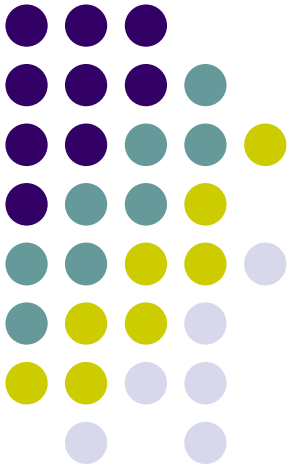
2015年度
実践的並列コンピューティング
第12回

GPUプログラミング (1)

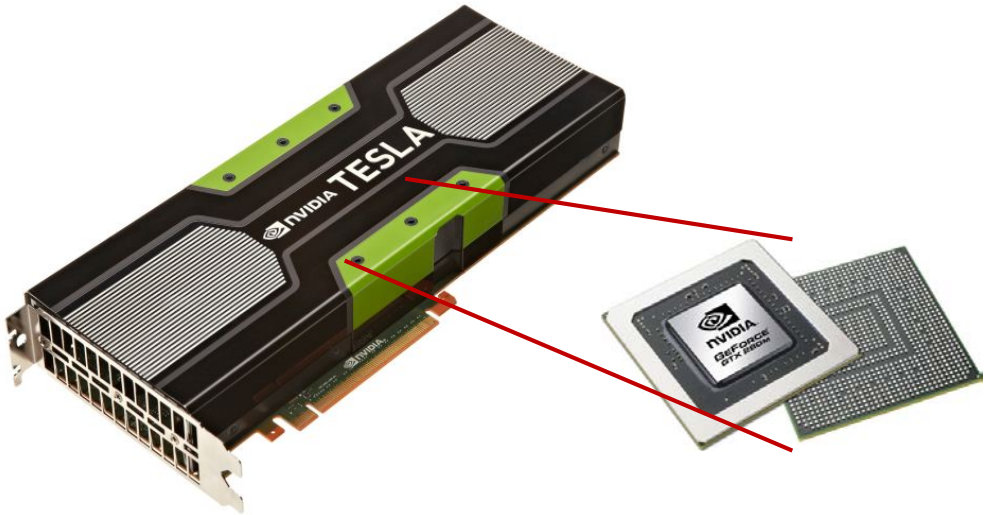


遠藤 敏夫

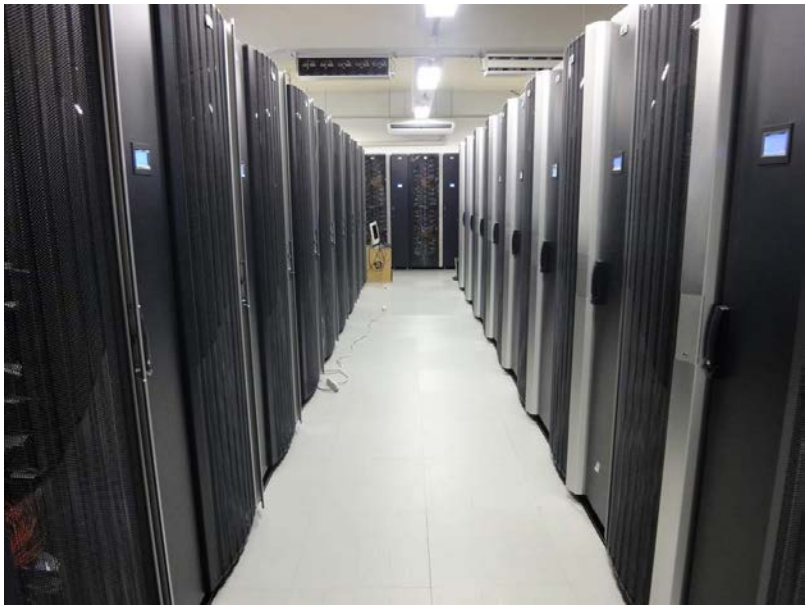
endo@is.titech.ac.jp

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

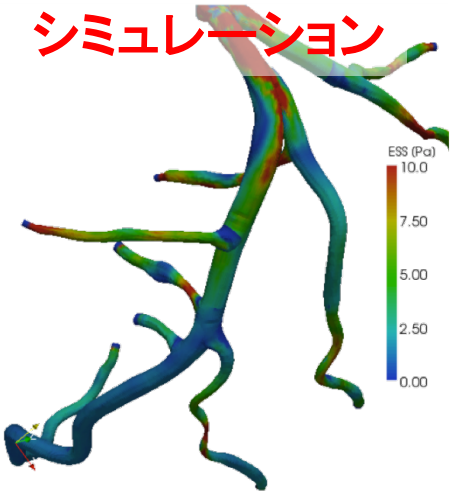
- TSUBAME1: 2006年～2010年に稼働したスパコン
- TSUBAME2.0: 2010年に稼働開始したスパコン
  - 2010年当初には、世界4位、日本1位の計算速度性能
- TSUBAME2.5: 2013年にGPUを最新へ入れ替え
  - 現在、世界13位、日本2位

