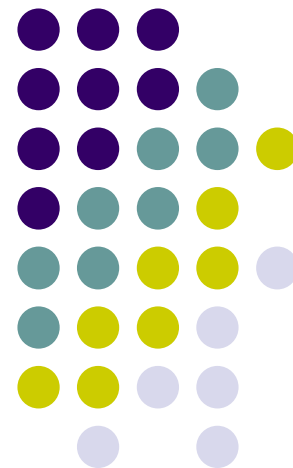


2016年度 実践的並列コンピューティング

GPUプログラミング (1)

遠藤 敏夫

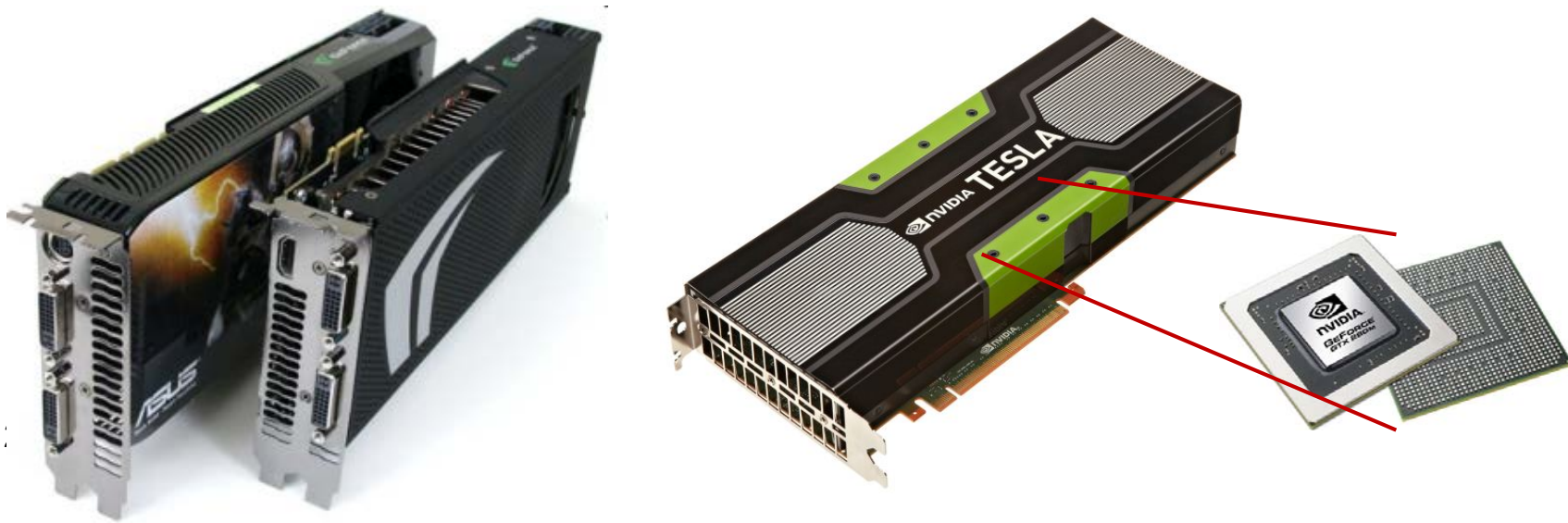
endo@is.titech.ac.jp

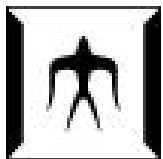


GPUコンピューティングとは



- グラフィックプロセッサ (GPU)は、グラフィック・ゲームの画像計算のために、進化を続けてきた
 - 現在、CPUのコア数は2～12個に対し、GPU中には数百コア
- そのGPUを一般アプリケーションの**高速化**に利用！
 - GPGPU (General-Purpose computing on GPU) とも言われる
- 2000年代前半から研究としては存在。2007年にNVIDIA社の**CUDA言語**がリリースされてから大きな注目





TSUBAME2スーパーコンピュータ



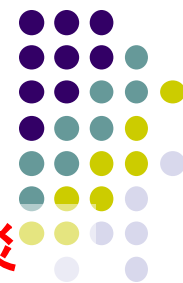
Tokyo-Tech
Supercomputer and
UBiquitously
Accessible
Mass-storage
Environment

「ツバメ」は東京工業大学の
シンボルマークでもある

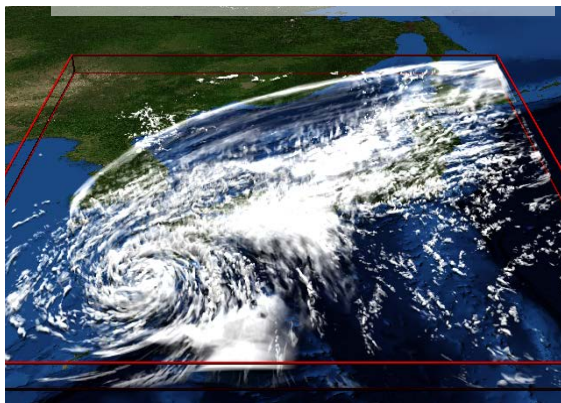
- TSUBAME1: 2006年～2010年に稼働したスパコン
- **TSUBAME2.0**: 2010年に稼働開始したスパコン
 - 2010年当初には、**世界4位、日本1位**の計算速度性能
- TSUBAME2.5: 2013年にGPUを最新へ入れ替え
 - 現在、世界24位、日本3位

