので，メインメモリを 64G に増やしている。以前メモリが 16G の Windows 10 パソコンで Python を使って数値計算しているとき，ユーザーメモリを使い切ってしまい，パソコンがカクカクとしか反応しなくなったのでメモリを増強している。また Ubuntu 16.10 を後からインストールして，Windows10 と Ubuntu16.10 をデュアルブートできるようにしている。Ubuntu 16.10 をインストールする際に，Windows 10 のファストブート機能は無効化して，起動時に Del キーの連打で UEFI を呼び出せるようにしている。ただ，UEFI のセキュアブートは残したままであった。

また，なぜ長期的にサポートされる Ubuntu 16.04 LTS でないかというと，Ubuntu 16.04 LTS 日本語 Remix の iso イメージのグラフィックドライバが古くて GeForce 1060 に対応していないこと，マザーボードのチップセットが Intel X99 で CPU ソケットが LGA2011-3 であり，グラフィック機能のない CPU が載っていること。ディスプレイへの表示も外付けの GeForce 1060 ボードによるので，Ubuntu 16.04 LTS 日本語版 Remix iso イメージから作った Live USB で起動しても，ディスプレイに何も表示されない。Ubuntu 16.10 は，サポート期間が９か月と短いが，Ubuntu 16.10 日本語版 Remix の iso イメージから作成した Live USB から起動すると，ディスプレイに表示できたので，Ubuntu 16.10 をインストールして用いている [1)]。

---

1) ただデュアルブートにしなくても，Windows 10 HOME 64 上でも Ubuntu を動かすことができる。一つは VirtualBox など Virtual 環境に Ubuntu をインストールして使う。Network インストールできる通信環境が整い，また Virtual マシーンに割ける PC のメモリやグラフィックメモリ，ディスク容量の増加に伴い快適に動くようになった。もう一つは Windows10 Anniversary Update に含まれる形で提供されるようになった WSL (Windows Subystem for Linux) だ。WSL を有効にして，セキュリティレベルを開発者モードに設定することで，Ubuntu がダウンロードされて，WSL として Ubuntu を動かすことができる。最初は CLI で面食らうと思うが，X Windows System や軽量デスクトップをインストールすると GUI 環境が構築できて，FireFox を起動できる。ただ日本語が入力できないなど面白いことが起こり元々は英語圏で開発されたものであることを痛感することになる。上記の Virtual 環境や WSL のいずれにおいても，GPGPU 環境を構築して良いパーフォーマンスを生み出せないと感じた。そこで，Mac ユーザーや Linux ユーザーにとってはお馴染みの Windows とのデュアルブート環境の構築である。昨年 Microsoft 社が Linux Foundation の法人会員として加入したことよって，Linux の勢力図に大きな変化が起こるかもしれない。

GPGPU するために NVIDIA のドライバと CUDA をインストールする必要がある[2]。Ubuntu を通常インストールすると使える NVIDIA の GPU 向けオープン・ソース・ソフトウェアのグラフィクス・ドライバ (nouveau) は，ディスプレイに表示させる本来の機能とは別の機能である GPGPU に対応するようには作られていない。そこで，NVIDIA が Linux 向けに提供しているグラフィックス・ドライバと GPU に汎用計算を行わせる開発環境「CUDA」をインストールする必要がある。UEFI のセキュアブートをオフにしておき，標準の nouveau ドライバを無効化して，DKMS パッケージをインストールして，Ubuntu の公式リポジトリから GeForce 1060 対応ドライバをインストールする[3]。ドライバと CUDA Toolkit がインストールできたら，GPU が認識されているか確認のためにターミナルで

nvidia-smi

コマンドを実行して，図１のような表示が出ると良い。GPU の ID は 0, 名前は GeForce GTX 106..., 温度は 27C などが表示されている。PCI Bus ID が 0000:03:00.0 メモリが 3010MiB (メビバイト)

```
kt@kt-X99-S04A:~$ nvidia-smi
Fri Apr 14 11:45:56 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 375.39                 Driver Version: 375.39                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 106...  Off  | 0000:03:00.0      On |                  N/A |
| 38%   27C    P8     6W / 120W |    206MiB /  3010MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|    0      1506    G   /usr/lib/xorg/Xorg                             162MiB |
|    0      3256    G   /usr/bin/compiz                                 39MiB |
|    0      4669    G   /usr/lib/firefox/firefox                         1MiB |
+-----------------------------------------------------------------------------+
```

さらに，CUDA Tookit　がうまくインストールできているか確認するため，CUDA のサンプルが動くかどうか確認する。標準的なインストールだと CUDA のサンプルは /usr/local/cuda-8.0/NVIDIA CUDA-8.0 Samples ディレクトリにある。1 Utilities/deviceQuery ディレクトリに移動して