高性能の秘訣が
GPUコンピューティング

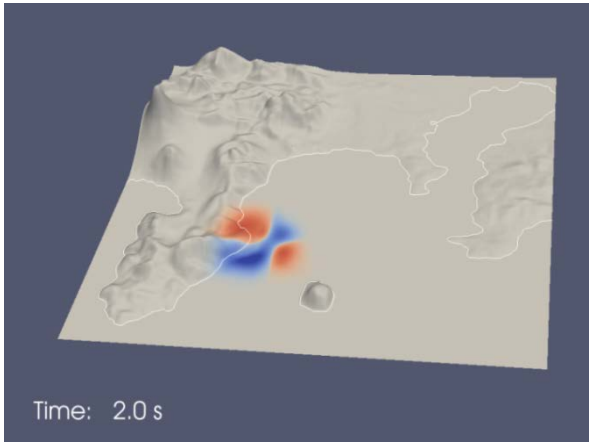
TSUBAME2.0スパコン・GPUは様々な研究分野で利用されている

気象シミュレーション

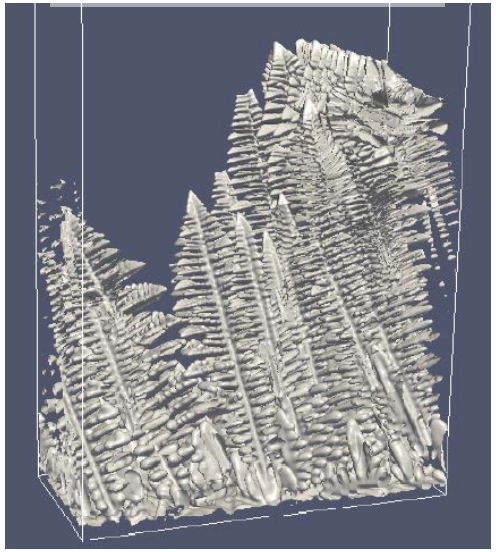
動脈血流
シミュレーション



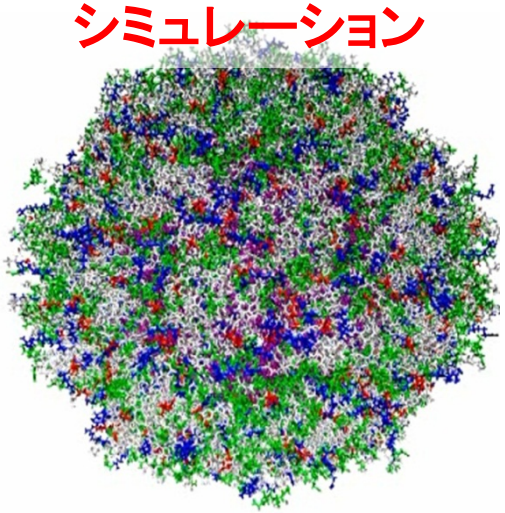
津波・防災
シミュレーション



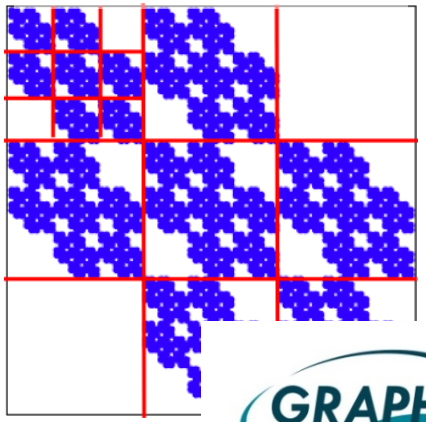
金属結晶凝固
シミュレーション



ウィルス分子
シミュレーション



グラフ構造解析



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 200m^2 (2,100 ft^2)

Processor

CPU

Intel Xeon X5670
(Westmere-EP)
- 2.93 GHz
- 76.8 GFLOPS

GPU

UPGRADE

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)

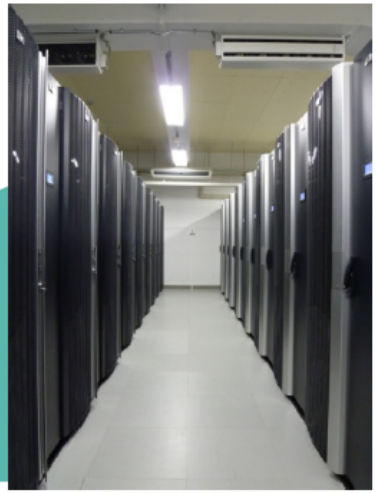
Node Chassis

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)