高性能の秘訣が
GPUコンピューティング

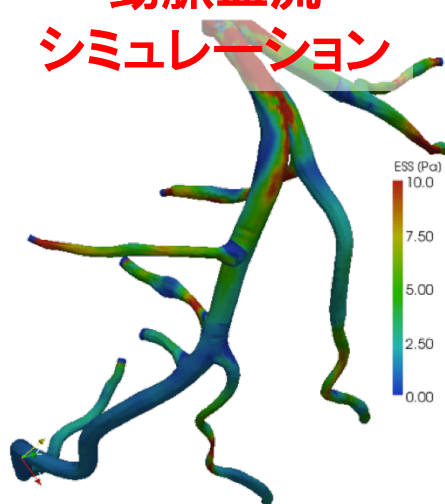
TSUBAME2スパコン・GPUは様々な研究分野で利用されている



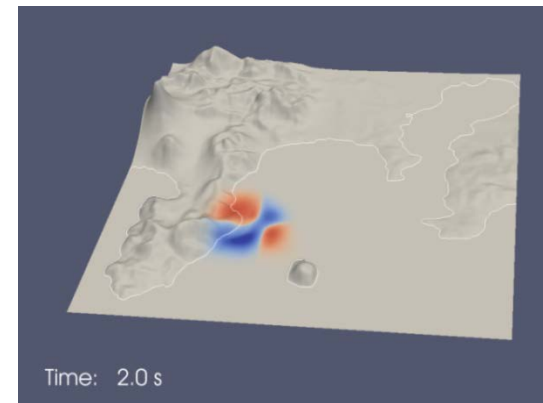
気象シミュレーション



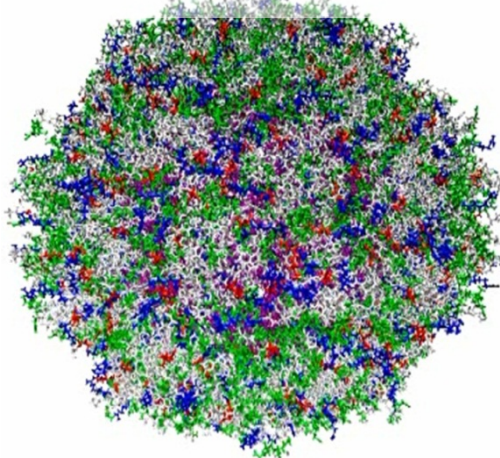
動脈血流
シミュレーション



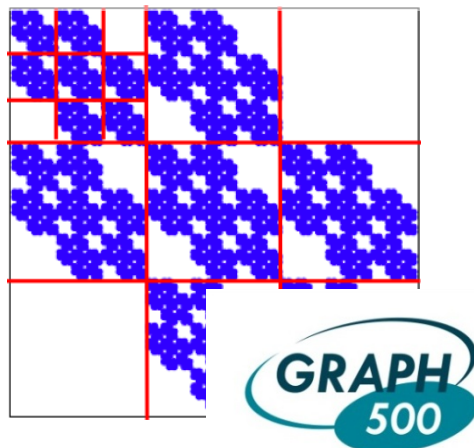
津波・防災
シミュレーション



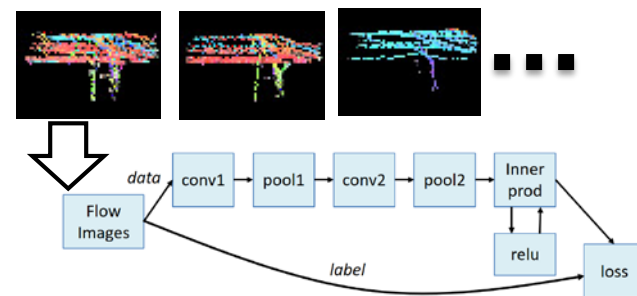
ウィルス分子
シミュレーション



ビッグデータ・グラフ解析



ディープラーニング



AlphaGoもGPUを
多数利用

TSUBAME 2.5 全体概要



TSUBAME 2.5: A GPU-centric Green 5.78 Petaflops Supercomputer

TSUBAME 2.5: "Tiny" footprint, very power efficient
- Floorspace less than 200m² (2,100 ft²)

Processor

CPU 



Intel Xeon X5670
(Westmere-EP)

- 2.93 GHz
- 76.8 GFLOPS

GPU



NVIDIA Tesla K20X
(Kepler GK110)

- 1.31 TFLOPS (DP)
- 3.95 TFLOPS (SP)
- 6 GB

Compute Node



HP ProLiant SL390s G7
(2 CPUs, 3 GPUs)

- 4.08 TFLOPS
- 58 GB (CPU)
- 18 GB (GPU)

Node Chassis



HP ProLiant S6500
(4 Compute Nodes)

- 16.3 TFLOPS
- 232 GB (CPU)
- 72 GB (GPU)

Rack



HP MCS Rack
(8 Node Chassis)

- 122 TFLOPS
- 1.74 TB (CPU)
- 540 GB (GPU)

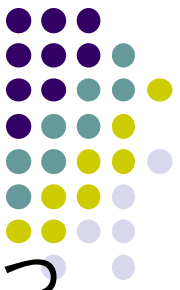
System



TSUBAME 2.5
(1,408 Compute Nodes)

5.78 PFLOPS
81.6 TB (CPU)
25.3 TB (GPU)

TSUBAME2.5の計算ノード



- TSUBAME2.0は、約1400台の計算ノード(コンピュータ)を持つ
 - 各計算ノードは、CPUとGPUの両方を持つ
 - CPU: Intel Xeon 2.93GHz 6コア x 2CPU=12 コア
 - GPU: NVIDIA Tesla K20X x 3GPU
- $0.07\text{TFlops} \times 2 \text{ (CPU)} + 1.31\text{TFlops} \times 3 \text{ (GPU)} = 4.08\text{TFlops}$

96%の性能がGPUのおかげ

- メインメモリ(CPU側メモリ): 54GB
 - デバイスメモリ(GPU側メモリ): 6GB x 3GPU
- SSD: 120GB
- ネットワーク: QDR InfiniBand x 2 = 80Gbps
- OS: SUSE Linux 11 (Linuxの一種)



GPUの特徴 (1)



- コンピュータにとりつける増設ボード
⇒ 単体では動作できず、CPUから指示を出してもらう
- 多数コアを用いて計算
⇒ 多数のコアを活用するために、多数のスレッドが協力して計算
- メモリサイズは1～12GB
⇒ CPU側のメモリと別なので、「データの移動」もプログラミングする必要

コア数・メモリサイズは、
製品によって違う



GPUの特徴 (2)



