

「GPU プログラミングと並列処理の基礎」学習会

実験教育支援センター 電気系共通実験室 土屋明仁

近年の高性能計算機の急速な発達に伴い、高度で大規模なデータ処理が必要とされる研究分野の進展がめざましい。今後の授業支援、研究支援においても HPC (High Performance Computing) の話題に触れたり技術が必要となる場面が増えると考えられ、その基礎的な知識を身につけることは業務においても役立つと期待される。

今回は、スーパーコンピュータ構成のトレンドである「GPGPU (General Purpose Graphics Processing Unit) クラスタ」の手法に倣い小規模なコンピュータクラスタを構築し、そのクラスタ上でのプログラミング演習を通して、GPU プログラミングと並列処理について理解を深めることを目的として学習会を行った。

勉強会実施（日時、場所、内容）：

- ・ 2015 年 5 月 13 日 23 棟 121 GPGPU (CUDA)、HPC とは 今後の予定について
 - GPGPU とは何なのか。NVIDIA 社の GPGPU である CUDA の紹介
 - 高速数値計算の手法であるマルチスレッドプログラミングについて紹介
 - 今後の学習会のすすめかたについて意見交換
- ・ 2015 年 6 月 17 日 23 棟 121 CUDA 基礎 マルチスレッドプログラミング解説
 - CUDA 開発ボードのセットアップ（各自で作業。不明点の確認のみ）
 - CUDA の仕組み
 - マルチスレッドプログラミングの考え方
- ・ 2015 年 7 月 15 日 23 棟 121 CUDA 基礎 マルチスレッドプログラミング解説
 - CUDA 開発ボードのセットアップ（各自で作業。不明点の確認のみ）
 - 「正方行列の掛け算プログラム基礎編（自作）」について
 - アルゴリズムや動作で分からなかった点について情報交換
- ・ 2015 年 8 月 19 日 23 棟 121 CUDA 基礎 行列計算のサンプルプログラム解説
 - 「正方行列の掛け算プログラム基礎編（自作）」について
 - アルゴリズムや動作で分からなかった点について情報交換
- ・ 2015 年 9 月 9 日 23 棟 121 CUDA 基礎 行列計算のサンプルプログラム解説
 - 「正方行列の掛け算プログラム応用編（自作）」について
 - アルゴリズムや動作で分からなかった点について情報交換
- ・ 2015 年 10 月 21 日 23 棟 121 CUDA 基礎 行列計算サンプルプログラム解説
 - 「正方行列の掛け算プログラム応用編（自作）」について
 - アルゴリズムや動作で分からなかった点について情報交換

2016年9月1日

- ・2015年12月9日 23棟121 GPGPUの応用例について ディスカッション
 - － GPGPUを用いた科学計算、画像処理など、応用例についてフリーディスカッション
- ・2016年1月13日 23棟121 GPGPUの応用例について ディスカッション
 - － GPGPUを用いた科学計算、画像処理など、応用例についてフリーディスカッション
- ・2016年2月24日 23棟121 GPGPUの応用例について ディスカッション
 - － GPGPUを用いた科学計算、画像処理など、応用例についてフリーディスカッション

CUDA 開発ボード、参考書と自作の正方行列掛け算プログラムを用いて、CUDA の仕組みや CUDA プログラミングの基礎について理解を深めることができた。CUDA プログラミングの難易度が高く、当初予定していた GPGPU クラスタ構築と並列計算プログラミングは実施することが出来なかったが、企画内容を自主的に継続し、今後、参加者に情報をフィードしていきたいと考えている。

参加者の感想

池田：とても理解したというレベルには達しませんでした。GPGPU の一端を垣間見ることができました。多数の GPU リソースを使って計算処理を並列に高速に行うことができるという言葉で言うのは簡単ですが、実際に学習するとその難しさを実感しました。引き続き勉強して業務に生かせるようにしたいと思います。

茂木：学習会を通じて、GPGPU によって「何ができるのか」ということを理解しました。実務で使用するには、技術的に少々難易度が高い部分もありますが、今後の業務課題解決におけるひとつの選択肢としての知識は身に付けることができ、とても有益な学習会であったと思います。

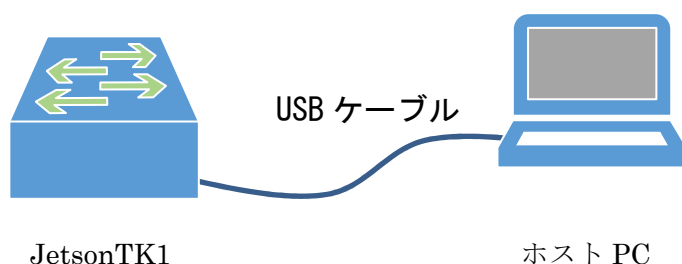
謝辞

本個人研修は、慶應義塾大学理工学部技術研修委員会の補助により行うことができました。ここに厚く御礼申し上げます。

NVIDIA JetsonTK1 セットアップ

JetsonTK1 は箱から出して電源入れるだけ使えるのですが、勉強会で必要な開発環境を構築したり OS を最新のものにするために、JetPack というソフトウェアキットをはじめにインストールしましょう。以下、セットアップ作業の流れです。

予備知識 普段は単体で動く JetsonTK1 ですが、OS の再インストールなどのメンテの際には、リカバリーモードとしてホスト PC と USB 接続して使います。



PC と USB ケーブルで接続し、[RESET] ボタンと [FORCE RECOVERY] ボタンを同時に押すとリカバリーモードで起動します。JetsonTK1 はホスト PC から USB デバイスとして認識されるようになります。

まず、ホストとなる Ubuntu12.04LST64（以下、ホスト PC、ホスト Ubuntu）をつくる Linux 専用 PC を作るのは大変なので、仮想化ソフトを使って、すでに使用中の Windows の中に Linux をインストールします。Linux は Windows 上の仮想化ソフト（仮想マシン）の中で動作することになります。

1. 仮想環境をつくるためのソフト、Oracle VirtualBox をインストールします。
2. VirtualBox から Ubuntu12.04LST (AMD64) をインストールします。
★Ubuntu 用の HDD 容量は 32GB 以上あったほうが良いと思います。
JetPack インストール途中で OpenCV や Cuda サンプルプログラムのクロスコンパイルなどがあり、意外と空き HDD 容量が必要でした。

※64bit Windows7 が動いている PC なら OK

※64BitOS をインストールするために BIOS 設定変更が必要な場合あり ↓

VirtualBox に 64bit 版 Ubuntu をインストールしようとするときに
64bit 関連のエラーが出てインストールプログラムが中断してしまうことがある
かもしれませんが、次の情報が参考になるかもしれません。

[仮想化支援機能 (VT-x/AMD-V) を有効化できません]

<http://d.hatena.ne.jp/yohei-a/20110124/1295887695>

VirtualBox の Ubuntu を立ち上げ&ログイン

Ubuntu の画面が“小さいまま”になってしまう場合はこちらご参考に。

<http://mogi2fruits.net/blog/os-software/windows/2389/>

画面が小さいままでも「Guest Additions」は実行した方がよいです。

JetPack TK1 Development Pack (JetPack) 1.0 をダウンロード

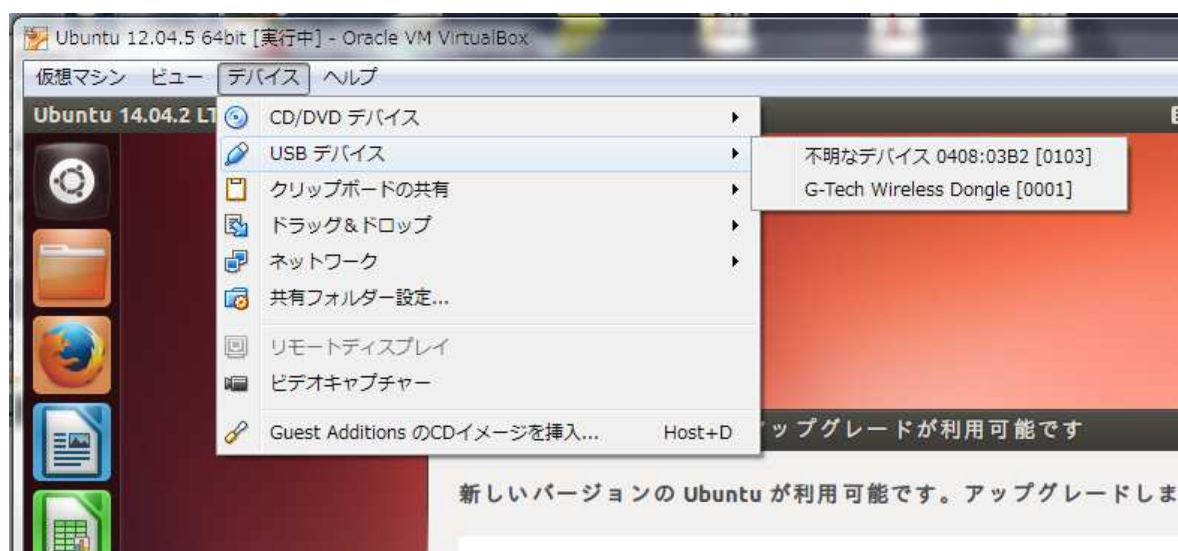
<https://developer.nvidia.com/jetson-tk1-development-pack>