Integrated by NEC

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

    CPU 0.07TFlops x 2 + GPU 1.31TFlops x 3 = 4.08TFlops

96%の性能がGPUのおかげ

- メインメモリ(CPU側メモリ): 54GB
- 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

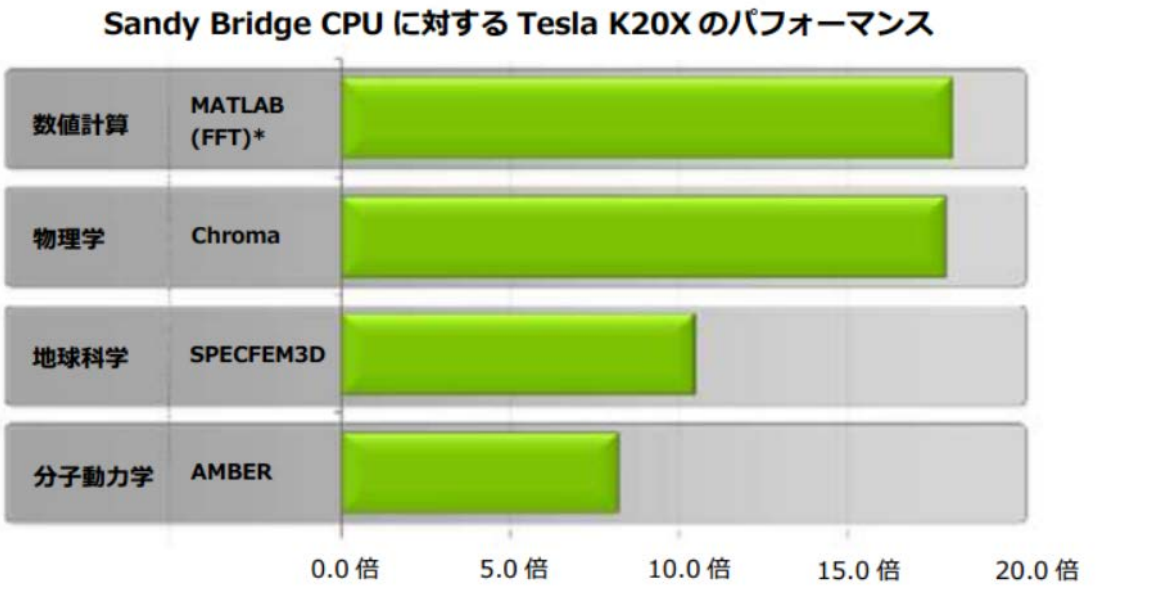
GPUの特徴 (2)

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、6GBのメモリを共有している。ホストメモリとは別
- 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