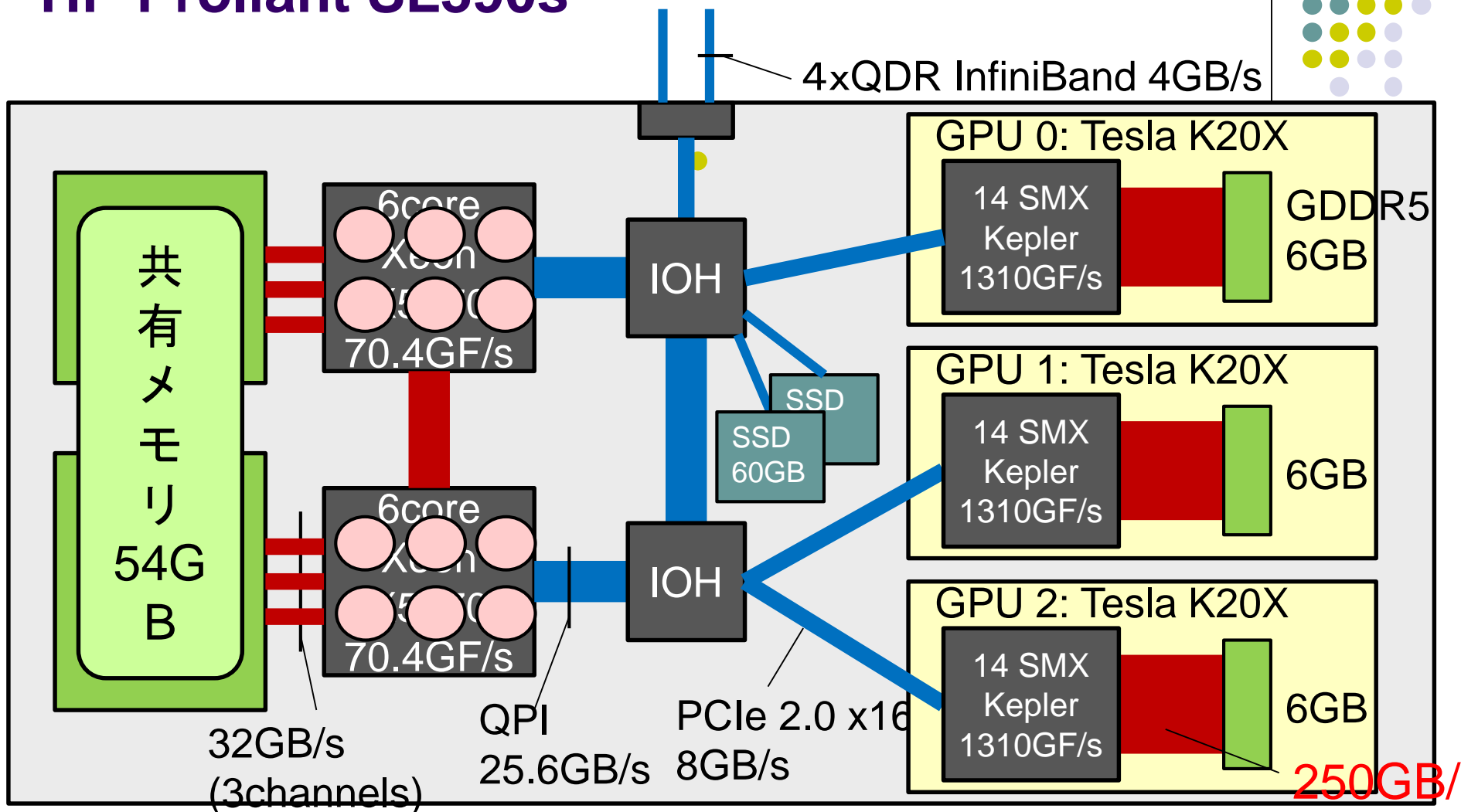
K20X GPU 1つあたりの性能

- 計算速度: 1.31 TFlops (倍精度)、3.95 TFlops (単精度)
 - CPUは20~100GFlops程度
- コア数:
 - 14SMX x 192CUDAコア = 2688CUDAコア
- メモリ容量: 6GB
 - 2688コアが、**デバイスメモリ**と呼ばれる、6GBのメモリを**共有**している
 - メインメモリとは別
- メモリバンド幅: 約250 GB/s
 - CPUは10~50GB/s程度
- その他の特徴
 - キャッシュメモリ (L1, L2)
 - ECC
 - CUDA, OpenAcc, OpenCLなどでプログラミング

