GPUコンピューティング No.3

GPUによる並列計算と CUDA C プログラミング概要

東京工業大学　学術国際情報センター

青木　尊之

並列処理

TSUBAMEに搭載されるGPU(M2050)の内部には400以上（最新は2500以上）のプロセッサ（CUDAコア）がある。これをうまく使うことによりGPU本来の性能を引き出すことができる。

並列処理（計算）を制する（マスターする）者は、GPUコンピューティングを制す。

並列計算は、

タスク並列　と　データ並列

に分けられる。

例：宿題の分担

仲の良い4人組（青木君、斉藤君、佐藤君、鈴木君）は、夏休みの宿題を楽に済ませようと考え、分担して片付けることにした。

・数学	大問1つの中に小問が3題
・英語	大問1つの中に小問が4題
・物理	大問1つの中に小問が5題
・化学	大問1つの中に小問が4題

青木君が数学、斉藤君が英語、佐藤君が物理、鈴木君が化学を担当する。

例：算数の練習問題

14 + 34 =	10 + 23 =	29 - 31 =
8 + 50 =	12 + 55 =	- 2 - 20 =
22 + 92 =	42 - 23 =	51 - 44 =
33 - 23 =	13 - 7 =	- 39 + 28 =
- 5 + 40 =	31 + 12 =	- 27 + 10 =
- 1 + 70 =	15 + 77 =	- 9 - 62 =
17 + 3 =	3 + 12 =	30 - 38 =
100 - 40 =	- 7 + 80 =	78 + 22 =
6 + 25 =	- 23 - 2 =	- 54 + 70 =

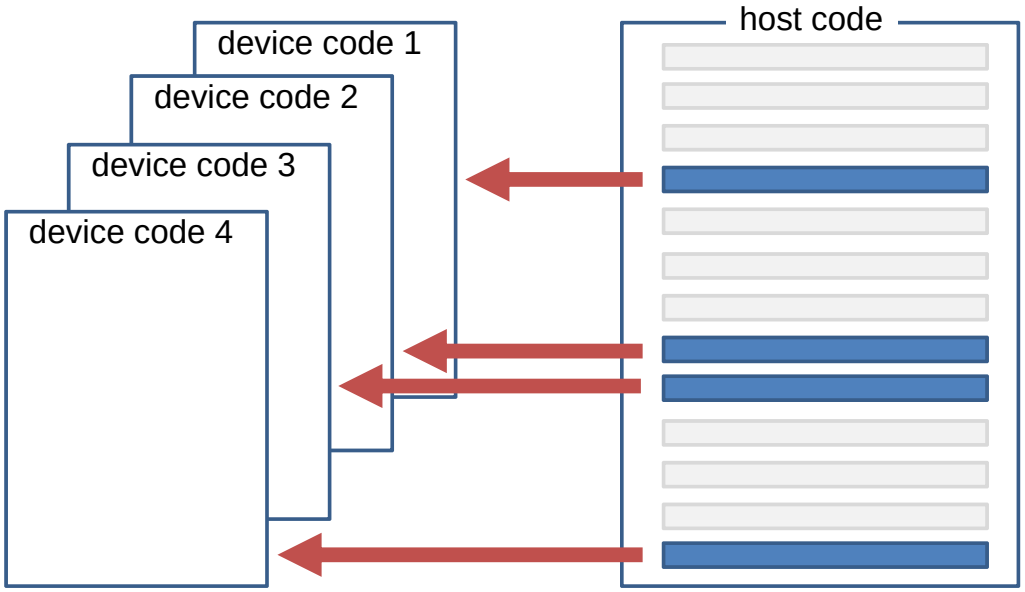
```
int  add(int a, int b) { return a + b; }
```