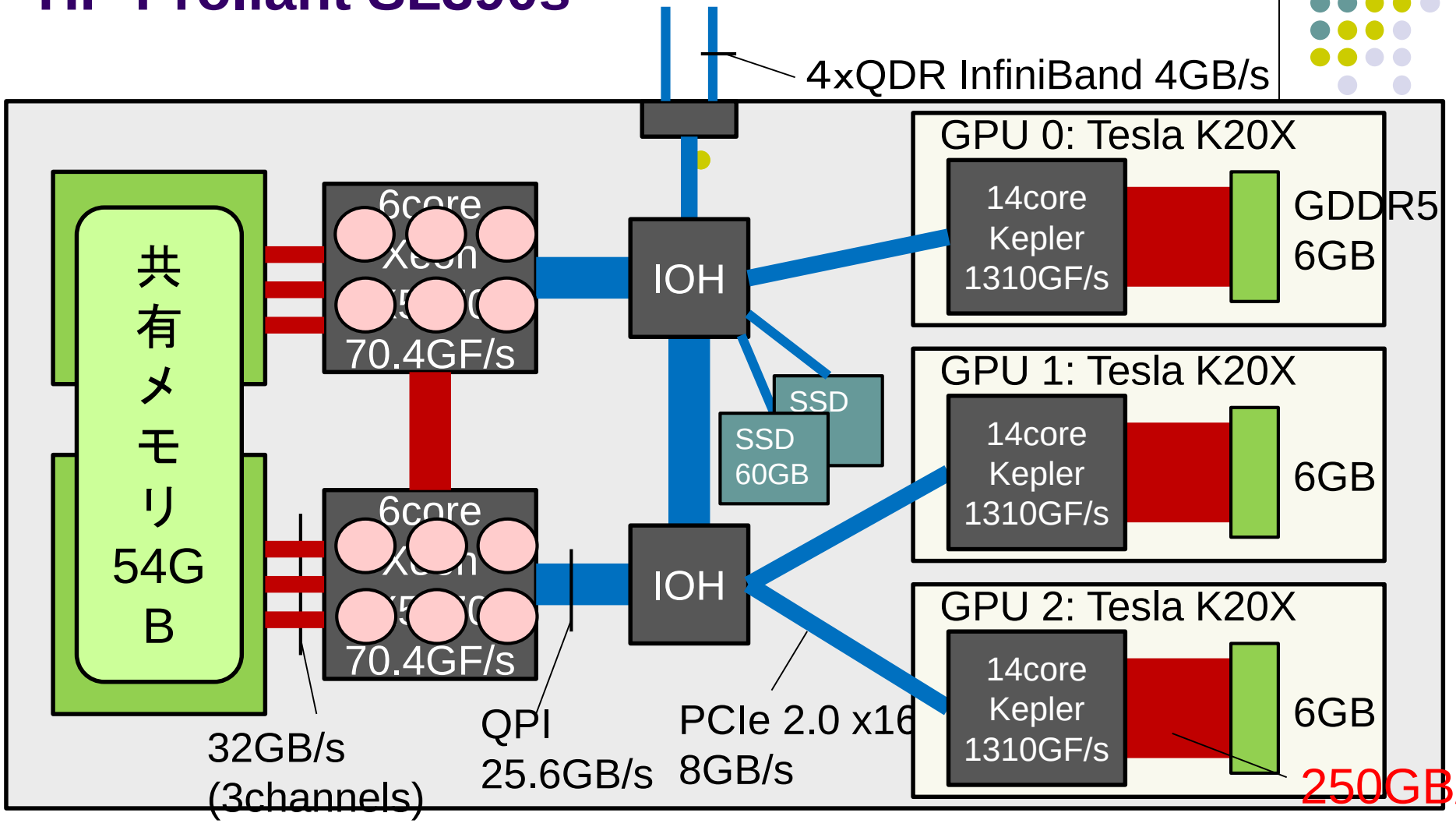
GPUの性能



NVIDIAの公開資料より

- CPU版の、同じ計算をするプログラムより数倍高速
  - CPU版もすでに並列化されている(はず)
- 宣伝通りにいくかどうかは、計算の性質とプログラミングの最適化しだい
  - どうしてもGPUに向かない計算はある

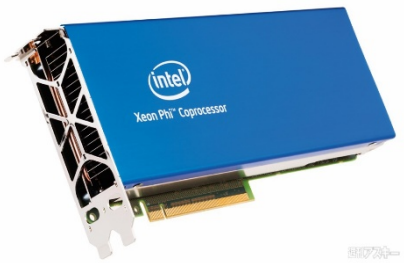
TSUBAME2.5のノードアーキテクチャ
HP Proliant SL390s



HyperThreadingのため，2ソケット×6コア×2HT=24プロセッサに見える．/proc/cpuinfoファイルの内容を参照

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, Nebulaeスパコン等
- Intel Xeon Phi
  - 約60コアプロセッサのボード
  - ボード上でLinux OSが動き、OpenMPも動く
  - 天河二号、Stampedeスパコン等
- 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



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	3120000	33862.7	54902.4	17808
2	DOE/SC/Oak Ridge National Laboratory United States	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590.0	27112.5	8209
3	DOE/NNSA/LLNL United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	17173.2	20132.7	7890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510.0	11280.4	12660
5	DOE/SC/Argonne National Laboratory United States	Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786432	8586.6	10066.3	3945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115984	6271.0	7788.9	2325
7	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	462462	5168.1	8520.1	4510
8	Forschungszentrum Juelich (FZJ) Germany	JUQUEEN - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458752	5008.9	5872.0	2301
9	DOE/NNSA/LLNL United States	Vulcan - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393216	4293.3	5033.2	1972
10	Government United States	Cray XC30, Intel Xeon E5-2697v2 12C 2.7GHz, Aries interconnect Cray Inc.	225984	3143.5	4881.3	
11	Exploration & Production - Eni S.p.A. Italy	HPC2 - iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, NVIDIA K20x IBM	62640	3003.0	4006.3	1067
12	Leibniz Rechenzentrum Germany	SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM	147456	2897.0	3185.1	3423
13	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.5 - Cluster Platform SL390s G7, Xeon X5670 6C 2.93GHz, Infiniband QDR, NVIDIA K20x	76032	2785.0	5735.7	1399

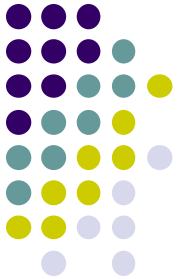
トップスパコンでのアクセラレータ利用

Intel Xeon Phi

NVIDIA GPU

www.top500.org
2014/6ランキング

アクセラレータ向けプログラミング言語

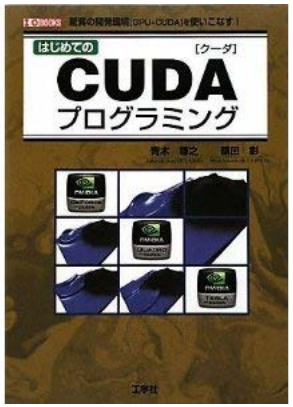


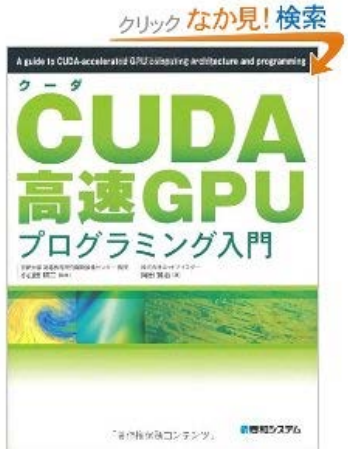
- CUDA (本講義でとりあげる)
  - NVIDIA GPU向けのプログラミング言語
- OpenACC
  - お手軽なGPUプログラミングのために最近提案された
  - CPU用プログラムに、「ヒント」を追加。OpenMPに似ている
  - TSUBAMEのPGIコンパイラでも利用可能
- OpenCL
  - NVIDIA GPU, AMD GPU, 普通のIntelマルチコアCPUでも動く
  - ただし、CUDAよりさらに複雑な傾向
- Xeon Phi用ディレクティブ
  - 並列Region部分をPhi, それ以外をCPUで実行
- OpenMP
  - Xeon Phi上では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V6.5
  - 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関連書籍もあり