---

2) CUDA ではなくて OpenCL を使うという選択肢もある。その他にも新しい技術が登場しているので，それらを使うのもいいと思う。今回は CUDA を使ってみる。

3) うかつにも，このような手順を踏まずにインストールしようとして，インストールが途中で止まっていることを確認せずに電源を落としてしまって，再起動したら GUI でログインできなくなってしまた。うっかりして，そんなことになったら，Ctrl+Alt+F1 で GUI を抜けて CLI モードに入り，ログインして NVIDIA 関連のファイルを purge してやり直すとよい。私の場合は Windows 10 とデュアルブートなので，再起動して Grub の画面で Windows10 を起動し，必要な情報を手に入れたが，ノートパソコンやスマホを手元に置いておくとよい。また，Windows10 も起動しない最悪の場合を考えて，LiveUSB も手元に用意しておくとよい。

sudo make

コマンドで deviceQuery のバイナリを作り

./deviceQuery

を実行すると，以下のような表示がされる [4]。

```
kt@kt-X99-S04A:/usr/local/cuda-8.0/NVIDIA_CUDA-8.0_Samples/1_Utilities/deviceQuery$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 1060 3GB"
  CUDA Driver Version / Runtime Version          8.0 / 8.0
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 3011 MBytes (3157131264 bytes)
  ( 9) Multiprocessors, (128) CUDA Cores/MP:     1152 CUDA Cores
  GPU Max Clock rate:                            1709 MHz (1.71 GHz)
  Memory Clock rate:                             4004 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16
84)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 3 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 8.0, CUDA Runtime Version = 8.0, NumDevs = 1,
Device0 = GeForce GTX 1060 3GB
Result = PASS
```

2 Python で GPGPU

CUDA を使って GPGPU のためのプログラムを直接書いてもよいが，Python で GPGPU することを考える。そこで，Python の環境を整えるために，Continum Analytics の Anaconda をインストールする [5]。Python で CUDA のプログラムを書けるものとして，まず頭に浮かぶの

---

4) ただ，残念なことにその他のサンプルでは，「gcc5 以上には対応していない」というエラーがでる . とりあえずインストールされている gcc6 を downgrade するために gcc-4.9 と g++-4.9 をダウンロード・インストールして , それぞれを /usr/local/ cuda/ bin/gcc と / usr / local /cuda / bin / g++ にリンクさせると make できる。

5) Anaconda のダウンロードページの中ほどにある Download for Linux タブをクリックして左側に書かれた３つの指示に従ってインストールするとよい。右側にある Python 3.6 version の 64-BIT INSTALLER（緑色）をクリックするとダウンロードが始まる。ダウンロードしたファイルの MD5 or SHA-256 でファイルが壊れたり改ざんされていないかチェックする。terminal で bash Anaconda3-4.3.1-Linux-x86 64.sh をタイプして指示に従う。ファイル名が長いので Tab キーによる補完機能を使うと間違いがない。terminal で python と打ち込んで Python と Anaconda のバージョンが表示されればインストールが成功したことになる。Anaconda をインストールすると，package manager と environment manager の役割を果たす conda も一緒にインストールされているので，まずは conda 自体を conda update conda で update する。

は PyCUDA だろう。しかしながら，ネットで少し調べてみると，CuPy というものがあることが分かった。これは Numpy の GPU 版という位置づけのもので，Chainer という Deep Learning 用の Python パッケージで GPU バックエンドとして動いている。Chainer をインストールすると自動的にインストールされる。そこで , まず Chainer がインストールされているかどうか確かめる。

python -c ”import chainer ”をターミナル上で実行するとエラーが出ることから，インストールされていないことが分かる。そこで，conda でインストールできるか確かめる。conda search chainer を実行しても表示されないので，conda ではインストールできないことが分かる。conda コマンドでインストールできない場合は pip を使ってインストールすることになる。

pip install chainer

インストールがうまくいったので，MNIST example を試す。

python train mnist.py -g=0

で gpu を使う設定で計算させると，53 秒ほどで計算が終了した。CPU だけの場合は 225 秒ほどなので，CPU だけでも待てないほどの処理時間ではないけれど，GPU を使うと 4 倍くらい計算が早くなっている。GPU を高性能なものに替えると処理をさらに高速化可能であると思われる [6)]。