L4T (Linux for Tegra) 21.2、cuda6.5 開発環境や OpenCV2.4 ライブラリなどが含まれています。

JetsonTK1 をホスト PC に接続し、リカバリーモードに

まずは JetsonTK1 にマウス、キーボード、モニタ (HDMI)、LAN ケーブルを差します。最後に AC アダプタを差します。冷却ファンが回り L4T の起動が始まった場合は、画面にログインプロンプトが出るまでしばらく放置しましょう。自動的に起動しない場合はそのまま大丈夫です。USB ケーブルでホスト PC に接続し、[RESET] ボタンと [FORCE RECOVERY] ボタンを同時に押します (同時に押したら両方すぐ離す)。リカバリーモードで再起動し、JetsonTK1 がホスト PC の Windows から USB デバイスとして認識されるようになります。

次に、VirtualBox の「デバイス」「USB デバイス」に“NVIDIA・・・”という項目が表示されるのでそれをクリックし、Ubuntu が JetsonTK1 を認識できるようにします。



ホスト Ubuntu のコマンドプロンプトで

```
$ lsusb
```

```
Bus 002 Device 002: ID 0955:7140 NVidia Corp.
```

NVIDIA のエントリが表示されていれば OK

JetPack のパーミッションを変更し実行できるように

```
$ chmod 755 JetPackTK1-1.0-cuda6.5-linux-x64.run
```

そしてダブルクリック

インストール . . .

いろいろ聞かれてきた場合は肯定的に答えてプロセスを進める。

途中で OS のフラッシュ (OS イメージの JetsonTK1 への展開) が実行され JetsonTK1 が自動的に再起動したりします。

「Install Guide」 <https://developer.nvidia.com/jetson-tk1-development-pack>

インストールプログラムが終了したら

USB ケーブルを外し、JetsonTK1 の「RESET」ボタンもしくは「POWER」ボタンを押して再起動させます。

おわり。

以上で JetsonTK1 内蔵の eMMC メモリ上にシステムがインストールされる。

SD カードに OS をインストールする場合は、以下の作業が必要

内蔵 eMMC メモリなどのシリコンメディアはそのうちデータが書き込みできなくなってしまう（書き込んでも電荷を保持できずにデータがすぐに消えてしまう）。SD カードなど交換可能なメディアにシステムを構築することをおすすめします。

作業の流れ：

1. JetsonTK1 が eMMC ブートしたら
2. SD カードを差し
3. fdisk でパーティション作成し
作成例)

```
$ sudo fdisk /dev/mmcblk1p1
```

```
/dev/mmcblk1p1 25GB ext4 /      ルートファイルシステム用
```

```
/dev/mmcblk1p2  4GB swap swap   スワップ領域
```
4. /dev/mmcblk1p1 を Linux ファイルシステムでフォーマット

```
$ sudo mkfs.ext4 /dev/mmcblk1p1
```

```
$ sudo sync; sudo sync; sudo sync
```
5. ホスト PC と JetsonTK1 を USB ケーブルで接続し、
JetsonTK1 をリカバリモードに切り替える
6. SD カードをホスト PC に差し替え、
ホスト Ubuntu が SD カードを認識できるようにする。そしてマウント。

```
$ sudo mount /dev/sdb1 /mnt/
```
7. ホスト Ubuntu 内に展開された JetPackTK1 から
L4T のシステムファイルを SD カードにコピーします。

```
$ cd ~/JetPackTK1-1.0/Linux_for_Tegra/rootfs
```

```
$ sudo cp -a * /mnt
```

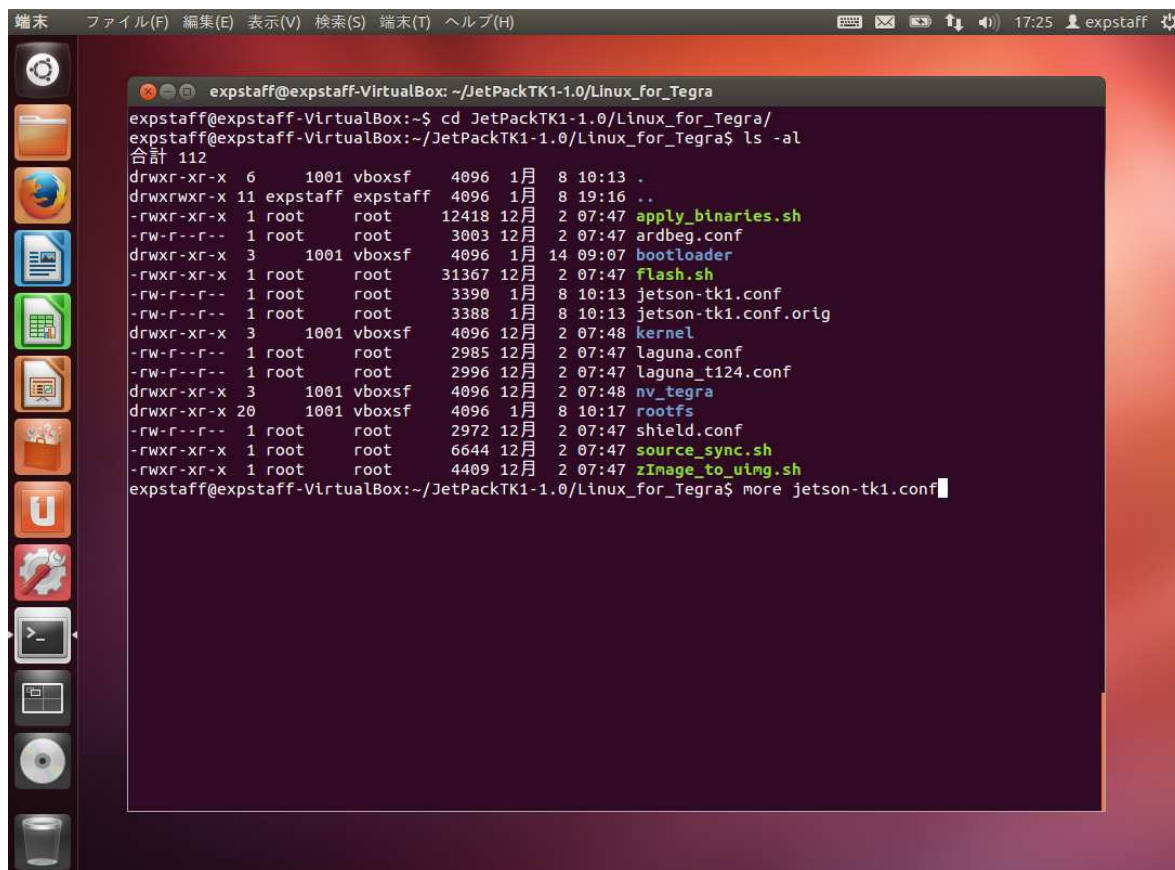
```
$ sudo sync; sudo umount /dev/sda1
```

次ページのホスト Ubuntu の画面参照
8. ホスト Ubuntu 上で L4T ブートローダの設定を行う (flash コマンドを実行)

```
$ cd ~/JetPackTK1-1.0/Linux_for_Tegra/
```

```
$ ./flash.sh jetson-tk1 mmcblk1p1    (と、その前に設定が必要(次ページ))
```