CUDA（最新は ver.5.0）

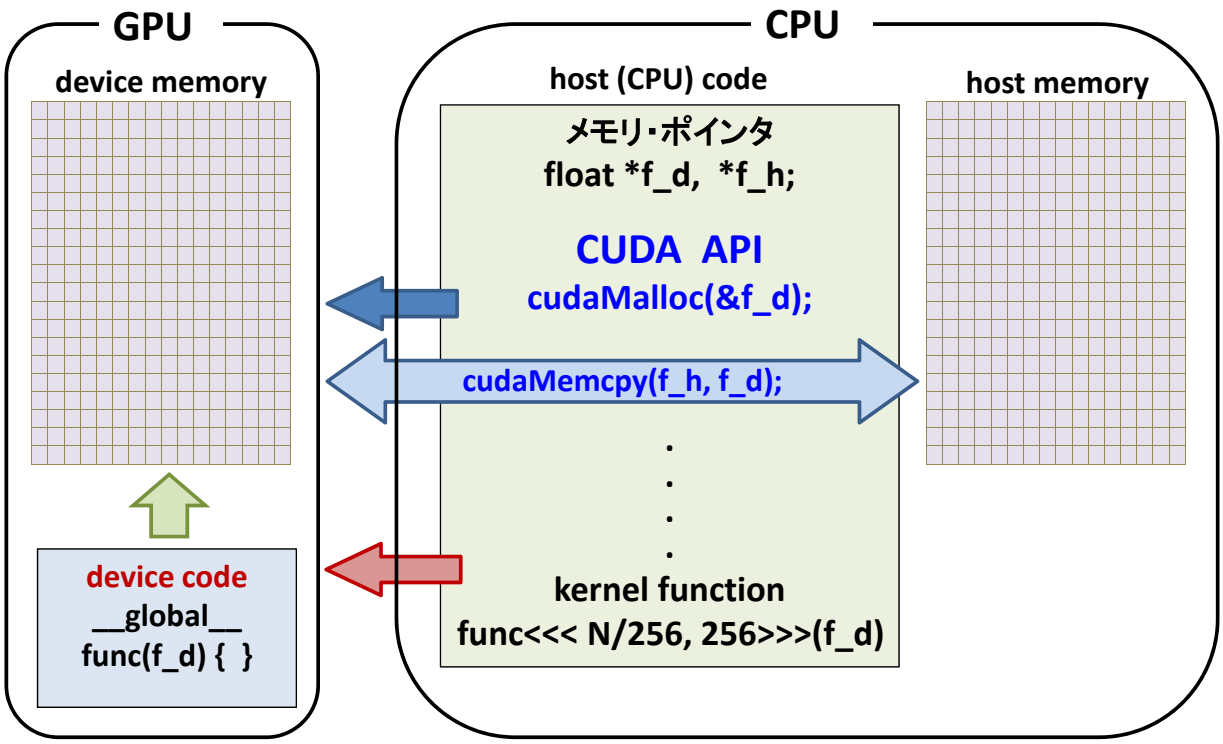
- NVIDIA GPU でGPUコンピューティングを可能にするコンパイラ・ライブラリ等を含むフレームワーク
- host (CPU) code と GPU kernel 関数を記述する device code から構成される
- host code には、CUDA API（それに付随した変数）と GPU kernel 関数 call が含まれる。
- device code（GPU kernel 関数）は、thread の実行内容を記述。thread ID 等のビルトイン変数、準備された特別な関数等は含まれる以外は通常のC言語が使える。return 値は不可。

host code と device code

GPUは単独では動かない。host を CPU で実行させ、その中からの CUDA API と GPU kernel 関数を call



CUDAのプログラム実行の概念図



CUDA ソースコードのコンパイル

- CUDA のソースファイルは拡張子 .cu を付ける。
- CUDA Toolkit の nvcc でコンパイルする。

  nvcc はCPUで実行するコードと、GPUで実行する GPU kernel 関数のコード、CUDA のAPI の部分を分離。

  CPUで実行するコードは gcc, g++ にコンパイルを任せる。GPUで実行する GPU kernel 関数の部分を GPU 用にコンパイルする。GPU 用の PTX コードも生成する。