3 OpenSSH で Windows ノートから GPGPU

GPGPU のメリットはわかるが，普段使っている Windows ノートには GPU は入っていない。普段持ち歩いている Windows ノートで GPGPU できないだろうか。一つの解決法として GPU がついているデスクトップ PC に，ユーザーとして登録してもらい，インターネットを通じてログインして Windows ノートでデスクトップ PC を操作することが考えられる。インターネットを通じて離れた場所から PC を使うには，データが盗み見られても大丈夫なようにしておく必要がある。そのためには，通信内容を暗号化することが考えられるが，一般に普及しているのは OpenSSH を用いた暗号通信である。そこで GPU のついた PC（ホスト）に Ubuntu 用の SSH サーバーをインストールし，Windows ノートに Windows 用 OpenSSH をインストールして SSH クライアントとしてホストにログインできるようにする。Ubuntu PC を SSH サーバにして，SSH クライアントの Windows ノートで作成した公開鍵を

6) Anaconda でインストールした Python のバージョンは 3.6 でなぜか NVIDIA の cuDNN (Deep Neural Network library) が enable にならないエラーが出たので，再インストールしたがうまくいかなかった。そこで一度頭を切り替えて，conda の environment management 機能を使って，Chainer 用の Python 3.5.1 の環境を作成してインストールすると，import cupy.cudnn のエラーが消えて cuDNN の enable エラーも表示されなくなった。しかし残念なことに処理速度に変わりはなかった。cuDNN をダウンロードするには NVIDIA の Accelerated Computing Program のメンバーになることが必要である。

受け取り登録する簡単な手順は以下の通りである。

1. Ubuntu PC に SSH をインストール

   `sudo␣apt-get␣install␣ssh`

   さらに，この PC の ip アドレスを確認する

   ifconfig

2. Windows ノートに OpenSSH をインストールする。

   OpenSSH のダウンロード・ページからダウンロードした 64bit 用の OpenSSH-Win64.zip ファイルを適当な場所（ディレクトリ）に展開する。ディレクトリに Path を通す。

3. 暗号化に使う鍵を Windows ノートで作成する

   ssh-keygen コマンド (`.\ssh-keygen.exe`) を使って

   public/private rsa key を作る。

   作ったキーを保管するファイルを尋ねられるが，デフォルトのままでよいので，そのまま Enter キーを押す，更に passphrase を入力するよう求められるので，入力する。この passphrase が login に必要なので書き留めておく。すると，デフォルトの場所に id_rsa（秘密鍵）と id rsa.pub（公開鍵）のファイルが作られる。

4. 公開鍵ファイル id_rsa.pub をホストに送って認可鍵として保管してもらう

   この公開鍵をサーバーに登録すれば暗号化通信ができる。Windows ノート（クライアント）から Ubuntu PC（ホスト）に向けて公開鍵を scp コマンドで送る

   `scp␣id_rsa.pub␣server-ip-address:~/`

5. ホストの Ubuntu PC では，送られてきた公開鍵ファイル id_rsa.pub を認可鍵ファイル authorized.keys に追加する

   鍵を保管するディレクトリを作る (sudo で一時的に root になる)

   `sudo␣mkdir␣.ssh`

   `sudo␣cat␣id_rsa.pub␣>>␣.ssh/authorized.keys`

   これで Windows ノート（クライアント）から SSH を用いた暗号化通信ができて，ホストの GPGPU を体験できる。

6. Windows ノートのコマンドプロンプトで

   `ssh␣user@host-ip-adress`

   鍵の作成時に passphrase を設定したので，そのとき設定した passphrase を入力する

   user として Ubuntu PC のキーボードから直接ログインするときの user password ではないので注意すること。また Ubuntu PC の文字コードが UTF-8 なので，Windows のコマンドプロンプトからログインすると文字化けする場合がある。文字化けしたら，一度 exit で接続を切って，

chcp 65001

を実行して，コマンドプロンプトの文字コードを UTF-8 に変更してから，再接続するとよい。

4 Ubuntu の Docker と LXD を利用する

Mac を使っていると 2001 年の Mac OSX 発売以来毎年 OS の新しいバージョンが発売され，新しくなるたびに新しい OS を購入してインストールするということが続いていて，ソフトウェアによると新しいバージョンの OS では動かないという事態が発生する。そこで USB の外付け HDD に，PC 本体とは異なるバージョンをインストールして，起動ディスクを使い分けることで対応してきた。いまも最新の macOS (Sierra) を外付けの SSD にインストールして，MAMP で moodle の最新版を使い STACK を利用する試験運用に使っている。名刺入れより小さい 256G の SSD で試験運用環境をポケットに入れて持ち歩くことができる。ただ，35G とサイズが大きいのが問題だ。MAMP に加えて moodle や STACK, Maxima, Gnuplot, Xterm など，必要なアプリをダウンロードして設定したものだけに絞って軽量化して，適当なバージョンの OS が起動しているサーバやデスクトップに簡単に受け渡し配備できないだろうか。