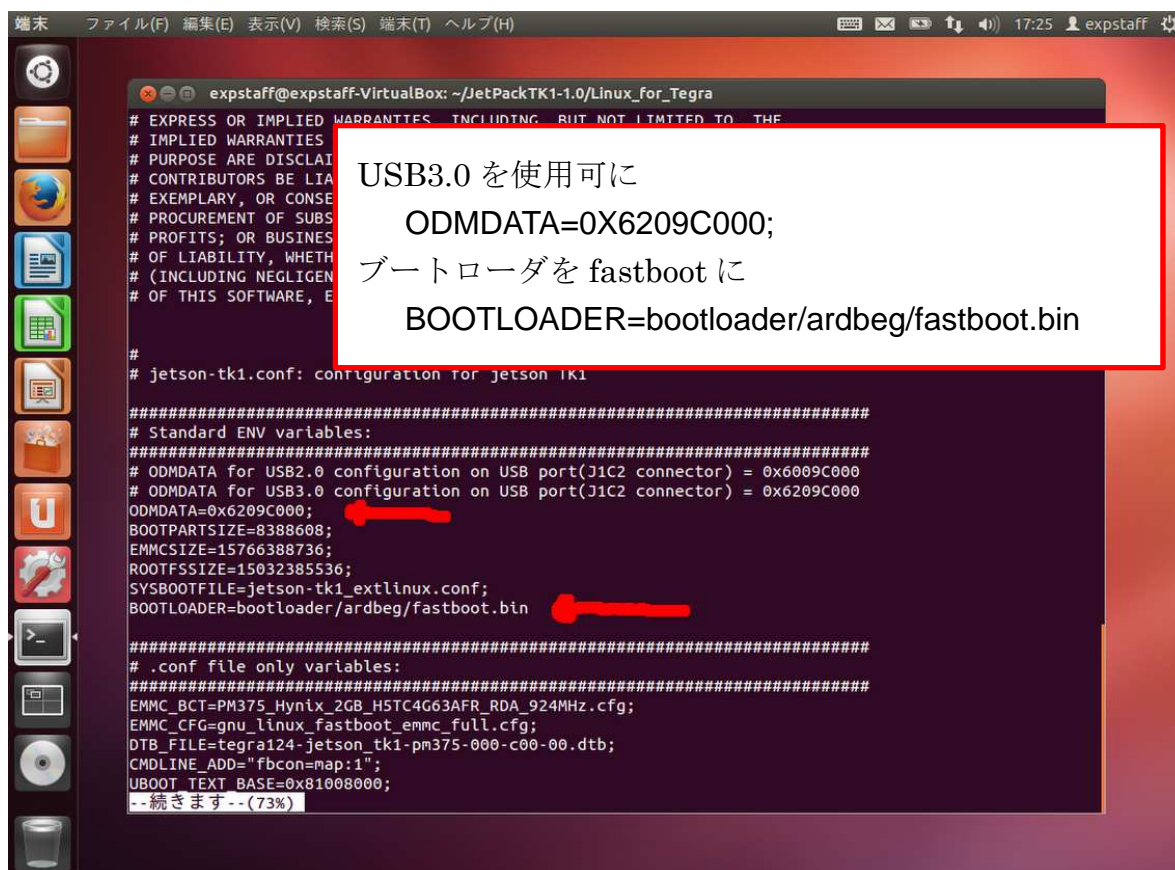
これで JetsonTK1 はブートデバイスとして SD カードを見るようになります



```
expstaff@expstaff-VirtualBox: ~/JetPackTK1-1.0/Linux_for_Tegra
expstaff@expstaff-VirtualBox:~$ cd JetPackTK1-1.0/Linux_for_Tegra/
expstaff@expstaff-VirtualBox:~/JetPackTK1-1.0/Linux_for_Tegra$ ls -al
合計 112
drwxr-xr-x 6 1001 vboxsf 4096 1月 8 10:13 .
drwxrwxr-x 11 expstaff expstaff 4096 1月 8 19:16 ..
-rwxr-xr-x 1 root root 12418 12月 2 07:47 apply_binaries.sh
-rw-r--r-- 1 root root 3003 12月 2 07:47 ardbeg.conf
drwxr-xr-x 3 1001 vboxsf 4096 1月 14 09:07 bootloader
-rwxr-xr-x 1 root root 31367 12月 2 07:47 flash.sh
-rw-r--r-- 1 root root 3390 1月 8 10:13 jetson-tk1.conf
-rw-r--r-- 1 root root 3388 1月 8 10:13 jetson-tk1.conf.orig
drwxr-xr-x 3 1001 vboxsf 4096 12月 2 07:48 kernel
-rw-r--r-- 1 root root 2985 12月 2 07:47 laguna.conf
-rw-r--r-- 1 root root 2996 12月 2 07:47 laguna_t124.conf
drwxr-xr-x 3 1001 vboxsf 4096 12月 2 07:48 nv_tegra
drwxr-xr-x 20 1001 vboxsf 4096 1月 8 10:17 rootfs
-rw-r--r-- 1 root root 2972 12月 2 07:47 shield.conf
-rwxr-xr-x 1 root root 6644 12月 2 07:47 source_sync.sh
-rwxr-xr-x 1 root root 4409 12月 2 07:47 zImage_to_uring.sh
expstaff@expstaff-VirtualBox:~/JetPackTK1-1.0/Linux_for_Tegra$ more jetson-tk1.conf
```

★flash.sh を実行する前に jetson-tk1.conf を編集します（下図参照）

- ・デフォルトでは USB3.0 が無効となっているので有効となるよう設定変更します
- ・L4T のブートローダはデフォルトで u-boot がインストールされますが、SSD や HDD にインストールする場合は fastboot がインストールされるよう設定変更します。u-boot が SATA ブートをサポートしていないため。



```
# EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
# IMPLIED WARRANTIES
# PURPOSE ARE DISCLA
# CONTRIBUTORS BE LIA
# EXEMPLARY, OR CONSE
# PROCUREMENT OF SUBS
# PROFITS; OR BUSINES
# OF LIABILITY, WHETH
# (INCLUDING NEGLIGEN
# OF THIS SOFTWARE, E

#
# jetson-tk1.conf: configuration for jetson tk1
#####
# Standard ENV variables:
#####
# ODMDATA for USB2.0 configuration on USB port(J1C2 connector) = 0x6009C000
# ODMDATA for USB3.0 configuration on USB port(J1C2 connector) = 0x6209C000
ODMDATA=0x6209C000;
BOOTPARTSIZE=8388608;
EMMC_SIZE=15766388736;
ROOTFS_SIZE=15032385536;
SYSBOOTFILE=jetson-tk1_extlinux.conf;
BOOTLOADER=bootloader/ardbeg/fastboot.bin

#####
# .conf file only variables:
#####
EMMC_BCT=PM375_Hynix_2GB_H5TC4G63AFR_RDA_924MHz.cfg;
EMMC_CFG=gnu_linux_fastboot_emmc_full.cfg;
DTB_FILE=tegra124-jetson_tk1-pm375-000-c00-00.dtb;
CMDLINE_ADD="fbcon=map:1";
UBOOT_TEXT_BASE=0x81008000;
--続きます--(73%)
```

ちなみに、SATA-HDD や SATA-SSD をブートデバイスにするときは

```
$ sudo ./flash.sh jetson-tk1 sda1
```

ブートデバイスを eMMC に戻すときは

```
$ sudo ./flash.sh jetson-tk1 mmcblk0p1
```

フラッシュが終わると JetsonTK1 が再起動します。

9. SD カードを JetsonTK1 に差し替える

ログイン後パーティションの容量を確認し増えていればSDカードが使えています。

```
$ df -k
```

10. JetPack インストーラを起動 ふたたびインストール

CUDA や OpenCV などとはまたインストールしなおさないといけない。

ホスト Ubuntu から JetsonTK1 に ID:ubuntu でパスワードなし SSH ログインできるように設定しないといけない。

ホスト Ubuntu にて ssh-keygen で公開鍵を作成し、
JetsonTK1 の ID:Ubuntu の ~/.ssh 下にコピーしておく。

SSH を使ってパスワードなしで接続 このあたりなどを参考に。

<http://ubuntu.u-aizu.ac.jp/004/index.html>

11. OS 以外のパッケージを選択、OS フラッシュを外し、インストールを進める

今回は OS インストールは必要ないので L4T と OS イメージ展開のチェックを
はずし、CUDA 開発環境、OpenCV ライブラリが展開されるように項目を
選択する。