以前のGPUにはキャッシュメモリが無かったので、高速なプログラム作成がより大変だった

TSUBAME2.5のノードアーキテクチャ

HP Proliant SL390s



CPUの詳細情報は"/proc/cpuinfo"ファイルを参照
GPUの詳細情報は"nvidia-smi -a"コマンドで表示

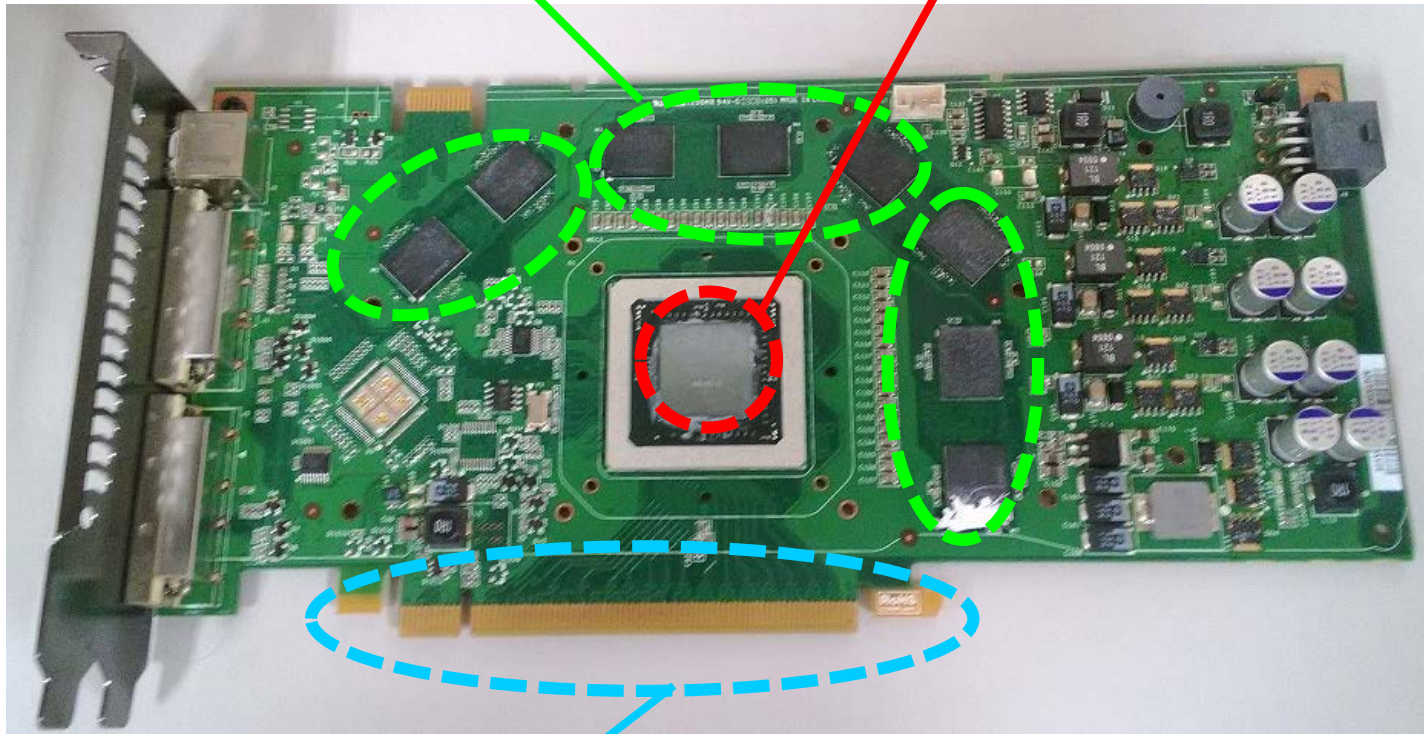


GPUボードの構成

- ふたとヒートシンクを外した様子

デバイスメモリ

GPUのプロセッサ



PCI-Expressコネクタ (マザーボードと接続)

様々なGPUやアクセラレータ



- NVIDIA GPU

- GeForceシリーズ: 一般のPCに搭載されているタイプで、比較的安価。パソコンショップで売っている
- Teslaシリーズ: GPUコンピューティング専用ハードウェア。
TSUBAME2に搭載されているのはTesla K20X
- Titan, Piz Daintスパコン等

- Intel Xeon Phi

- 約60コアプロセッサ。各コアはx86
- ボード上でLinux OSが動き、OpenMPも動く
- 天河二号、Stampedeスパコン等
- なお、2016年予定のKnights Landingは「単体で動く」



- AMD/ATI GPU

- 東芝・Sony・IBM Cellプロセッサ

- プレイステーション3に搭載

トップスパコンでの アクセラレータ利用



Intel Xeon Phi
NVIDIA GPU

www.top500.org
2015/11ランキング

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	DOE/NNSA/LANL/SNL United States	Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	301,056	8,100.9	11,078.9	
7	Swiss National Supercomputing Centre (CSCS) Switzerland	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
8	HLRS - Höchstleistungsrechenzentrum Stuttgart Germany	Hazel Hen - Cray XC40, Xeon E5-2680v3 12C 2.5GHz, Aries interconnect Cray Inc.	185,088	5,640.2	7,403.5	
9	King Abdullah University of Science and Technology Saudi Arabia	Shaheen II - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	196,608	5,537.0	7,235.2	2,834
10	Texas Advanced Computing Center/Univ. of Texas United States	Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510

- 24位のTSUBAME2.5もNVIDIA GPU利用
- 東大+筑波大の次期スパコンOakforest-PACSは(新)Xeon Phi₁₂

アクセラレータ向けプログラミング言語



- **CUDA** (本講義でとりあげる)
 - NVIDIA GPU向けのプログラミング言語
- OpenACC
 - お手軽なGPUプログラミングのために最近提案された
 - CPU用プログラムに、「ヒント」を追加。OpenMPに似ている
 - TSUBAMEのPGIコンパイラでも利用可能
- OpenCL
 - NVIDIA GPU, AMD GPU, 普通のIntelマルチコアCPUでも動く
 - ただし、CUDAよりさらに複雑な傾向
- OpenMP
 - Xeon Phi上ではOpenMPも動く

プログラミング言語CUDA

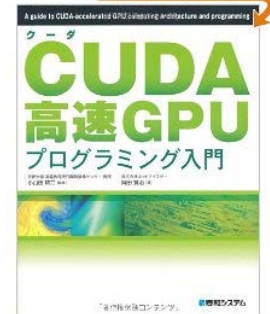


- NVIDIA GPU向けのプログラミング言語
 - 2007年2月に最初のリリース
 - TSUBAME2で使えるのはV7.5
 - 基本的に1GPU向け → 多数GPUはCUDA+MPIなどで
- 標準C言語サブセット＋GPGPU用拡張機能
 - C言語の基本的な知識(特にポインタ)は必要となります
 - Fortran版もあり
- **nvccコマンド**を用いてコンパイル
 - ソースコードの拡張子は**.cu**