一台の物理マシーン（ハードウェア）上に Virtual マシーン（ファイル）を複数インストールして，macOSX や Ubuntu，Windows など異なる OS を使い分けたり，同じ OS でも違うバージョンを使ったり，64 ビットの物理マシーン上に 32 ビットの仮想マシーンを動かすことができるなど，仮想環境は一台の物理マシーンを有効に使うために重要な役割を担ってきた。そのような仮想化技術は，ホスト OS 型とハイパーバイザー型 , に分けることができる。ホスト OS 型は例えば Mac OSX（ホスト OS）が動いているマシーンに VirtualBox などの仮想化ソフトをインストールして仮想マシーンを立ち上げて，たとえば Windows や Ubuntu などの OS（ゲスト OS）を動かす。このときゲスト OS を動かすためには，物理マシーンの上にホスト OS と仮想化ソフトの２つの層が必要である。他方ハイパーバイザー型の仮想化技術では，複数の OS を動かすためのハイパーバイザーと呼ばれる仮想化のための１層が物理マシーンの上に作られる。最近注目を浴びているコンテナ型の仮想化は OS とコンテナ管理ソフトウェアの上にコンテナによる仮想環境を提供する。Docker 社の Docker が最近注目されている。また，LXD は LXC が提供する API を利用したコンテナ管理コマンドやサービスで，ホストとは異なる OS をインストールしたコンテナをいくつか作って，様々な運用検証環境を整備できる。

Ubuntu のプログラムをインストールして動かすためには，Ubuntu のカーネルのバージョンとプログラムのバージョンがうまく合わないと，インストールにすら失敗してしまう。う

かつにシステムをバージョンアップしないことに注意しないといけないが，最新のデバイスを買ってしまって，最新のバージョンの Ubuntu をインストールしてしまうと様々な困難に遭遇することになる。とくに NVIDIA のデバイスドライバが関わると，一苦労する，上で記したように，NVIDIA のドライバと CUDA，cuDNN をインストールして，Chainer を動かすことに成功したが，さらに Tensorflow を , 色々な依存関係をクリアするように調整して，インストールして試してみる時間的余裕がない。そこで，Tensorflow が動く組み合わせをパッケージにしてくれたものがあれば，ありがたいと思ってネットを調べたら nvidia-docker なるものがあり，それを使って Tensorflow を簡単に利用できるようなので試してみた [7]。簡単そうでもやはり一手間が必要である。Github の nvidia-docker のリポジトリに行き Installation のページを見ると，次のような記述がある。

The list of prerequisites for running nvidia-docker is described below.

1. GNU/Linux x86 64 with kernel version > 3:10
2. Docker>= 1:9 (official docker-engine only)
3. NVIDIA GPU with Architecture> Fermi (2.1)[8]
4. NVIDIA drivers >= 340:29 with binary nvidia-modprobe

Your driver version might limit your CUDA capabilities (see CUDA requirements)

さらに CUDA toolkit version 8.0 を利用する場合には，Driver version が 367.48 以上でないといけないと書いてあった。

kernel のバージョンは uname -rv コマンドで表示できる。バージョンは 4.8 なので十分である。

NVIDIA GPU は GeForce 1060 (3G) で Pascal Architecture なので十分である。NVIDIA ドライバーのバージョンは nvidia-smi コマンドで確認すると，375.39 であり十分である。残るは

---

7) nvidia-docker については，https://github.com/NVIDIA/nvidia-docker/wiki/nvidia-docker に説明がある。nvidia-docker is a thin wrapper on top of docker and act as a drop-in replacement for the docker command line interface. This binary is provided as a convenience to automatically detect and setup GPU containers leveraging NVIDIA hardware. Internally, nvidia-docker calls docker and relies on the NVIDIA Docker plugin to discover driver files and GPU devices. Note that nvidia-docker only modifies the behavior of the run and create Docker commands. All the other commands are just pass-through to the docker command line interface. As a result, you can't execute GPU code when building a Docker image.If the nvidia-docker-plugin is installed on your host and running locally, no additional step is needed. nvidia-docker will perform what is necessary by querying the plugin when containers using NVIDIA GPUs need to be launched. したがって，nvidia-docker は docker command line interface の完全互換であり，docker コマンドの run と create だけに修正を加える。また nvidia-docker から GPU の情報を得るための nvidia-smi コマンドを使うためには，ドライバファイルや GPU を探してくれる nvidia-docker-plugin が起動していないといけないことがわかる。