おわり。お疲れさまでした。

テキストで使用している `cuda` が古いので、サンプルプログラムをコンパイルするときにエラーが出ます。そのときの対処法です。

① タイマー関数、エラーチェック関数を使うときに必要なヘッダーファイル

```
#include <cutil.h>
```

とある部分、

`cutil.h` が見つからないというコンパイルエラーが出ます。

`helper_timer.h` `helper_cuda.h` をインクルードするよう修正してください。

```
//#include <cutil.h>
#include <helper_cuda.h>
#include <helper_timer.h>
```

また、コンパイルオプションに

`-I/home/ubuntu//NVIDIA_CUDA-6.5_Samples /common/inc` を追加してください。

② タイマー関数名の修正

```
unsigned int  timer;
cutCreateTimer(&timer);
cutResetTimer(timer);
cutStartTimer(timer);
cutStopTimer(timer);
float  elapsed_time = cutGetTimerValue(timer)*1.0e-03;
```

となっているのを

以下のように修正してください。 `cutResetTimer()` の分はなくなる。

```
StopWatchInterface *timer = NULL;
sdkCreateTimer(&timer);
sdkStartTimer(&timer);cutStartTimer(timer);
sdkStopTimer(&timer);
float  elapsed_time = sdkGetTimerValue(&timer)*1.0e-03;
```

③ エラーチェック関数名の修正

```
CUDA_SAFE_CALL( cudaMemcpy( ... ) );
```

のようになっているのを

以下のように修正してください。

```
checkCudaErrors( cudaMemcpy( ... ) );
```

CUDA 参考 URL

独立行政法人 理化学研究所 情報基盤センター HPC>講習会テキスト

「CUDA プログラミング入門」

<http://accc.riken.jp/hpc/training/>

東京大学情報基盤センタースーパーコンピューティング部門

HOME>利用者支援>広報・刊行物>スーパーコンピューティングニュース

<http://www.cc.u-tokyo.ac.jp/support/press/news/>

- ・ これからの並列計算のための GPGPU 連載講座 (I)
GPU と GPGPU の歴史と特徴(PDF)
- ・ これからの並列計算のための GPGPU 連載講座 (II)
GPGPU プログラミング環境 CUDA 入門編(PDF)
- ・ これからの並列計算のための GPGPU 連載講座 (III)
GPGPU プログラミング環境 CUDA 最適化編(PDF)

NVIDIA DEVELOPER ZONE

「CUDA C Best Practices Guide」

<http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>

SFC ITC

平成 26 年 12 月 18 日

SFC-CNS/ERNS 利用者各位

湘南藤沢 ITC

新共用計算サーバの試験運用の開始について (12/18)

SFC-CNS では、従来異なるアーキテクチャの計算資源を提供するという方針から SPARC アーキテクチャと Intel アーキテクチャそれぞれの 共用計算サーバを提供してまいりました。近年コンピューティングがマルチコア・並列処理へと変化をしていることから、GPGPU などを用いた並列処理環境 の導入の検討を実施してまいりました。

その結果、x86_64 アーキテクチャとの互換性が高い Intel Xeon Phi の導入を決定し、本日より 新共用計算サーバの提供を開始します。

新共用計算サーバは以下の特徴があります

- 最大 1,056 スレッドを利用した並列分散処理が可能（ホスト 80 スレッド、コプロセッサ 976 スレッド）
- Intel コンパイラを利用することで特定の処理のみのオフロードが可能（ただし教育目的に限る）
- x86_64 アーキテクチャと互換性があるため、既存のソース資源を活用できる

詳しい利用方法などは、新共用計算サーバに順次公開いたします。

SFC における授業・研究にご活用ください。

以上

最終更新日: 2014 年 12 月 18 日

```

1 // プログラミング演習 ～基礎編～ サンプルプログラム 1
2 // HELLO WORLD 2015.5.12
3 // TSUCHIYA, Akihito
4 //
5 // 参考 :
6 // 日本GPUコンピューティングパートナーシップ
7 // 第8回 プログラミング演習 ～基礎編～
8 // http://www.gdep.jp/page/view/255
9 //
10 // 以下、
11 // CPU-> ホスト
12 // GPU-> デバイス と呼ぶ
13 //
14
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <string.h>
18 #include <time.h>
19
20 // cudaの便利関数用ヘッダー
21 #include <helper_timer.h>
22
23 // c++で標準入出力関数を使うときにインクルードするもの
24 #include <iostream>
25
26 // バッファサイズを定義する
27 #define BUFFERSIZE 128
28
29 // c++で書くときのお約束的なもの。標準関数名前空間
30 using namespace std;
31
32
33 //////////////////////////////////////
34 // これは普通のcの関数
35 // ホスト上での文字列コピー関数
36 //
37 // 引数srcの文字列を引数dstにコピーする
38 // C標準ライブラリ関数strcpy() と同等の関数
39 //
40 void host_strcpy( char *dst, char *src ) {
41     // 文字列srcの終端に達するまでループする
42     while ( *src != '\0' ) {
43         // srcが示すメモリ番地のデータをdstが示すメモリ番地にコピー

```

```

44     *dst = *src;
45     // dstが示すメモリ番地をインクリメント
46     dst++;
47     // srcが示すメモリ番地をインクリメント
48     src++;
49 }
50 }
51
52
53 ///////////////////////////////////////////////////
54 // cuda特有！
55 // __device__ 修飾子：
56 //   デバイス側で動作し、デバイス側から呼び出される関数
57 //
58 // 引数srcの文字列を引数dstにコピーする
59 // デバイスではC標準ライブラリ関数が使えないため
60 // 自分で関数を定義しなければならない
61 //
62 __device__
63 void device_strcpy( char *dst, char *src ) {
64     while ( *src != '\0' ) {
65         *dst = *src;
66         dst++;
67         src++;
68     }
69 }
70
71
72 ///////////////////////////////////////////////////
73 // cuda特有！
74 // __global__ 修飾子：
75 //   デバイス側で動作し、ホスト側から呼び出される関数（カーネル）
76 //   戻り値は必ずvoidになる
77 //
78 // デバイスメモリ上に文字列データを格納する
79 // 引数のstrは、確保されたデバイスメモリの先頭アドレスを示すポインタ
80 //
81 __global__
82 void helloGpu( char *str ) {
83     // 文字列を strにコピーする
84     // strはデバイスメモリ（グローバルメモリ）に格納される
85     device_strcpy( str, "Hello GPU!" );
86 }

```

```

87 |
88 |
89 | //////////////////////////////////////////////////
90 | // cuda特有！
91 | // kernelFunc<<<grid-dim, block-dim, SharedMemory-size, Stream>>>(引数1, 引数2, ...):
92 | //   カーネル実行環境の設定
93 | //
94 | // デバイスメモリ上に文字列データを格納する
95 | // 引数のstrは、確保されたデバイスメモリの先頭アドレスを示すポインタ
96 | //
97 | int main( int argc, char **argv ) {
98 |
99 |     // ホストメモリ用に使う文字型変数
100 |     char *text_Hst;
101 |     // デバイスメモリ用に使う文字型変数
102 |     char *text_Dev;
103 |
104 |     // ホストメモリを sizeof(char)*BUFFERSIZEビット(=8*128ビット=128バイト)確保する
105 |     text_Hst = ( char * )malloc( sizeof( char ) * BUFFERSIZE );
106 |     // text_Hstに文字列を格納
107 |     host_strcpy( text_Hst, "Hello world!" );
108 |
109 |     // text_Hstを表示
110 |     cout << "BEFORE" << endl;
111 |     cout << "text_Hst: " << text_Hst << endl << endl;
112 |
113 |     // cuda特有！
114 |     // デバイスメモリを sizeof(char)*BUFFERSIZEビット(=8*128ビット=128バイト)確保する
115 |     // デバイスメモリを確保するときは cudaMalloc() を使う
116 |     cudaMalloc( ( void ** )&text_Dev, sizeof( char ) * BUFFERSIZE );
117 |
118 |     // cuda特有！
119 |     // カーネル関数呼び出し
120 |     // グリッド次元1、ブロック次元1。つまり1つの単発スレッドで
121 |     // カーネル関数を実行する。という意味
122 |     dim3 gridDim( 1, 1, 1 );
123 |     dim3 blockDim( 1, 1, 1 );
124 |     // 文字列コピーだけなので、複数スレッドで並列処理する必要が無いため
125 |     // 単発スレッドによる実行としました
126 |     helloGpu<<<gridDim, blockDim>>>( text_Dev );
127 |
128 |     // スレッド実行の同期をとる。
129 |     // デバイスでの処理が終わるのを待つ、というような意味。