- Library をリンクして、実行ファイルを生成する。

  CUDA core library (cuda) -lcuda
  CUDA runtime library (cudart) -lcudart

CUDA Compiler: nvcc

■ 重要なコンパイル・オプション

-arch sm_20	Compute Capability に応じたコンパイルを行う。DeviceQuery で確認し、それ以下を指定す。
--maxrregcount <N>	1つのkernel 関数当たりに使用するレジスタ数を <N> に制限する。このことにより、指定した並列数でthreadが実行可能となるが、溢れた部分はlocal メモリ上に置かれ、実行速度は低下する。
-use_fast_math	高速な数学関数を利用する。
-G	device コードに対して、デバッグを可能にする
--ptxas-options=-v	レジスタやメモリの使用状況を表示する

CUDA Memory 確保（1/2）

メモリ・ポインターは、device (GPU) memory にも host (CPU) memory にも使える。

例）単精度実数: float *f_d, *f_h;

device 上にメモリを確保する runtime API

```
cudaMalloc(void **devptr, size_t count);
```

devptr: デバイスメモリアドレスへのポインタ。
確保したメモリのアドレスが書き込まれる

count: 領域のサイズ

例） cudaMalloc((void **)&f_d, sizeof(float)*n);

f_d[n] の配列が GPU のメモリ上に確保される

CUDA Memory 確保（2/2）

float *f_h;

host 側にpinned メモリを確保する

cudaMallocHost(void **devptr, size_t count);

devptr: ホストメモリアドレスへのポインタ。
Page lock（pinned）された確保したメモリのアドレス
が書き込まれる

count: 領域のサイズ

例） **cudaMallocHost((void **)&f_h, sizeof(float)*n);**

**f_h[n] の配列が Host メモリ上にpage lock (pinned)で確保さ
れる。通常の pageable メモリとして確保された場合より、転送
速度が速い。また、非同期通信の場合も page lock メモリに限
定される。**

f_h = (float *) malloc(sizeof(float)*n);（通常）

CUDA データ転送

float *f_d, *f_h;

**cudaMemcpy(void *dst, const void *src, size_t count,
enum cudaMemcpyKind kind)**

dst: 転送先メモリ･アドレス
src: 転送元メモリ･アドレス
count: 領域のサイズ
kind: 転送タイプを指定する定数
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice

例） **cudaMemcpy (f_d, f_h, sizeof(float)*n,
cudaMemcpyHostToDevice);**

host上のf_h[n] の配列のデータをdevice上のf_d[n] にコピーする。

CUDA 非同期データ転送

float *f_d, *f_h;

cudaMemcpyAsync(void *dst, const void *src, size_t count,
enum cudaMemcpyKind kind,
cudaStream_t stream)

dst: 転送先メモリ･アドレス
src: 転送元メモリ･アドレス
count: 領域のサイズ
kind: 転送タイプを指定する定数
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost

例） **cudaMemcpyAsync (f_d, f_h, sizeof(float)*n,**
cudaMemcpyHostToDevice, stream);

Host上のf_h[n] の配列のデータをDevice上のf_d[n] に非同期でコピーする。

GPU kernel-function call

host code の中で次のように call する。

kernel_function<<< Dg, Db, Ns, S>>>(a, b, c, ...);

Dg: dim3 タイプの grid のサイズ指定
Db: dim3 タイプの block のサイズ指定
Ns: 実行時に指定する shared メモリのサイズ
省略可： 省略した場合は、0 が設定
S: 非同期実行の stream 番号
省略可： 省略した場合は、0 が設定され、
GPUのthread間は同期実行となる

Dg, Db で指定される数の thread が実行される。
kernel function の実行は、CPU に対して絶えず非同期。

dim3 宣言

kernel_function<<< Dg, Db, Ns, S>>>(a, b, c, ...);