8) NVIDIA の architecture には著名な科学者の名前がつけられてきた。Tesla,Fermi,Kepler,Maxwell そして昨年市場に投入された Pascal, ようやくこの３月に姿をみせた次世代の Volta という具合だ。括弧の中の数字 2.1 Compute Capability のバージョンである。Pascal archiitecture の GeForce GTX 1060 の Compute Capability は 6.1

Docker のバージョンが 1.9 以上でしかも official docker-engine only ということで，条件を満たす official docker-engine をインストールする。

Ubuntu の公式リポジトリで提供している docker.io ではなくて，Docker 社のリポジトリの docker-engine でないといけないようだが，見当たらない。Docker 社のページにある無料の Docker CommunityEdition (Docker CE) をインストールすればよいことが分かったので，ダウンロードしてインストールする。次に，プラグインである nvidia-docker をインストールしようとして，Quick start の例ににならって　dpkg -i を試してみたが，依存するファイル stv-rc と file-rc がないというエラーでインストールが止まってしまう。そこで，ソースファイルをダウンロードして make install するとエラーメッセージをはきながら，とにかくインストールができた。

`sudo␣docker␣images` で表示させると，nvidia-docker が build されている。

そこで, nvidia-docker が使えるか試しに, GPU の情報を nvidia-smi を使って表示させてみる。

`sudo␣nvidia-docker␣run␣nvidia/cuda␣nvidia-smi`

plugin not found というエラーで止まってしまう。そこで，手動でプラグインを起動することにする。

`sudo␣nvidia-docker-plugin`

これでプラグインがアクティベイトされる。

プロセスが動いているので，別のターミナルを立ち上げて

`sudo␣nvidia-docker␣run␣nvidia/cuda␣nvidia-smi`

と入力すると GPU のドライバのバージョンが 375.39 であることなどが表示され，ドライバのインストールと GPU の認識がうまくいっていることがわかる。

以上で，docker-ce と nvidia-docker がインストールできたので，つぎは nvidia-docker を使うとどれくらい容易く TensorFlow をインストールして運用できるか確かめてみる。

Tensoflow の github の Docker ディレクトリや Docker Hub から tensorflow のイメージ

gcr.io/tensorflow/tensorflow:xxx-gpu (TensorFlow with all dependecies and support for NVidia CUDA) をためす。ここで，gpu がついていることを確認する。gpu がないと，CPU だけを使うものであり GPU は使えない。xxx はバージョンである。

```
sudo␣nvidia-docker␣run␣-itd␣--name=tensorflow-xxx-gpu␣-p␣8888:8888␣
-p␣6006:6006␣gcr.io/tensorflow/tensorflow:xxx-gpu
```

うまくいくと，`sudo␣docker␣images`　や `sudo␣docker␣ps␣-a` で一覧表示させる (-a オプションをつけないと停止中の docker は表示されない）と，gcr.io/tensorflow/tensorflow : xxx-gpu のイメージファイルをもつ tensorflow-xxx-gpu という名前の docker コンテナが作られている。

そこで，この docker コンテナに入って作業をする。

```
sudo␣nvidia-docker␣exec␣-it␣tensorflow-xxx-gpu␣bash
```

docker の中に入って bash でインターラクティブ (-it) に作業ができ，プロンプトの表示が次のように変化する（ホストの時の $ から # に代わる）[9]。

```
root@docker-ID:/notebooks#
```

Github から tensorflow をクローンする。このとき，git が無いというエラーが出たら git をインストールするとよい。

```
git␣clone␣-b␣r0.10␣--single-branch␣--recurse-submodules␣https://
git.com/tensorflow/tensorflow.git
```

うまくクローンが作れたら

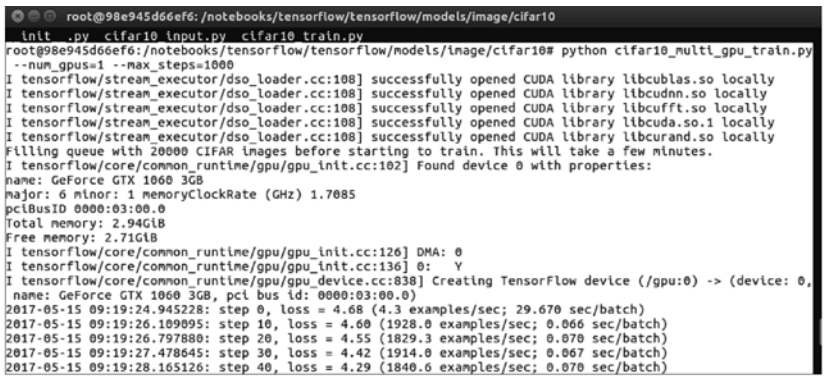
カレントディレクトリの下に tensorflow というディレクトリが作られているので，ls コマンドで確認するとよい。

cd コマンドで `tensorflow/tensorflow/models/image/cifar10/` まで移動する。このとき，Tab キーの入力補完機能を使うと間違いが少ない。

```
python␣cifar10_multi_gpu_train.py␣--num_gpus=1␣--Vmax_steps=1000
```