```

```
130 // ホストとデバイスの動作は非同期であることに注意。
131 // ホストはカーネル関数を呼び出したあと、デバイスでの処理が終わったかどうかを
132 // 確認しないで次の行の実行に移るため。
133 cudaDeviceSynchronize();
134
135 // text_Hstを表示
136 cout << "check" << endl;
137 cout << "text_Hst: " << text_Hst << endl << endl;
138
139 // デバイスメモリのデータをホストメモリにコピーする
140 cudaMemcpy( text_Hst, text_Dev, sizeof( char ) * BUFFERSIZE, cudaMemcpyDeviceToHost );
141
142 // text_Hstを表示
143 // 文字列が変わっているはず！
144 cout << "AFTER" << endl;
145 cout << "text_Hst: " << text_Hst << endl << endl;
146
147 return 0;
148 }
149
```

```

1 // プログラミング演習 ～基礎編～ サンプルプログラム2
2 // HELLO WORLD (マルチスレッド版) 2015. 5. 13
3 // TSUCHIYA, Akihito
4 //
5 // 参考 :
6 //
7 // 以下、
8 // CPU-> ホスト
9 // GPU-> デバイス と呼ぶ
10 //
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <string.h>
15 #include <time.h>
16
17 // cudaの便利関数用ヘッダー
18 #include <helper_timer.h>
19
20 // c++で標準入出力関数を使うときにインクルードするもの
21 #include <iostream>
22
23 // バッファサイズを定義する
24 #define BUFFERSIZE 128
25
26 // c++で書くときのお約束的なもの。標準関数名前空間
27 using namespace std;
28
29
30 //////////////////////////////////////
31 // cuda特有 !
32 // __global__ 修飾子:
33 //   デバイス側で動作し、ホスト側から呼び出される関数 (カーネル)
34 //   戻り値は必ずvoidになる
35 //
36 // 引数のstrは、確保されたデバイスメモリの先頭アドレスを示すポインタ
37 //
38 // 32個のスレッドそれぞれが "HELLO GPU! ..."のうちの1文字を受け持つ
39 // ★詳細はパワポスライド2～5参照
40 __global__
41 void helloGpu( char *str ) {
42     char *hello = "HELLO GPU! This is a simple code";
43

```


1 main2. cu

```
44     int i = gridDim.x * blockDim.x * blockDim.y * blockIdx.y +
45             gridDim.x * blockDim.x * threadIdx.y +
46             blockDim.x * blockIdx.x + threadIdx.x;
47
48     str[ i ] = hello[ i ];
49 }
50
51
52 //////////////////////////////////////
53 //
54 int main( int argc, char **argv ) {
55
56     // ホストメモリ用に使う文字型変数
57     char *text_Hst;
58     // デバイスメモリ用に使う文字型変数
59     char *text_Dev;
60
61     // ホストメモリを sizeof(char)*BUFFERSIZEビット(=8*128ビット=128バイト)確保する
62     text_Hst = ( char * )malloc( sizeof( char ) * BUFFERSIZE );
63
64     // cuda特有 !
65     // デバイスメモリを sizeof(char)*BUFFERSIZEビット(=8*128ビット=128バイト)確保する
66     // デバイスメモリを確保するときは cudaMalloc() を使う
67     cudaMalloc( ( void ** )&text_Dev, sizeof( char ) * BUFFERSIZE );
68
69     // 変数text_Hst に 文字列"Hello CPU!"を格納
70     strcpy( text_Hst, "Hello CPU!" );
71     // text_Hstを表示
72     cout << "BEFORE" << endl;
73     cout << "text_Hst: " << text_Hst << endl << endl;
74
75     // cuda特有 !
76     // カーネル関数呼び出し
77     // グリッド次元x: 2, y: 1, z: 1
78     // ブロック次元x: 4, y: 4, z: 1
79     // つまり2*4*4=32個のスレッドを作って実行する
80     //
81     // kernelFunc<<<grid-dim, block-dim, SharedMemory-size, Stream>>>(引数1, 引数2, ...):
82     // カーネル実行環境の設定
83     // ★詳細はパワポスライド1 参照
84     //
85     // スレッドの数を32にした理由 :
86     //   39行目で宣言している文字列が32文字だから。
```

```
87 //
88 dim3 gridDim( 2, 1 );
89 dim3 blockDim( 4, 4 );
90 helloGpu<<<gridDim, blockDim>>>( text_Dev );
91
92 // cuda特有 !
93 // 32個のスレッド動作の同期をとる。
94 //
95 // ホストはカーネル関数を呼び出したあと、デバイスでの処理終了を待たずに直ちに次の行の実行に移る。
96 // ホストとデバイスの動作は非同期であることに注意。
97 //
98 cudaDeviceSynchronize( );
99
100 // 念のためこの時点での変数text_Hstの中身を確認
101 cout << "check" << endl;
102 cout << "text_Hst: " << text_Hst << endl << endl;
103 // まだ変化なし
104
105 // デバイスのデータをホストにコピーする
106 cudaMemcpy( text_Hst, text_Dev, sizeof( char ) * BUFFERSIZE, cudaMemcpyDeviceToHost );
107
108 // text_Hstを表示
109 cout << "AFTER" << endl;
110 cout << "text_Hst: " << text_Hst << endl << endl;
111 // 文字列が変わっている !
112
113 return 0;
114 }
115
```

main2.cu: 86-88行目

```
dim3 gridDim( 2, 1 );
```

```
dim3 blockDim( 4, 4 );
```

```
helloGpu<<<gridDim,blockDim>>>( text_Dev );
```