の Dg, Db を dim3 で指定する。

dim3 a;	等価	dim3 a(1,1,1);
dim3 a(n, m);	等価	dim3 a(n, m, 1);

dim3 a(n, m, k);	等価	a.x = n;
		a.y = m;
		a.z = k;

dim3 a(n0, m0, k0); は宣言と共に値の代入であり、

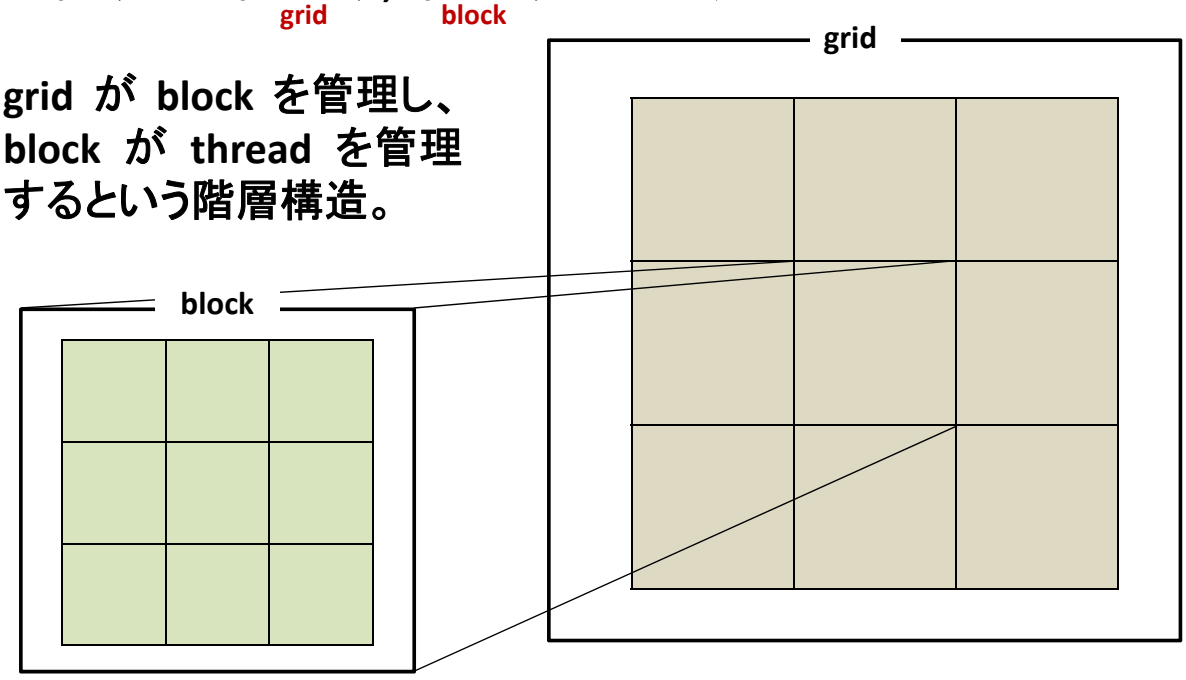
随時 a.x = n1; a.y = m1; a.z = k1; と変更可能である。

Threadの管理

kernel 関数<<< 第1引数, 第2引数>>>で指定

grid block

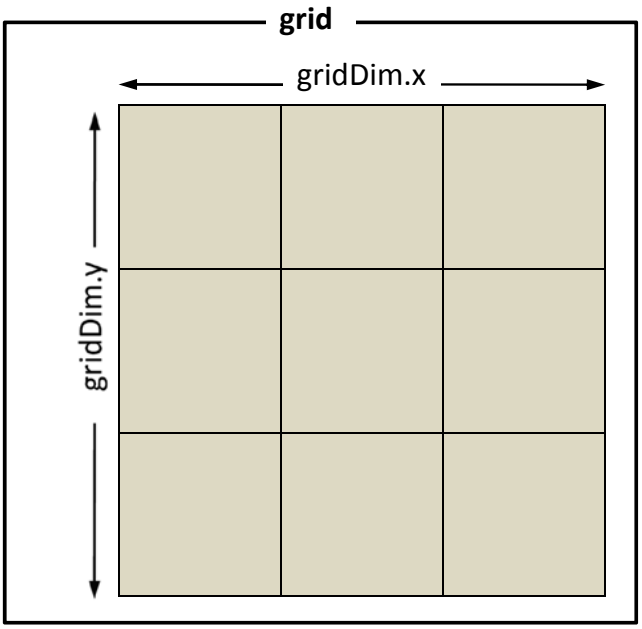
grid が block を管理し、block が thread を管理するという階層構造。