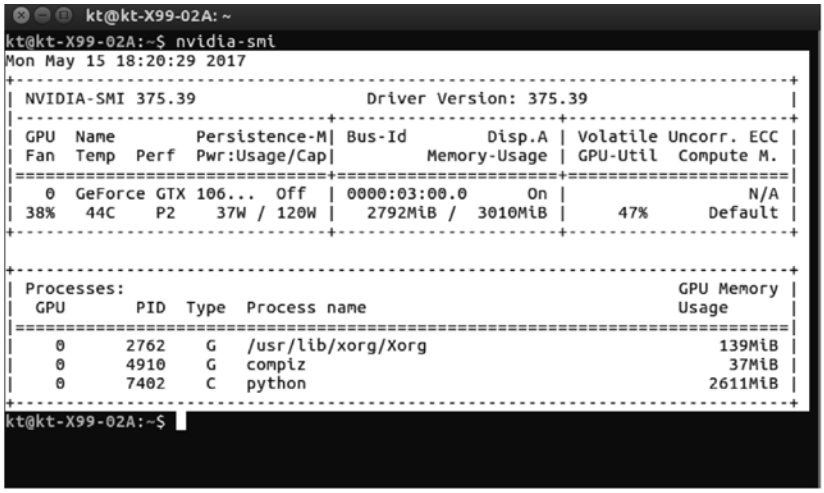
を実行する。

以下の図のように，cuda のライブラリが読み込まれていることがわかる。簡単すぎるので当惑する。



9) 一度シャットダウンして，翌日に再現しようとするときは，sudo nvidia-docker-plugin でプラグインを立ち上げておいて，別のターミナルで sudo docker start tensorflow-xxx-gpu で docker をスタートさせてからでないと，sudo nvidia-docker exec コマンドで docker の中には入れない症状が現れる場合がある。そこで，ホームディレクトリに上記のコマンドを列挙したシェルスクリプトファイルを作る。１行目にシバンと呼ばれるおまじないを書いておく。#!/bin/bash 次の行には sudo nvidia-docker-plugin & 行の最後の & は，起動したプロセスをバックグラウンドで実行させるためのもので，次の行のコマンド sudo docker start tensorflow-xxx-gpu に進み，最後の行で sudo docker exec -it tensorflow-xxx-gpu bash を実行させる。ファイルを保存したら，ターミナルで bash nvidia-docker.sh と実行して，tensorflow-xxx-gpu docker 内の notebooks ディレクトリに入っていることを確認するとよい。

このとき，プログラム実行中の GPU の状況を nvidia-smi コマンドで確認したものが，以下の図である。



GPU の温度は 44 度で, 消費電力は 37W である。GPU のメモリ 3010MiB のうち 2792MiB を使っていて，python の Process が 2611MiB 利用して計算等を行っている。Volatile GPU-Util が 47% と表示されており，メモリがボトルネックになっているのかもしれない。GeForce GTX 1060 (3G) が GeForce GTX 1060 (6G) の廉価版であることが何らかの影響を与えているのだろうか。この点は調査を要する。

ここで，tensorflow-xxx-gpu を立ち上げたときに，コンテナの中で jupyter notebook サーバが起動して 8888 ポートを使っている。このコンテナの 8888 ポートとホストの 8888 ポートをマップしている。ホストからコンテナに Firefox でアクセスするために，Firefox を立ち上げて http://localhost:8888 と入力すると jupyter notebook の Home が開く。1 hello tenssorflow.ipynb や 3 mnist from scratch などを開いてセルを実行させると，エラーもなく実行できる。

4.1 TensorBoard を使う

コンテナの tensorflow を用いて簡単な計算をして，その結果を TensorBoard で表示できる。

TensorBoard は TensorFlow とともにインストールされて，パスが通っているので，docker の中で tensorboard コマンドで起動すると，

Starting TensorBoard 41 on port 6006

(You can navigate to http://172.17.0.2:6006)