によって、下図のようなスレッド `(threadIdx.x, threadIdx.y)` が生成される。

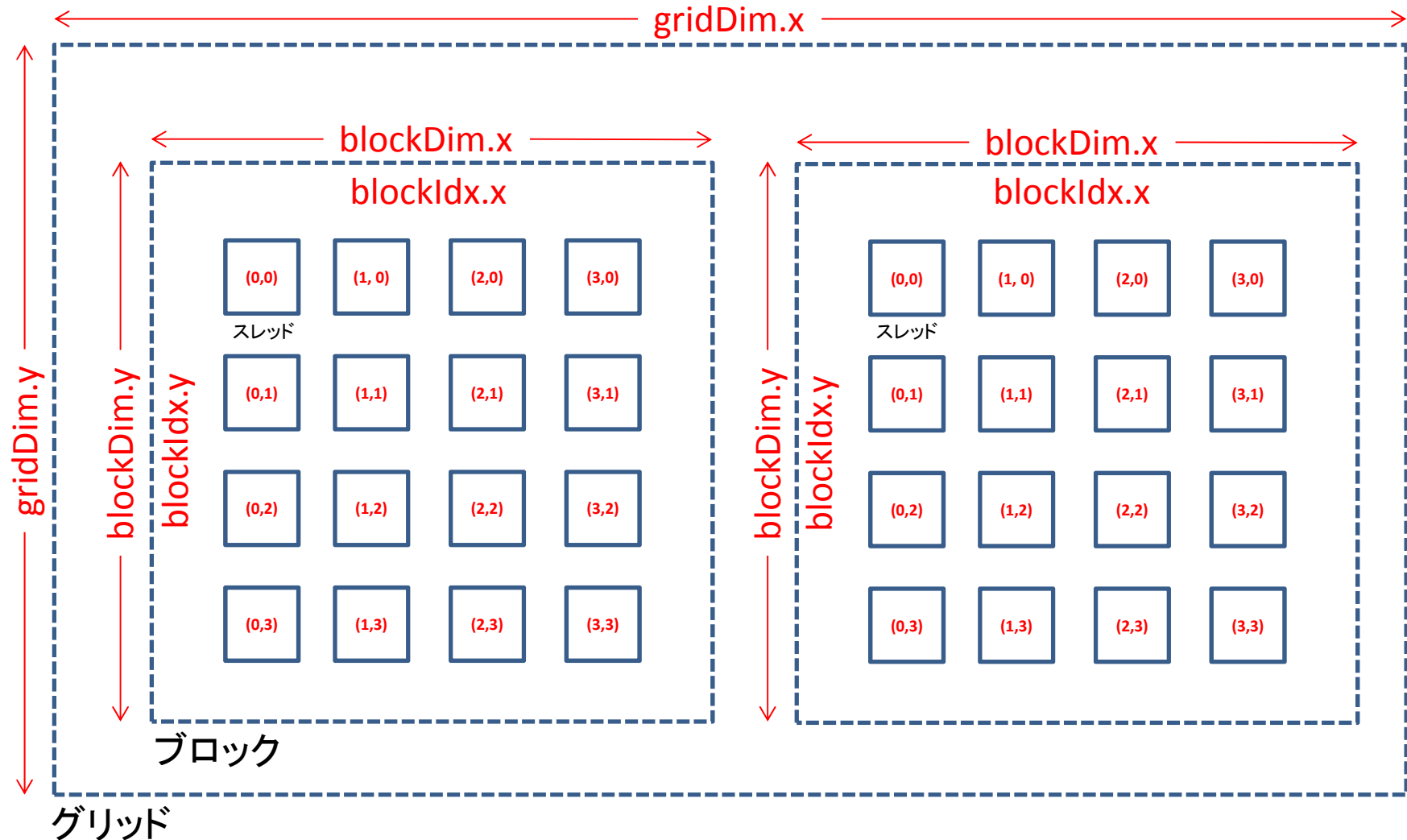
赤字の変数がプログラム中で参照できる。

`gridDim.x = 2`

`gridDim.y = 1`

`blockDim.x = 4`

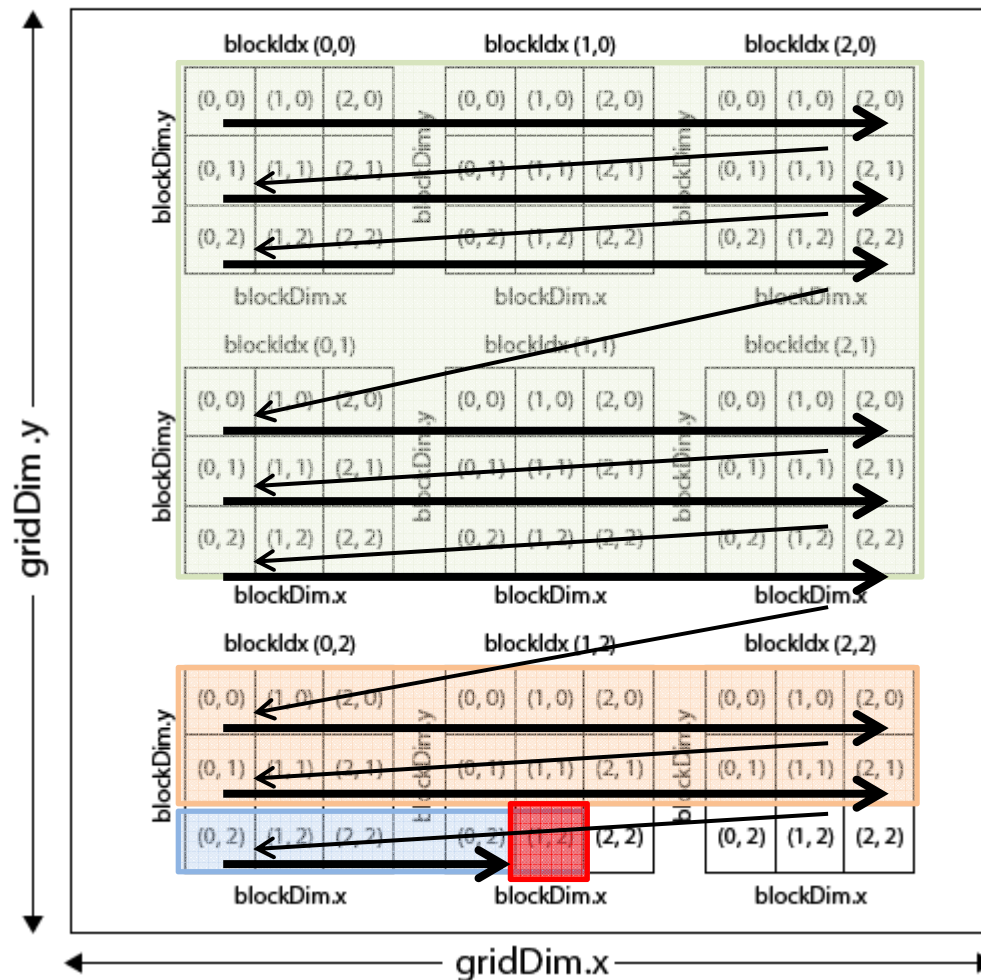
`blockDim.y = 4`



main2.cu: 43-45行目の意味の説明

```
int i = gridDim.x * blockDim.x * blockDim.y * blockIdx.y +  
       gridDim.x * blockDim.x * threadIdx.y +  
       blockDim.x * blockIdx.x + threadIdx.x ;
```

CUDA Grid



とあるスレッドが、自分は

左上から何番目なのかを知りたいとすると

$$\begin{aligned} i &= 3 * 3 * 3 * 2 + 3 * 3 * 2 + 3 * 1 + 1 \\ &= 54 + 18 + 4 \\ &= 76 \text{番目 と分かる} \end{aligned}$$

注: ゼロから数える

```
dim3 gridDim( 2, 1 );
dim3 blockDim( 4, 4 );
helloGpu<<<gridDim,blockDim>>>( text_Dev );
```

それぞれのスレッドの
“中身”のイメージ

スレッド(0,0)

```
char *hello =
"HELLO GPU!
  This is a simple code";
Int i = 0;
str[ i ] = hello[ i ];
```

```
blockIdx.x = 0
blockIdx.y = 0
threadIdx.x = 0
threadIdx.y = 0
```

スレッド(1,0)

```
char *hello =
"HELLO GPU!
  This is a simple code";
Int i = 1;
str[ i ] = hello[ i ];
```

```
blockIdx.x = 0
blockIdx.y = 0
threadIdx.x = 1
threadIdx.y = 0
```

スレッド(1,0)

```
char *hello =
"HELLO GPU!
  This is a simple code";
Int i = 2;
str[ i ] = hello[ i ];
```

```
blockIdx.x = 0
blockIdx.y = 0
threadIdx.x = 2
threadIdx.y = 0
```

スレッド(0,0)

```
char *hello =
"HELLO GPU!
  This is a simple code";
Int i = 8;
str[ i ] = hello[ i ];
```

```
blockIdx.x = 0
blockIdx.y = 0
threadIdx.x = 0
threadIdx.y = 1
```

スレッド(1,0)

```
char *hello =
"HELLO GPU!
  This is a simple code";
Int i = 9;
str[ i ] = hello[ i ];
```

```
blockIdx.x = 0
blockIdx.y = 0
threadIdx.x = 1
threadIdx.y = 1
```

スレッド(1,0)

```
char *hello =
"HELLO GPU!
  This is a simple code";
Int i = 10;
str[ i ] = hello[ i ];
```

```
blockIdx.x = 0
blockIdx.y = 0
threadIdx.x = 2
threadIdx.y = 1
```

スレッド(0,0)

```
char *hello =
"HELLO GPU!
  This is a simple code";
```

```
blockIdx.x = 0
blockIdx.y = 0
threadIdx.x = 0
threadIdx.y = 2
```

スレッド(1,0)

```
char *hello =
"HELLO GPU!
  This is a simple code";
```

```
blockIdx.x = 0
blockIdx.y = 0
threadIdx.x = 1
threadIdx.y = 2
```