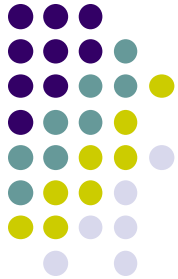




著者は東工大の
青木先生・額田先生

http://gpu-computing.gsic.titech.ac.jp/ にも多数情報あり

CUDAプログラムのコンパイルと実行例

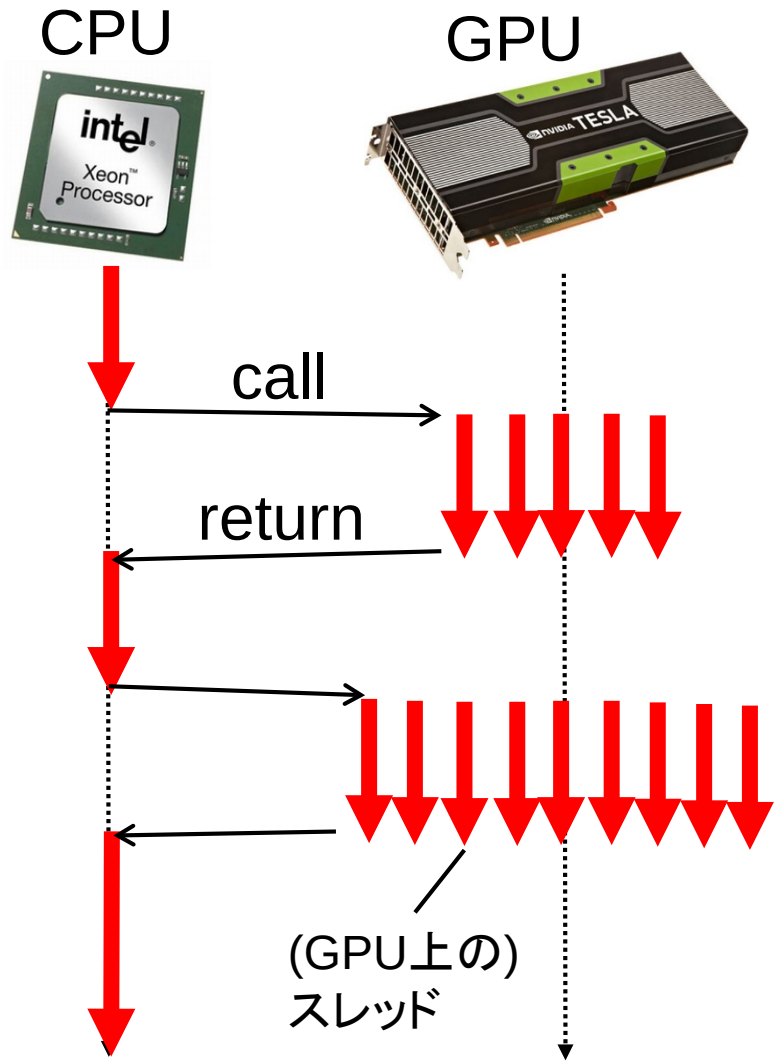


- ~endo-t-ac/ppcomp/15/cuda ディレクトリ
- サンプルプログラム inc_seq.cu などをコピーしてください
- 以下のコマンドをターミナルから入力し、CUDAプログラムのコンパイル、実行を確認してください
  - “$” はコマンドプロンプトです

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

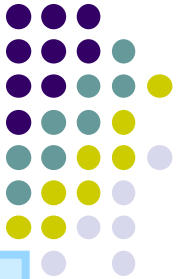
- -arch sm_35 は、最新のCUDA機能を使うためのオプション (TSUBAMEでは普段つけておくとよいかも)