と表示される。

docker の外に出て，ホスト上で firefox などのブラウザを起動して，URL に 172.17.0.2:6006

と入力すると TensorBoard の画面が表示される。

5 Docker とはどんなもの

Docker についてもう少し知りたいと Docker 社のページをのぞいてみると，ソフトウエアのコンテナのプラットフォームだという。「僕の PC では動くけど」(”works on my machine”) と言われても困るケースがある。プログラムを明示的な合意に基づいて共同開発する場合以外にも，コンピュータプログラムを用いた先行研究を検証したり，先行研究のプログラムを利用しようとするとき，同じ環境を構築するのに時間がかかり，結局は動かなかったということがしばしば起きる。コンテナを使うとソフトウエアを動かすのに必要なすべてをコンテナにパッケージでき，どのような環境に配備しても動くようになるらしい。VM はマシーンレベルの仮想化技術であり，VM 上に OS 全部をインストール必要とするが，コンテナはソフトウエアが動くためのライブラリと設定だけで十分なのである。インストールや設定に時間を費やさなくてすみ。複雑な依存関係は Docker イメージから引っ張てくると良い。

必要に迫られて nvidia-docker を使い Tensorflow 等の GPU を利用する Python のパッケージを動くようにしたかったので，Docker に手を出したから，Docker の全体像を語るほどの知識はいまはない。ただ，初心者にとって必要最小限の情報は伝えることができると思う。

Docker を支えるものは，イメージ (image)，コンテナ (container)，サービス (sevice)，スタック (stack) とスワム (swarm) である。イメージからコンテナを作って，コンテナで作業をする。イメージは既製品と手作りの二つがある。既製品は Docker Hub をはじめとする様々なレジストリ (Registry) に保存されている色々なリポジトリの Image ファイルとして提供されている。手作りは Dockerfile という名のファイルに必要事項を書き，必要なものを同じディレクトリに集めておいて作る。したがって，作業するためのディレクトリをまず mkdir コマンドで作って，そこで作業する。手作りの際も既製品をもとに作ることが多いように思える。

Docker Hub にある既製品の Image は，username/repository:tag で指定して使うことができる。docker run コマンドで元になる Image を指定してコンテナをつくる。このとき –name= でコンテナに名前をつけることができ，-p でコンテナのポートとホストのポートを対応付けること (mapping) ができ，コンテナ内で提供される Web ベースのサービスにホストの Web Browser からこのポートを通じてアクセスできるようになる。

Get started with Docker Part 1:Orientation を見ながら Docker について理解を深めてみるとよいと思う。ただ，読み進めるためには，次のものについて予備知識があったほうがよい。

1. IP addresses and Ports
2. Virtual Machines

3. Editing configuration files

4. Basic familiarity with the ideas of code dependencies and building

5. Machine resource usage terms, like CPU percentages, RAM use in bytes,etc.

これに加えて，terminal での基本的な Linux コマンド操作 cd, ls,cp,mkdir,cat,curl. やファイル作成・編集に用いる vi などのエディターに慣れている方がよい。

Get started Part 2 : Containers を読み進んで　friendyhello イメージを作って，イメージをもとにコンテナを作り所望の結果が出た後で，Docker アカウントを作って Docker の public registry にイメージをアップするように書かれている。その部分を，ローカルマシン上にプライベートな registry を作ってそこに保存するようにしてみる。

`$sudo␣docker␣run␣-d␣-p␣5000:5000␣-v␣/var/opt:/var/lib/registry␣registry:2.3.0`

プライベートな registry を作成運用するためのイメージもたくさんあるが，public registry にあるイメージ registry のバージョン 2.3.0 を使ってプライベートレジストリのためのコンテナを作成する。-d でコンテナをバックグラウンドで走らせ，-p でコンテナのポート 5000 とホストのポート 5000 をマップする。-v でホストの /var/opt ディレクトリをコンテナの /var/lib/registry ディレクトリにマウントしてコンテナに加えた変更をホストの /var/opt に保存する。そうすることで，コンテナを停止すると基本的には加えた変更は消去されるのだが，ホスト上にその内容を保存することで消去から復元することができる。

先に作ったイメージ friendlyhello にローカル registry の場所である localhost:5000 を指定し，さらに repository の名前をつける。

`$sudo␣docker␣tag␣friendlyhello␣localhost:5000/repository` 名 `/friendlyhello`

`$sudo␣docker␣images` で localhost:5000/repository 名 /friendlyhello—ができていることを確認して，このイメージを push する。

`$sudo␣docker␣push␣localhost:5000/repository` 名 `/friendlyhello`

そうすると，マウントしたホストのディレクトリ `/var/opt` に docker/registry/v2/repositoires/repositry 名 /friendlyhello—ディレクトリが作られイメージが保存されている。