スレッド(1,0)

```
char *hello =
"HELLO GPU!
  This is a simple code";
```

```
blockIdx.x = 0
blockIdx.y = 0
threadIdx.x = 2
threadIdx.y = 2
```

```
char *hello = "HELLO GPU! This is a simple code";
int i = gridDim.x * blockDim.x * blockDim.y * blockIdx.y +
    gridDim.x * blockDim.x * threadIdx.y +
    blockDim.x * blockIdx.x + threadIdx.x;
str[ i ] = hello[ i ];
```

そうすると・・・
それぞれのスレッドが受け持つ
文字のイメージ

グリッド

ブロック

スレッド

(0,0) H i=0	(1,0) E i=1	(2,0) L i=2	(3,0) L i=3
(0,1) U i=8	(1,1) ! i=9	(2,1) i=10	(3,1) T i=11
(0,2) i i=16	(1,2) s i=17	(2,2) i=18	(3,2) a i=19
(0,3) p i=24	(1,3) l i=25	(2,3) e i=26	(3,3) i=27

(0,0) O i=4	(1,0) i=5	(2,0) G i=6	(3,0) P i=7
(0,1) h i=12	(1,1) i i=13	(2,1) s i=14	(3,1) i=15
(0,2) i=20	(1,2) s i=21	(2,2) i i=22	(3,2) m i=23
(0,3) c i=28	(1,3) o i=29	(2,3) d i=30	(3,3) e i=31

GPU グローバルメモリ

str [0] [1] [2] [3] [4] [5] [6] [7]

H	E	L	L	O		G	P
---	---	---	---	---	--	---	---

....

[28][29][30][31]

p	l	e		c	o	d	e
---	---	---	--	---	---	---	---

グリッド

代入していく

ブロック

スレッド

(0,0) H i=0	(1,0) E i=1	(2,0) L i=2	(3,0) L i=3
(0,1) U i=8	(1,1) ! i=9	(2,1)	(3,1) T i=11
(0,2) i i=16	(1,2) s i=17	(2,2)	(3,2) a i=19
(0,3) p i=24	(1,3) l i=25	(2,3) e i=26	(3,3)

(0,0) O i=4	(1,0)	(2,0) G i=6	(3,0) P i=7
(0,1) h i=12	(1,1) i i=13	(2,1) s i=14	(3,1)
(0,2)	(1,2) s i=21	(2,2) i i=22	(3,2) m i=23
(0,3) c i=28	(1,3) o i=29	(2,3) d i=30	(3,3) e i=31

行列を2次元配列で表現する

A =

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

例) 配列に値をセットする

```
int N = 3;
```

```
int A[ N ][ N ];
```

```
for ( int i = 0; i < N; i++ )
```

```
    for ( int j = 0; j < N; j++ )
```

```
        A[ i ][ j ] = aij;
```

結果:

A[0][0]=a00 A[0][1]=a01 A[0][2]=a02

A[1][0]=a10 A[1][1]=a11 A[1][2]=a12

A[2][0]=a20 A[2][1]=a21 A[2][2]=a22

行列を1次元配列で表現する

A =

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

例) 配列に値をセットする

```
int N = 3;
```

```
int A[ N * N ];
```

```
for ( int i = 0; i < N; i++ )
```

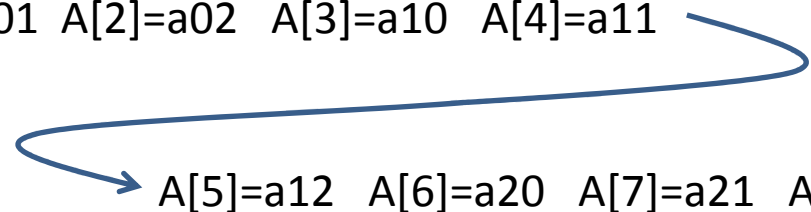
```
    for ( int j = 0; j < N; j++ )
```

```
        A[ i * N + j ] = aij;
```

結果:

A[0]=a00 A[1]=a01 A[2]=a02 A[3]=a10 A[4]=a11

A[5]=a12 A[6]=a20 A[7]=a21 A[8]=a22



行優先(row-major)と列優先(column-major)

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

行優先(row-major)での答え

$$\begin{pmatrix} 15 & 18 & 21 \\ 42 & 54 & 66 \\ 69 & 90 & 111 \end{pmatrix}$$

列優先(Column-major)での答え

$$\begin{pmatrix} 15 & 18 & 21 \\ 42 & 54 & 66 \\ 69 & 90 & 111 \end{pmatrix}$$

```

1 // プログラミング演習 ～基礎編～ 行列の掛け算サンプルプログラム
2 //
3 // mulmat.cu
4 // 正方行列の掛け算 C=A*B
5 // 注意！！
6 // column-majorで書かれています
7 // ふだん慣れ親しんでいるのはrow-major
8 // row-major B*A = column-major A*B
9 //
10 // コンパイルの仕方
11 // nvcc -o mulmat mulmat.cu -I/home/ubuntu/NVIDIA_CUDA-6.5_Samples/common/inc
12 //
13 // test program 2015.01.12
14 // test program 2015.03.04
15 //
16 #include <stdio.h>
17 #include <stdlib.h>
18 // #include <time.h>
19 #include <helper_timer.h>
20
21 #define N 320 // 行列のサイズ (計算量は320*320=102400)
22 #define BLOCKSIZE 32 // ブロックのサイズ (32*32=1024スレッドで1ブロック)
23 #define TILE_DIM 32 // シェアードメモリを使うときの部分行列のサイズ (Warpが32なので32にしてみました)
24
25 //////////////////////////////////////
26 // 行列表示
27 // Matlab/Octave format
28 void printmat( int W, int M, float *A, int LDA ) {
29     printf( "%n " );
30     for ( int i = 0; i < W; i++ ) {
31         printf( " " );
32         for ( int j = 0; j < M; j++ ) {
33             printf( "%.2f", A[i+j*LDA] );
34             if ( j < M - 1 )
35                 printf( ", " );
36         }
37         if ( i < W - 1 )
38             printf( "%n " );
39         else
40             printf( " " );
41     }
42     printf( "%n" );
43 }

```

```

44 |
45 |
46 | //////////////////////////////////////////////////
47 | // CPU 普通の掛け算
48 | void cpuMultiply( float *A, float *B, float *C ) {
49 |     for ( int col = 0; col < N; col++ )
50 |         for ( int row = 0; row < N; row++ ) {
51 |             C[ row * N + col ] = 0;
52 |             for ( int i = 0; i < N; i++ )
53 |                 C[ row * N + col ] += A[ row * N + i ] * B[ i * N + col ];
54 |         }
55 | }
56 |
57 |
58 | //////////////////////////////////////////////////
59 | // GPU グローバルメモリ & マルチスレッドによる行列掛け算
60 | __global__ // use global memory
61 | void globalMultiply( float *A, float *B, float *C ) {
62 |
63 |     int col = blockDim.x * blockIdx.x + threadIdx.x;
64 |     int row = blockDim.y * blockIdx.y + threadIdx.y;
65 |     float tmp = 0.0f;
66 |
67 |     for ( int i = 0; i < N; i++ )
68 |         tmp += A[ row * N + i ] * B[ i * N + col ];
69 |
70 |     // blockDim.x*blockDim.x equals N.
71 |     C[ row * N + col ] = tmp;
72 | }
73 |
74 |
75 | //////////////////////////////////////////////////
76 | // GPU シェアードメモリ & マルチスレッドの方法
77 | __global__ // shared memory
78 | void sharedMultiply( float *A, float* B, float *C ) {
79 |
80 |     int bx = blockIdx.x;
81 |     int by = blockIdx.y;
82 |     int tx = threadIdx.x;
83 |     int ty = threadIdx.y;
84 |     int row = blockIdx.y * blockDim.y + threadIdx.y;
85 |     int col = blockIdx.x * blockDim.x + threadIdx.x;
86 |     float tmp = 0.0f;