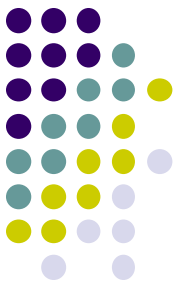
CUDA関連書籍は多数

クリック なか見! 検索

著者は東工大の
青木先生・額田先生



公式マニュアル(英語) <http://docs.nvidia.com/cuda/>
ほか、web上に初心者向けの情報も多数



CUDAプログラムのコンパイルと実行例

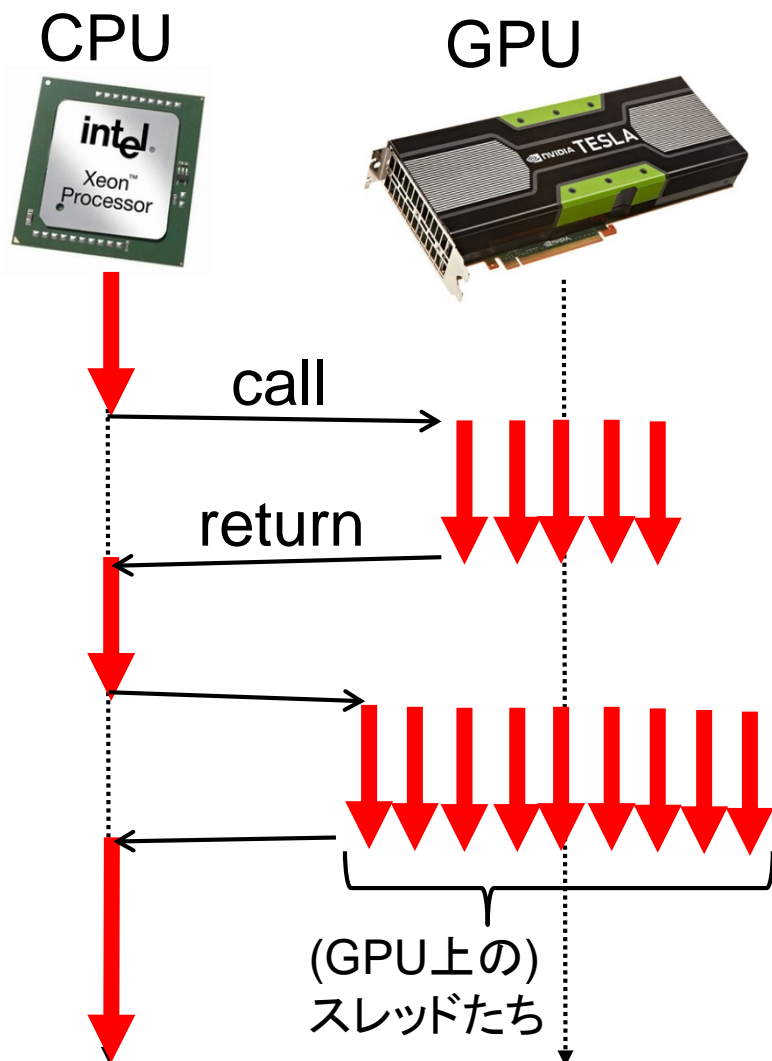
- ~endo-t-ac/ppcomp/16/cuda ディレクトリ
- サンプルプログラム inc_seq.cu などをコピーしてください
- 以下のコマンドをターミナルから入力し、CUDAプログラムのコンパイル、実行を確認してください
 - “\$” はコマンドプロンプトです

```
$ nvcc -arch sm_35 inc_seq.cu -o inc_seq  
$ ./inc_seq
```

- -arch sm_35 は、最新のCUDA機能を使うためのオプション (TSUBAMEでは普段つけておくとよいかも)

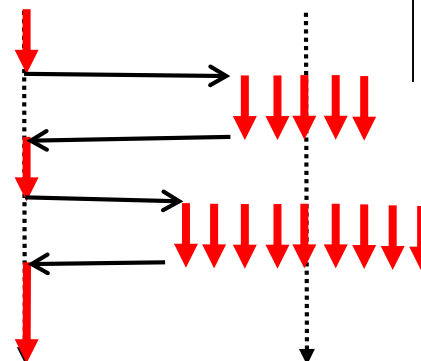
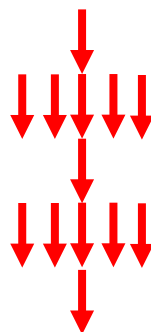
CUDAのプログラミングモデル

～分散メモリと共有メモリの両方登場～



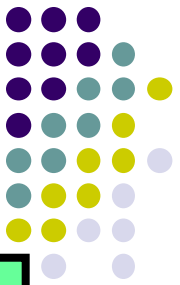
- `main()`から開始し、当初はCPUだけ動く
 - GPUは、CPUから処理を依頼された時だけ動く
 - GPU上で動く関数 = GPUカーネル関数
- CPUとGPU間は(原則)別のメモリ空間
 - ここは分散メモリ
- GPU上では多数のスレッド達が動き、スレッド達の間は共有メモリ

CUDAとOpenMPの比較



	OpenMP	CUDA
並列実行に使うプロセッサ	CPU	GPU
並列実行箇所の指定	#pragmaをブロックまたは文につける	GPUカーネル関数として切り出す (pthreadの考え方が近い?)
スレッド数の指定	環境変数・通常固定	呼び出しごとに変えられる
望ましいスレッド数	CPUコア数以下	CUDAコア数以上
メモリの扱い	すべて共有	GPU上スレッド間には共有、CPUとGPU間(または異なるGPU間)では分散

CUDAプログラム構成



ホスト関数

+

GPUカーネル関数

- 二種類の関数がcuファイル内に混ざっている
- ホスト関数
 - CPU上で実行される関数
 - ほぼ通常のC言語。main関数から処理がはじまる
 - GPUに対してデータ転送やGPUカーネル関数呼び出しを実行
- GPUカーネル関数
 - GPU上で実行される関数
 - ホストプログラムから呼び出されて実行
 - (単にカーネル関数と呼ぶ場合も)



サンプルプログラム: cuda/inc_seq.cu

int型配列の全要素を1加算

GPUを使う意味がない
(速くない)例ですが

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

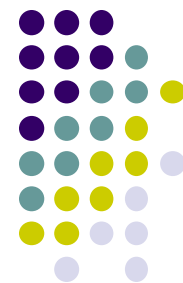
```
#define N (32)
__global__ void inc(int *array, int len)
{
    int i;
    for (i = 0; i < len; i++)
        array[i]++;
    return;
}
```

```
int main(int argc, char *argv[])
{
    int i;
    int arrayH[N];
    int *arrayD;
    size_t array_size;
```

```
    for (i=0; i<N; i++) arrayH[i] = i;
    printf("input: ");
    for (i=0; i<N; i++)
        printf("%d ", arrayH[i]);
    printf("\n");

    array_size = sizeof(int) * N;
    cudaMalloc((void **)&arrayD, array_size);
    cudaMemcpy(arrayD, arrayH, array_size,
               cudaMemcpyHostToDevice);
    inc<<<1, 1>>>(arrayD, N);
    cudaMemcpy(arrayH, arrayD, array_size,
               cudaMemcpyDeviceToHost);
    printf("output: ");
    for (i=0; i<N; i++)
        printf("%d ", arrayH[i]);
    printf("\n");
    return 0;
}
```