Threadの管理（grid)

kernel 関数の第1引数を以下の grid で指定すると、

```
dim3 grid(m, n, k);
  grid.x = m;
  grid.y = n;
  grid.z = k;
```



grid の中に n*m 個の blockがある。

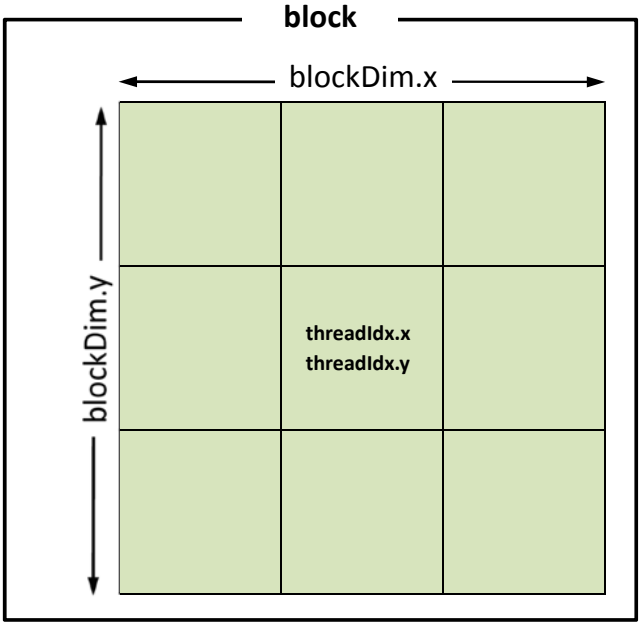
kernel 関数の中では、

m → gridDim.x
n → gridDim.y

Threadの管理（block)

kernel 関数の第2引数を以下の block で指定すると、

```
dim3 block(m, n, k);
  block.x = m;
  block.y = n;
  block.z = k;
```



block の中に n*m*k 個の threadが動く。

kernel 関数の中では、

blockIdx.x, blockIdx.y
blockIdx.z で指定できる

m → blockDim.x
n → blockDim.y
k → blockDim.z

cont.

block 内のThreadは同一 SM (Multi Processor)で実行される。

同一block 内のthreadの最大値は 1024 であり、MP内には32個のSPしかないにも関わらず、64以上のThread数で実行した方が効率が良い（性能が出る、たぶん）。

同一block内で実行されるthreadの同期を取ることができる。（__syncthreads(); ）

同一block 内のthreadは、shared メモリ（後述）を共有することができる。

block内のthread数を増やすと、使えるレジスタ数が減少する。（Fermi の場合、256　thread 並列
→ レジスタ数 32768/256）

Exercise

TSUBAME に login:

$ sh /opt/cuda/5.0/cuda.sh

```
#include <stdio.h>

int main(void) {

    printf("GPU compting¥n");

    return 0;
}
```

$ nvcc sample01.cu　　// nvcc によるコンパイル
$ ./a.out　　// 実行

最も簡単なカーネル関数


```
#include <stdio.h>
#include <cuda.h>

__global__  void   built_in(void)
{
   printf("threadIdx.x=%d¥n",threadIdx.x);
}

int main(void) {
   printf("GPU compting¥n");
   built_in<<<1,3>>>();

   return 0;
}
```

$ nvcc –arch sm_20 sample01.cu
$ ./a.out

最も簡単なカーネル関数

GPU カーネル関数の中で printf() 等を使うには、nvcc によるコンパイルの際に、オプション

–arch sm_20 （GPU cabability に応じてそれ以上）

が必要。

$ ./a.out

実行しても、期待通りに標準出力で文字列が表示されない。

何故？