```

```

87
88 __shared__ float aTile[ TILE_DIM ][ TILE_DIM ];
89 __shared__ float bTile[ TILE_DIM ][ TILE_DIM ];
90
91 for ( int i = 0; i < N; i += TILE_DIM ) {
92
93     aTile[ ty ][ tx ] = A[ row * N + i + tx ];
94     bTile[ ty ][ tx ] = B[ ( i + ty ) * N + col ];
95     __syncthreads();
96
97     for ( int j = 0; j < TILE_DIM; j++ ) {
98         tmp += aTile[ ty ][ j ] * bTile[ j ][ tx ];
99     }
100     __syncthreads();
101 }
102
103 C[ row * N + col ] = tmp;
104 }
105
106
107 ///////////////////////////////////////////////////
108 int main( int argc, char **argv ) {
109
110     cudaError_t error;
111
112     // ホスト(CPU)用
113     float *A_h, *B_h, *C_h;
114     // デバイス(GPU)用
115     float *A_d, *B_d, *C_d;
116
117     // 乱数初期化
118     // 行列 A_h, B_h の初期値用
119     srand( ( unsigned int )time( NULL ) );
120
121     // ホストメモリを float(32bit) x 行列サイズ の分だけ確保
122     A_h = ( float * )malloc( sizeof( float ) * N * N );
123     B_h = ( float * )malloc( sizeof( float ) * N * N );
124     C_h = ( float * )malloc( sizeof( float ) * N * N );
125
126     // A_h, B_h を初期化
127     for ( int i = 0; i < N * N; i++ ) {
128         A_h[ i ] = ( float )( rand() % 10 );
129         B_h[ i ] = ( float )( rand() % 10 );

```

```

130 }
131
132 // デバイスメモリを float(32bit) x 行列サイズ分だけ確保
133 cudaMalloc( ( void ** )&A_d, sizeof( float )*N*N );
134 cudaMalloc( ( void ** )&B_d, sizeof( float )*N*N );
135 cudaMalloc( ( void ** )&C_d, sizeof( float )*N*N );
136
137 // 開始イベント（時間計測用）を作成
138 cudaEvent_t start;
139 error = cudaEventCreate(&start);
140 if (error != cudaSuccess) {
141     fprintf(stderr, "Failed to create start event (error code %s)!\n", cudaGetErrorString(error));
142     exit(EXIT_FAILURE);
143 }
144 // 終了イベント（時間計測用）を作成
145 cudaEvent_t stop;
146 error = cudaEventCreate(&stop);
147 if (error != cudaSuccess) {
148     fprintf(stderr, "Failed to create stop event (error code %s)!\n", cudaGetErrorString(error));
149     exit(EXIT_FAILURE);
150 }
151
152 // デバイスメモリに A_h, B_h を転送
153 cudaMemcpy( A_d, A_h, sizeof( float )*N*N, cudaMemcpyHostToDevice );
154 cudaMemcpy( B_d, B_h, sizeof( float )*N*N, cudaMemcpyHostToDevice );
155
156 dim3 block( BLOCKSIZE, BLOCKSIZE );
157 dim3 grid( N/BLOCKSIZE, N/BLOCKSIZE );
158
159 // 暖機運転 //////////////////////////////////////
160 // warmup
161 // global memory
162 //globalMultiply<<<grid,block>>>( A_d, B_d, C_d );
163 // shared memory
164 //coalescedMultiply<<<grid,block>>>( A_d, B_d, C_d );
165 // shared memory
166 sharedMultiply<<<grid,block>>>( A_d, B_d, C_d );
167 //cudaThreadSynchronize();
168 cudaDeviceSynchronize();
169
170 // Allocate CUDA events that we'll use for timing
171 error = cudaEventCreate(&start);
172 if (error != cudaSuccess) {

```

```

173     fprintf(stderr, "Failed to create start event (error code %s)!\n", cudaGetErrorString(error));
174     exit(EXIT_FAILURE);
175 }
176 error = cudaEventCreate(&stop);
177 if (error != cudaSuccess) {
178     fprintf(stderr, "Failed to create stop event (error code %s)!\n", cudaGetErrorString(error));
179     exit(EXIT_FAILURE);
180 }
181 //////////////////////////////////////////////////
182
183 // 本番運転 //////////////////////////////////////
184 // 時間計測開始
185 // Record the start event
186 error = cudaEventRecord(start, NULL);
187 if (error != cudaSuccess) {
188     fprintf(stderr, "Failed to record start event (error code %s)!\n", cudaGetErrorString(error));
189     exit(EXIT_FAILURE);
190 }
191
192 // GO
193 // global memory
194 //globalMultiply<<<grid,block>>>( A_d, B_d, C_d );
195 // shared memory
196 //coalescedMultiply<<<grid,block>>>( A_d, B_d, C_d );
197 // shared memory
198 sharedMultiply<<<grid,block>>>( A_d, B_d, C_d );
199 //cudaThreadSynchronize();
200 //cudaDeviceSynchronize();
201
202 // 時間計測終了
203 error = cudaEventRecord(stop, NULL);
204 if (error != cudaSuccess) {
205     fprintf(stderr, "Failed to record stop event (error code %s)!\n", cudaGetErrorString(error));
206     exit(EXIT_FAILURE);
207 }
208 // Wait for the stop event to complete
209 error = cudaEventSynchronize(stop);
210 if (error != cudaSuccess) {
211     fprintf(stderr, "Failed to synchronize on the stop event (error code %s)!\n", cudaGetErrorString(error));
212     exit(EXIT_FAILURE);
213 }
214 //////////////////////////////////////////////////
215

```

```

216 // デバイスメモリからホストメモリに計算結果を転送
217 cudaMemcpy( C_h, C_d, sizeof( float ) * N * N, cudaMemcpyDeviceToHost );
218
219 // 時間の計測結果を表示
220 float msecTotal = 0.0f;
221 error = cudaEventElapsedTime(&msecTotal, start, stop);
222 if (error != cudaSuccess) {
223     fprintf(stderr, "Failed to get time elapsed between events (error code %s)!%n", cudaGetErrorString(error));
224     exit(EXIT_FAILURE);
225 }
226 // 計算時間結果の表示 (ミリ秒であることに注意
227 printf("It takes %f msec on GPU%cn", msecTotal );
228 //printf("%fGflops%cn", (2*N*N*N/(msecTotal*0.001*1e+9) ));
229 printf( "%f Gflops%cn%cn", 2*N*N*N/(msecTotal*0.001*1000000000) );
230 //printf( "please calc 2*%d*%d*%d/(%.2f*0.001*1000000000) Gflops%cn%cn", N, N, N, msecTotal );
231 //printmat( N, N, A_h, N );
232 //printf( "%cn" );
233 //printmat( N, N, B_h, N );
234 //printf( "%cn" );
235 //printmat( N, N, C_h, N );
236
237 //////////////////////////////////////////
238 // いちおうCPUバージョンも
239 // ただし、行列の次元が大きくなると
240 // 計算に時間がかかるので注意！！
241
242 // タイマーを作成して計測開始
243 StopwatchInterface *timer = NULL;
244 sdkCreateTimer( &timer );
245 sdkStartTimer( &timer );
246
247 // use cpu
248 cpuMultiply( A_h, B_h, C_h );
249 //printf( "%cn" );
250 //printmat( N, N, C_h, N );
251
252 // 時間計測終了
253 sdkStopTimer( &timer );
254 // 計算時間結果の表示 (ミリ秒であることに注意
255 printf("It takes %f msec on CPU%cn", sdkGetTimerValue( &timer ) );
256 printf("%f Gflops%cn", (2*N*N*N/(sdkGetTimerValue( &timer )*0.001*1e+9) ));
257 //printf( "please calc %f Gflops%cn%cn", 2*N*N*N/(sdkGetTimerValue( &timer )*0.001*1000000000) );
258 //printf( "please calc 2*%d*%d*%d/(%.2f*0.001*1000000000) Gflops%cn%cn", N, N, N, msecTotal );

```



```
259 |
260 |     sdkDeleteTimer( &timer );
261 |     //////////////////////////////////////////
262 |
263 |     // デバイスメモリを解放
264 |     cudaFree( A_d );
265 |     cudaFree( B_d );
266 |     cudaFree( C_d );
267 |
268 |     // ホストメモリを解放
269 |     free( A_h );
270 |     free( B_h );
271 |     free( C_h );
272 |
273 |     cudaThreadExit();
274 |     return 1;
275 | }
276 |
```