典型的なCUDAプログラムの流れ



CPU上

GPU上

(1) GPU側メモリにデータ用領域を確保

(2) 入力データをGPUへ転送

(3) GPUカーネル関数を呼び出し

(5) 出力をCPU側メモリへ転送

GPUカーネル関数の印

```
__global__ void kernel_func()  
{  
    ↓ (4)カーネル関数  
    return, 実行  
}
```

入力

出力

入力

出力

CPU側メモリ(メインメモリ)

GPU側メモリ(デバイスメモリ)

この2種類のメモリの
区別は常におさえておく

inc_seqの流れ(1)

CPU上: GPU側メモリ領域確保



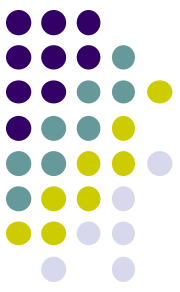
- `cudaMalloc(void **devpp, size_t count)`
 - GPU側メモリ(**デバイスメモリ**と呼ばれる)に領域を確保
 - `devpp`: デバイスメモリアドレスへのポインタ。確保したメモリのアドレスが書き込まれる
 - `count`: 領域のサイズ

例: 長さ32のintの配列を確保

```
int *arrayD;  
cudaMalloc((void **)&arrayD, sizeof(int) * 32);  
// arrayD has the address of allocated device memory
```

inc_seqの流れ(2)

CPU上: 入力データ転送



- `cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)`
 - 先に`cudaMalloc`で確保した領域に指定したCPU側メモリのデータをコピー
 - `dst`: 転送先デバイスメモリ
 - `src`: 転送元CPUメモリ
 - `count`: 転送サイズ(バイト単位)
 - `kind`: 転送タイプを指定する定数。ここでは`cudaMemcpyHostToDevice`を与える

例: CPU上のデータ`arrayH`の内容を、先ほど確保した`arrayD`へ転送

```
int arrayH[32];
cudaMemcpy(arrayD, arrayH, sizeof(int)*32,
            cudaMemcpyHostToDevice);
```

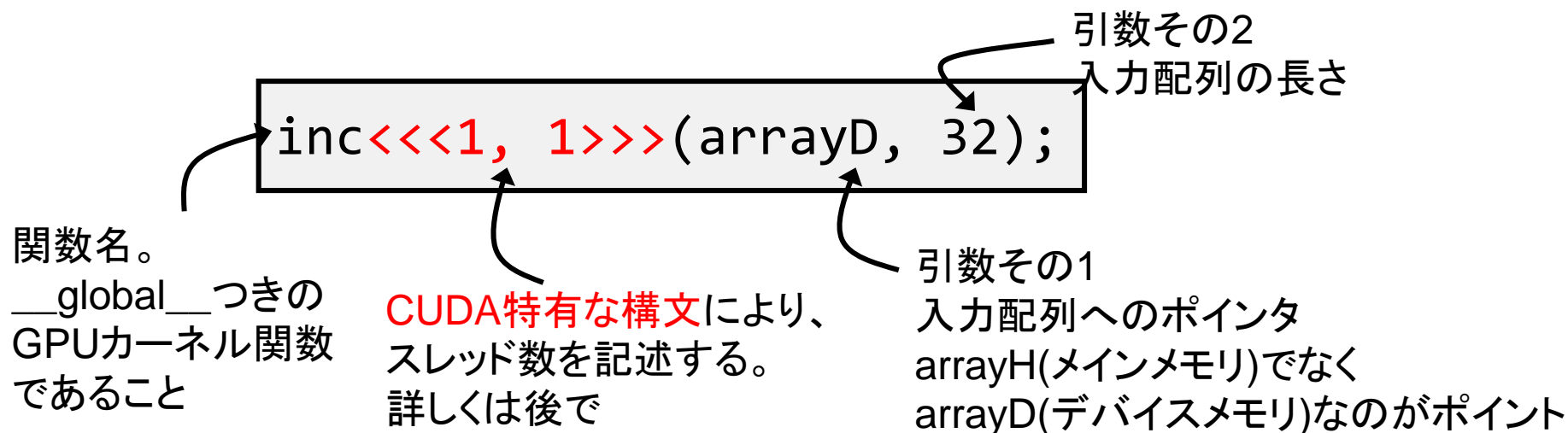
inc_seqの流れ(3)

CPU上: GPUカーネルの呼び出し



- `kernel_func<<<grid_dim, block_dim>>>(kernel_param1, ...);`
 - `kernel_func`: カーネル関数名
 - `kernel_param`: カーネル関数の引数

例: カーネル関数 “inc” を呼び出し



inc_seqの流れ(4)

GPU上: カーネル関数の実行



- GPU上で実行される関数
 - `__global__`というキーワードをつける
注:「global」の前後にはアンダーバー2つずつ
- GPU側メモリのみアクセス可、CPU側メモリはアクセス不可
- 引数利用可能
- 値の返却は不可 (voidのみ)

例: int型配列をインクリメントするカーネル関数(でも逐次)

```
__global__ void inc(int *array, int len)
{
    int i;
    for (i = 0; i < len; i++) array[i]++;
    return;
}
```

inc_seqの流れ(5)

CPU上: 結果配列をGPUから取得



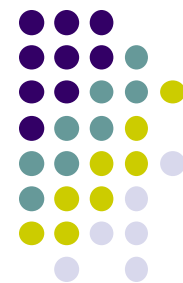
- 入力転送と同様にcudaMemcpyを用いる
- ただし、転送タイプは cudaMemcpyDeviceToHostを指定

例: 結果の配列をCPU側メモリへ転送

```
cudaMemcpy(arrayH, arrayD, sizeof(int)*32,  
            cudaMemcpyDeviceToHost);
```

もう使わないデバイスメモリ上の領域を、
`cudaFree(arrayD);`
で解放

GPUカーネル関数内でできること・ できないこと (1)