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

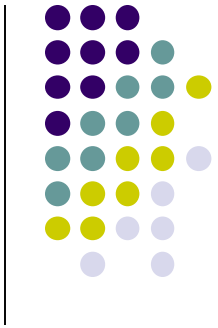
CUDAプログラム構成

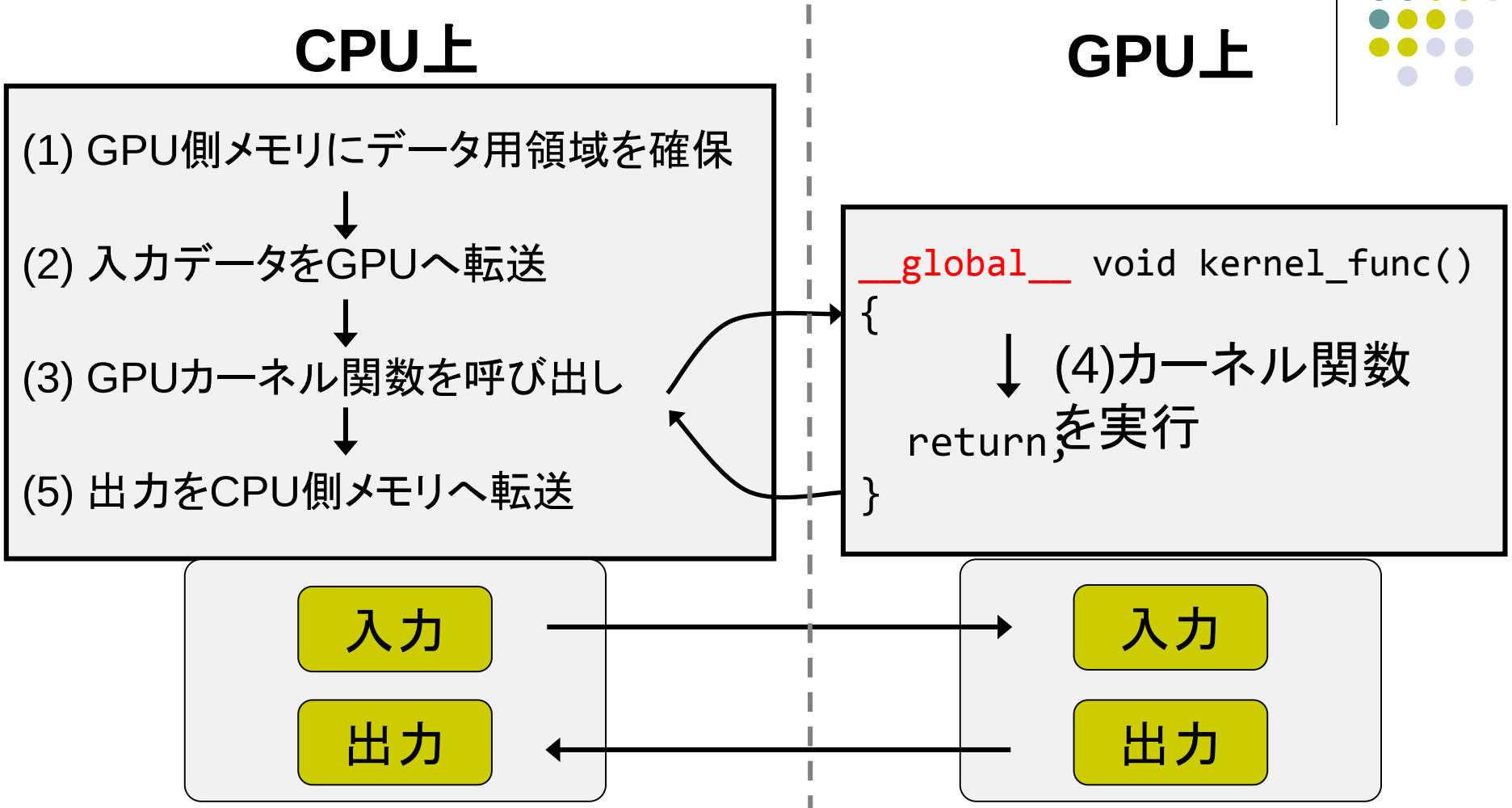


ホスト関数 ＋ GPUカーネル関数

- 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、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 (サンプルではinc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)

典型的な制御とデータの流れ





サンプルプログラム: inc_seq.cu

int型配列の全要素を1加算

GPUであまり意味がない(速くない)例ですが

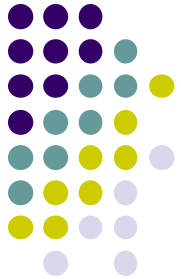
```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

#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;
}
```

(1) CPU上: GPU側メモリ領域確保

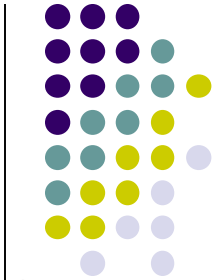


- `cudaMalloc(void **devpp, size_t count)`
  - GPU側メモリ(*デバイスメモリ*、*グローバルメモリ*と呼ばれる)に領域を確保
  - devpp: デバイスメモリアドレスへのポインタ。確保したメモリのアドレスが書き込まれる
  - count: 領域のサイズ
- `cudaFree(void *devp)`
  - 指定領域を開放

例: 長さ1024のintの配列を確保

```
#define N (1024)
int *arrayD;
cudaMalloc((void **)&arrayD, sizeof(int) * N);
// arrayD has the address of allocated device memory
```

(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: 先に確保した領域へCPU上のデータarrayHを転送

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

(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

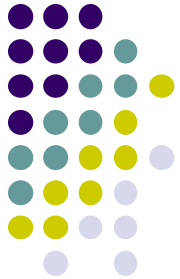
```
inc<<<1, 1>>>(arrayD, N);
```

引数その2
入力配列の長さ

引数その1
入力配列へのポインタ

CUDA特有な構文により、
スレッド数を記述する。
詳しくは後で

(4) GPU上: カーネル関数



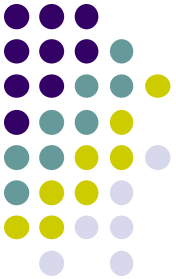
- GPU上で実行される関数
  - __global__というキーワードをつける