このプライベート registry のイメージは

`$sudo␣docker␣pull␣localhost:5000/repository` 名 `/friendlyhello` で呼び出すことができる。

Get Started, Part 4:Swarms を試してみると、複数のコンテナを使ったアプリの利用やコンテナにかかる負荷の調整などについて理解が深まる。

5.1 Docker for Windows について

Docker for Windows もあり今度は圧倒的シェアをもつ Windows 10 環境で試してみたくな

る。Microsoft の 64bit windows 10 pro でハイパーバイザベースの x64 向け仮想化システムである Hyper-V が動くことが必要である。VirtualBox は使えなくなるという。手元のノートパソコンで確認すると，「Windows の機能の有効化または無効化」の中に Hyer-V があったので有効化して，再起動してみた。「Windows 管理ツール」に Hyper-V マネージャーが追加されている。VM 型の仮想化とは排他的なので再起動後確かに VirtualBox の Ubuntu16.04LTS が起動できなくなっている。Docker 社の Docker for Windows の Windows インストーラーパッケージ InstallDocker.msi をダウンロードしてインストールする。すぐに cmd で Docker コマンドが使える。docker version で docker のバージョンを確認すると，17.03.1-ce である。OS/Arch は Client は windows/amd64 であり, Server は linux/amd64 である。また Hyper-V マネージャーを起動すると，MobyLinuxVM という仮想マシンが稼働し 2048MB のメモリが割り当てられている。Docker forWindows をインストールした後 cmd で docker run hello-world を実行したら Hello from Docker! This message shows that your installation appears to be working correctly. と表示されて docker がうまく動いている。Windows ユーザーも docker が使えるようになった。

6 おわりに

スーパーコンピュータを使うほどでもないが，パソコンだと少し時間がかかるような問題を，パソコンを使って何度も試行錯誤するには，プログラムを書く時間が短く，計算時間が短い方がいい。プロクラムを書く時間を短縮するには，Python のような簡易言語を使うのも一つの解決策だと思う。計算時間を短縮するための選択肢の一つにオンプレミス（自前）で GPGPU がある。ただ，高速で安全は通信が可能な環境がある場合は，オンプレミス（自前）なのかクラウドなのかという選択肢がある。今回はオンプレミスで GPGPU する環境を構築することを考えてみた。Linux の一つである Ubuntu を OS として選んで NVIDIA の GPU と CUDA,Python という組み合わせで考えた。Linux の Docker を使うとプログラムの Deployment にかかる時間を節約できることがわかった。

ただ，GeForce GTX 1060 (3G) では，すぐに GPU メモリを使い切ってしまって計算エラーを起こすので，予算に余裕があれば，メモリの大きい GPU を 2 枚以上使えるように，マザーボードや CPU が複数の GPU に対応するように考えないといけない .。CPU と GPU を効率的に冷却することも必要だろう。そうするとシステム電力にも配慮が必要であるし，交流を直流に変換するときの発熱によるロスが少ない電源ユニットを選ぶ必要が出てくる。NVIDIA のページを見ると，GeForce GTX 1060 (3G) の場合消費電力は 120W で，最小限必要な（システム）電力は 400W である。GeForce GTX 1080 Ti (11G) の場合は消費電力 250W で，最小限必要な（システム）電力は 600W ということだ。GPU を 2 枚利用する場合についても考えてみたい。ただ本稿を書くために使った PC の電源は 850W に増強しているが，

GPU を２枚使うのは慎重にしないといけないだろう。

2016 年の Windows 10 kernel with the Anniversary Update では，WSL (Windows Subsystem for Linux) として Ubuntu を動かすことができるようになった。Hyper-V を有効にして Linux ベースの Docker も利用できるようになった。さらに，もう一つ Windows containers (Linux ではなくて Windows ネイティブのコンテナ) が使えるようにもなった。2017 年の Windows 10 kernel with the Creators Update では，さらに WSL の Ubuntu のバージョンが 16.04 になり，Linux の主要な機能が動くようになっている。また Hyper-V を更新している。Windows が Linux の成果を着実に取り入れ，開発環境として使いやすくなってきている。ただ，本稿を執筆している時点では，Hyper-V は Windows 10 Pro, Enterprise,Educaton に対応していて，Windows 10 Home には対応していないので，Pro にアップグレードする必要がある。Windows 10 の最新の機能を使ってみたい学生は，そのような最新機能が早く反映される Windows10 Pro がインストールされた PC を買っておくべきだろう。また本稿を書くにあたっては，まとまった書籍がなくネットの検索に頼るしかなく，無償で提供された情報に感謝します。とくにメーカーやソフトウェアの開発元のホームページを丹念に読むことを学生にもお勧めします。