- if, for, while, returnなどの制御構文はok
- GPU側メモリのアクセスはok、CPU側メモリのアクセスは不可
 - inc_seqサンプルで、arrayDと間違っarrayHをカーネル関数に渡してしまうとバグ!! (何が起こるか分からない)
- ファイルアクセス、MPI呼び出し、各種システムコールは不可
 - GPUの外界に関わることは原則できない
 - **printfは例外的にok**なので、デバグに役立つ
- カーネル関数内のmallocは、小サイズならok
 - CPU関数がmalloc → メインメモリ上
 - CPU関数がcudaMalloc → デバイスメモリ上
 - GPUカーネル関数がmalloc → デバイスメモリ上
- sin, sqrtなどの基本的な数値計算関数あり
- libcと同様のrandはなし → 使い方の異なるcurandライブラリ

GPUカーネル関数内でできること・ できないこと (2)

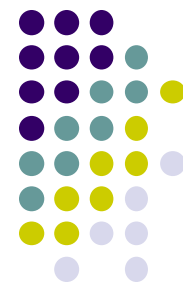


- 関数呼び出しは、「__device__つき関数」に対してならok
 - __device__関数は再起呼び出しもok (初期のCUDAではだめだった)



- 上図の矢印の方向にのみ呼び出しできる
 - GPU内からCPU関数は呼べない
- __device__つき関数は、返り値を返せるので便利
- __global__つき関数は、**必ず返り値void型**

CUDAにおける並列化

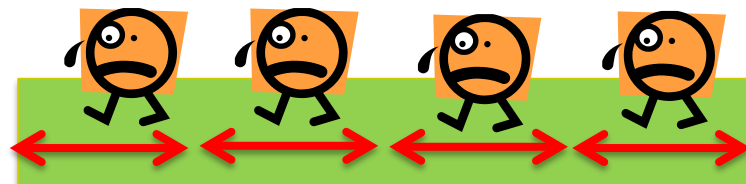


- **たくさんのスレッドがGPU上で並列に動作することにより、初めてGPUを有効活用できる**
 - inc_seqプログラムは1スレッドしか使っていない
- データ並列性を基にした並列化が一般的
 - 例：巨大な配列があるとき、各スレッドが一部ずつを分担して処理 → 高速化が期待できる

一人の小人が大きな畑を耕す場合

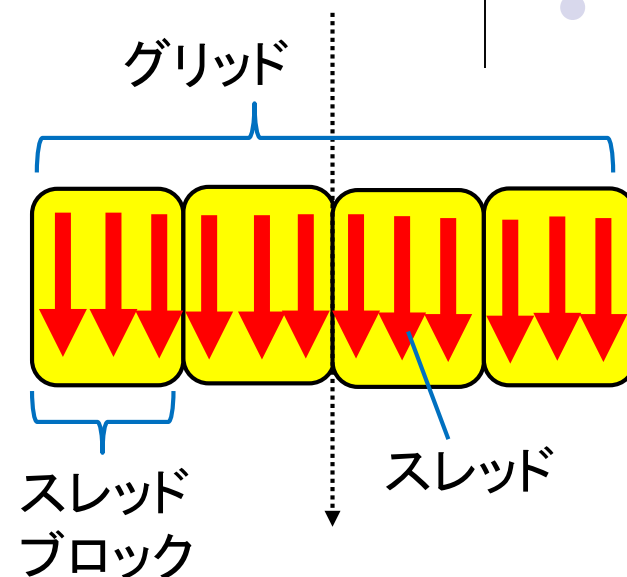


複数の小人が分担して耕すと速く終わる



CUDAにおけるスレッド

- CUDAでのスレッドは階層構造になっている
 - **グリッド**は、複数の**スレッドブロック**から成る
 - **スレッドブロック**は、複数の**スレッド**から成る
- カーネル関数呼び出し時に数値を二段階で指定

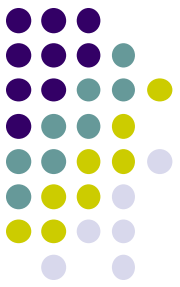


例: `kernel_func<<<100, 30>>>(a, b, c);`

スレッドブロックの数

(スレッドブロックあたりの)
スレッドの数

- この例では、**100x30=3000個(!)のスレッド**が `kernel_func`を並列に実行する

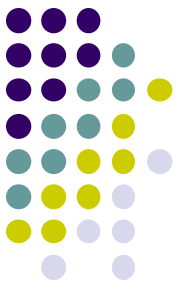


自分のスレッド番号を知るには

- GPUカーネル関数内^内で、以下の特殊な変数を見ることができる
- 自分のID
 - blockIdx.x: 自分が何番目のブロックにいるか(0以上)
 - threadIdx.x: 自分がブロック内で何番目のスレッドか(0以上)
- スレッド数など
 - gridDim.x: 全体でいくつブロックがあるか
 - blockDim.x: 各ブロックにいくつのスレッドがあるか

注: 通し番号を表す変数はない

→ $\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$ で計算



サンプルプログラムの改良

inc_parは、inc_seqと同じ計算を行うが、
N要素の計算のためにNスレッドを利用する点が違う

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

#define N (32)
#define BS (8)
__global__ void inc(int *array, int len)
{
    int i = blockIdx.x * blockDim.x +
           threadIdx.x;
    array[i]++;
    return;
}

int main(int argc, char *argv[])
{
    int i;
    int arrayH[N];
    int *arrayD;
    size_t array_size;
```