  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

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

(5) CPU上: 結果の返却



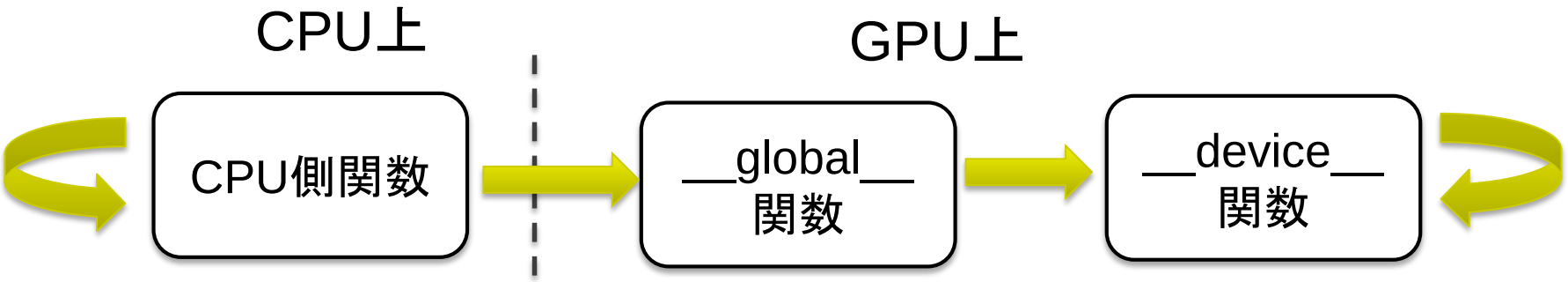
- 入力転送と同様にcudaMemcpyを用いる
- ただし、転送タイプは cudaMemcpyDeviceToHost を指定

例： 結果の配列をCPU側メモリへ転送

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

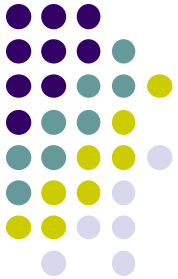
カーネル関数内でできること・できないこと

- if, for, while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と間違って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)
- ファイルアクセスなどは不可
  - printfは例外的にokなので、デバグに役立つ
- 関数呼び出しは、「__device__つき関数」に対してならok



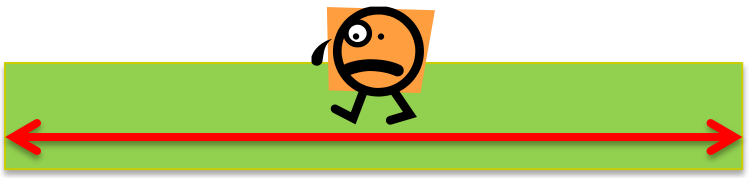
- 上図の矢印の方向にのみ呼び出しできる
  - GPU内からCPU関数は呼べない
- __device__つき関数は、返り値を返せるので便利

CUDAにおける並列化

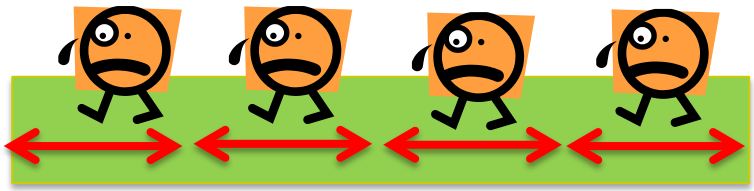


- たくさんのスレッドがGPU上で並列に動作することにより、初めてGPUを有効活用できる
  - inc_seqプログラムは1スレッドしか使っていない
- データ並列性を基にした並列化が一般的
  - 例：巨大な配列があるとき、各スレッドが一部づつを分担して処理 → 高速化が期待できる

一人の小人が大きな畑を耕す場合

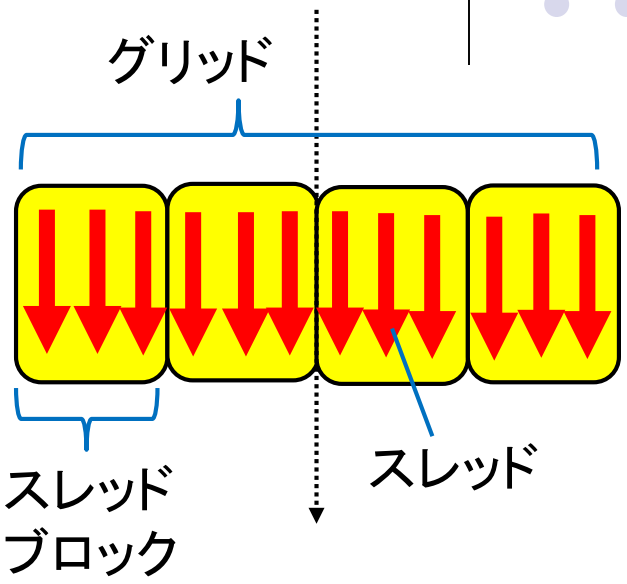


複数の小人が分担して耕すと速く終わる



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



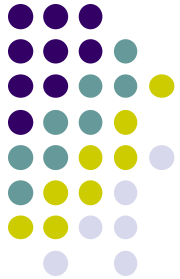

```
例：kernel_func<<<100, 30>>>(a, b, c);
```

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



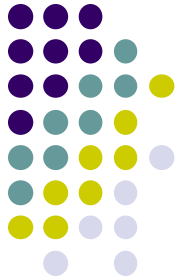
- GPUカーネル関数内で、以下の特殊な変数を見ることができる
- 自分のID
  - blockIdx.x: 自分が何番目のブロックにいるか(0以上)
  - threadIdx.x: 自分がブロック内で何番目のスレッドか(0以上)
- スレッド数など
  - gridDim.x: 全体でいくつブロックがあるか
  - blockDim.x: 各ブロックにいくつのスレッドがあるか