```
    for (i=0; i<N; i++) arrayH[i] = i;
    printf("input: ");
    for (i=0; i<N; i++)
        printf("%d ", arrayH[i]);
    printf("\n");

    array_size = sizeof(int) * N;
    cudaMalloc((void **)&arrayD, array_size);
    cudaMemcpy(arrayD, arrayH, array_size,
               cudaMemcpyHostToDevice);
    inc<<<N/BS, BS>>>(arrayD, N);
    cudaMemcpy(arrayH, arrayD, array_size,
               cudaMemcpyDeviceToHost);
    printf("output: ");
    for (i=0; i<N; i++)
        printf("%d ", arrayH[i]);
    printf("\n");
    return 0;
}
```



inc_parプログラムのポイント (1)

- N要素の計算のためにNスレッドを利用

inc<<<N/BS, BS>>>(.);

グリッドサイズ

スレッドブロックサイズ
この例では、前もってBS=8とした

ちなみに、<<<N, 1>>>や
<<<1, N>>>でも動くのだ
が非効率的である。

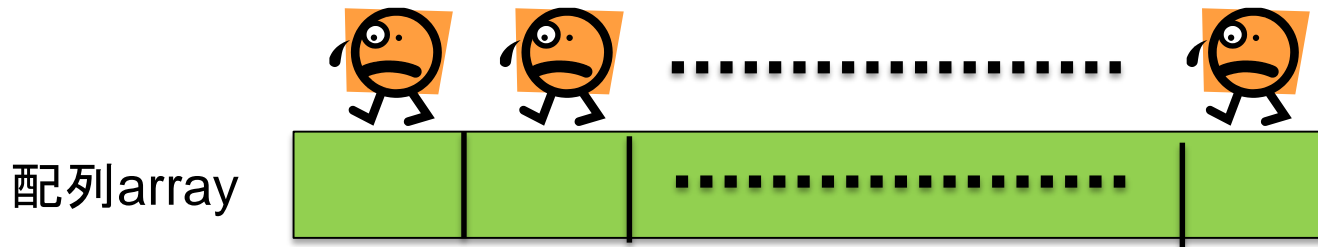
ちなみに、このままでは、NがBSで割
り切れないときに正しく動かない。ど
う改造すればよいか？

inc_parプログラムのポイント (2)



inc_parの並列化の方針

- (通算で)0番目のスレッドにarray[0]の計算をさせる
- 1番目のスレッドにarray[1]の計算
- N-1番目のスレッドにarray[N-1]の計算



- 各スレッドは「自分は通算で何番目のスレッドか?」を知るために、下記を計算

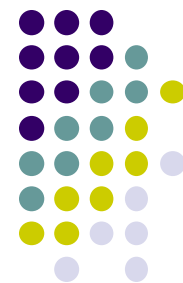
$i = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$

使いまわせる
便利な式

- 1スレッドは"array[i]"の1要素だけ計算 → forループは無し

変数・メモリに関するルール

~OpenMPに近いことが多い~

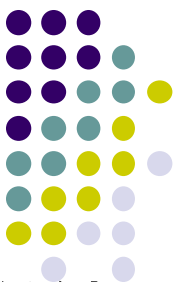


- カーネル関数内で宣言される変数は、各スレッド独自の値を持つ
 - あるスレッドでは $i=0$, 別のスレッドでは $i=1\cdots$
- カーネル関数に与えられた引数は、全スレッド同じ値
 - inc_parプログラムでは、arrayポインタとlen
- 全スレッドはGPU側メモリを共有しており、読み書きできる
 - ただし、複数スレッドが同じ場所に書き込むとぐちゃぐちゃ (race condition)になるので注意
 - 同じ場所を読み込むのはok



本授業のレポートについて

- 各パートで課題を出す。2つ以上のパートのレポート提出を必須とする
 - 予定パート:
 - OpenMPパート
 - MPIパート
 - GPUパート
- サンプルプログラムについては、TSUBAMEの
~endo-t-ac/ppcomp/16/ 以下から各自コピーすること



GPUパート課題説明 (1)

以下のG1, G2, G3の、いずれかについてレポートを提出してください

[G1] 行列積サンプルmm-cudaの性能を、行列サイズを変化させながら性能評価してください

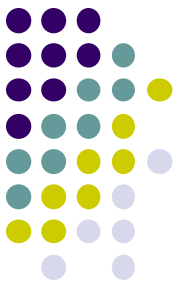
- CPU(OpenMP)版の性能とも比較してください。
- データ転送コストを考慮に入れる場合・入れない場合それぞれについて速度を示すこと
 - 転送コストが相対的に大きくなるのはどういう場合か。計算量オーダー、転送量オーダーにも触れて議論すること
- プログラムを改良してもok
 - 各スレッドの担当領域の変更、共有メモリによる高速化、
 - 小行列の組み合わせ(ブロッキング)、などなど



GPUパート課題説明 (2)

[G2] diffusionサンプルプログラムをGPUを用いて並列化し、性能評価してください。

- 参考プログラム: advection-cuda
- 改良してもok。たとえば
 - Divergent分岐の影響の削減
 - Shared memoryの利用による高速化
 - マルチGPUの利用
 - ほか



GPUパート課題説明 (3)

[G3] 自由課題: 任意のプログラムを, GPU を用いて並列化し、性能評価してください

- たとえば、過去のSuperConの本選問題
<http://www.gsic.titech.ac.jp/supercon/>
たんぱく質類似度(2003), N体問題(2001)・・・
入力データは自分で作る必要あり
- たとえば、行列積をOpenACCで記述したときの性能は？
- たとえば、Reduction処理を含むpiサンプルをGPU化できるか？
- たとえば、自分が研究している問題

課題の注意



- いずれの課題の場合も、レポートに以下を含むこと
 - 計算・データの割り当て手法の説明
 - TSUBAME2などで実行したときの性能
 - プロセッサ(コア)数を様々に変化させたとき
 - 問題サイズを様々に変化させたとき(可能な問題なら)
 - 「XXコア以上で」「問題サイズXXX以上で」発生する問題に触れているとなお良い
 - 高性能化・機能追加などのための工夫が含まれているとなお良い
 - 「XXXのためにXXXを試みたが高速にならなかった」のような失敗でもgood
 - 作成したプログラムも提出
 - zipなどで圧縮してOCW-ilに提出
 - 困難な場合は、TSUBAME2の自分のホームディレクトリに置き、置き場所を連絡(パーミッションに注意)



課題の提出について

- GPUパート提出期限
 - 6/16(木)
 - OpenMPパート(5/19)、MPIパート(6/13)にも注意
- OCW-i ウェブページから下記ファイルを提出のこと
- レポート形式
 - 本文: PDF, Word, テキストファイルのいずれか
 - プログラム: zip形式に圧縮するのがのぞましい
- OCW-iからの提出が困難な場合、メールでもok
 - 送り先: `ppcomp@el.gsic.titech.ac.jp`
 - メール題名: `ppcomp report`

次回

- GPUパート(2)
 - スレッド数の指定のコツ
 - 行列積サンプル