注:通し番号を表す変数はない

→ blockIdx.x * blockDim.x + 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？

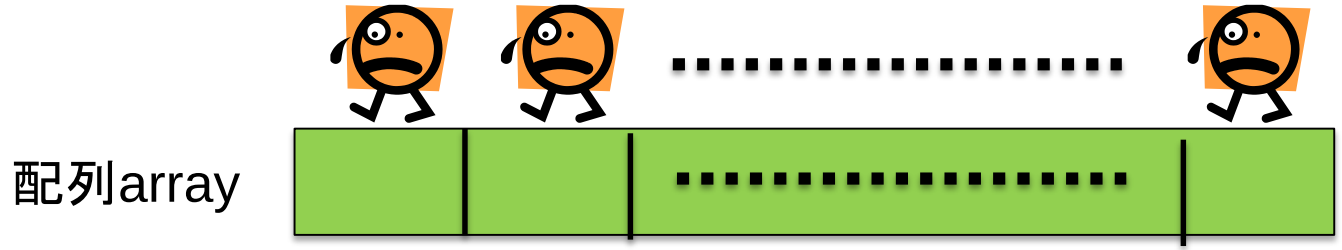
inc_parプログラムのポイント (2)

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

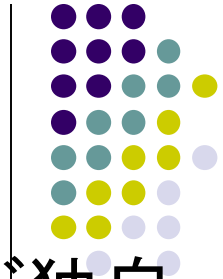


- 各スレッドは「自分は通算で何番目のスレッドか?」を知るために、下記を計算

  i = blockIdx.x * blockDim.x + threadIdx.x;

使いまわせる
便利な式

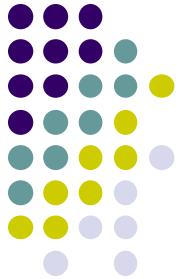
- 1スレッドは”array[i]”の1要素だけ計算 → forループは無し



変数・メモリに関するルール

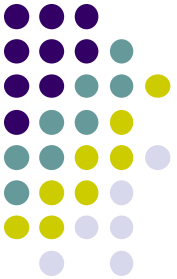
- カーネル関数内で宣言される変数は、各スレッド独自の値を持つ
  - あるスレッドではi=0, 別のスレッドではi=1・・・
- カーネル関数に与えられた引数は、全スレッド同じ値
  - inc_parプログラムでは、arrayポインタとlen
- 全スレッドはGPU側メモリを共有しており、読み書きできる
  - ただし、複数スレッドが同じ場所に書き込むとぐちゃぐちゃ(race condition)になるので注意
  - 同じ場所を読み込むのはok

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
  - OpenMPパート
    - ノード内のCPUコアを使う並列プログラミング
  - MPIパート
    - 複数ノードを使う並列プログラミング
  - GPU(CUDA)パート
    - 1GPU内の数百コアを使う並列プログラミング

GPUパート課題説明 (1)

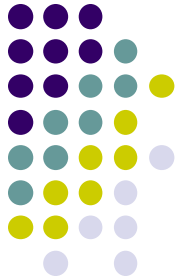


以下のG1, G2, G3の、いずれかについてレポートを提出してください.

[G1] 行列積プログラムの性能を、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)版とも比較してください。

- データ転送コストを速度計算に入れる場合・入れない場合それぞれについて測定
  - 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
- GPU版とCPU版の性能比についても調べること。差が大きいとき、小さいときはどういう場合か
- プログラムを改良してもok

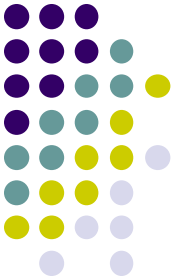
GPUパート課題説明 (2)



[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。

- CPU1コア版は
  ~endo-t-ac/ppcomp/15/diffusion/
- 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

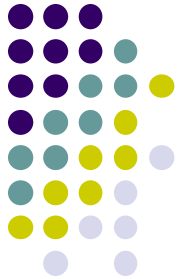
GPUパート課題説明 (3)



[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，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
    - スレッド数、スレッドブロック数を様々に変化させたときの変化に触れているとなお良い
    - 問題サイズを様々に変化させたとき(可能な問題なら)
  - 高性能化のための工夫が含まれているとなお良い
    - 「XXXのためにXXXをしてみたが高速にならなかった」のような失敗でも可
  - プログラムについては，zipなどで圧縮して添付
    - 困難な場合，TSUBAME2の自分のホームディレクトリに置き，置き場所を連絡

課題の提出について

- GPUパート提出期限
  - 8/10 (月) 23:50
  - MPIパート〆切7/13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

次回以降

- 7/13(月)
  - 特別講義：ソフトウェア高速化のためのメモリシステム
  - MPI+CUDA実用ソフトウェア例(予定)
- 7/20(月）祝日ですが授業日
  - CUDAによるGPUプログラミング (2)

  - スケジュールについてはOCW pageも参照
    - http://www.el.gsic.titech.ac.jp/~endo/

    → 2015年度前期情報(OCW) → 講義ノート