担当
- 大島聡史（助教） ohshima@cc.u-tokyo.ac.jp
- 星野哲也（助教） hoshino@cc.u-tokyo.ac.jp

質問やサンプルプログラムの提供についてはメールでお問い合わせください

# 第58回お試しアカウント付き並列プログラミング講習会「GPUプログラミング入門」

2016年6月8日（水）

東京大学情報基盤センター

# スケジュール

- 09：00 – 09：30　受付
- 09：30 – 12：00　GPU入門，CUDA入門
- 13：00 – 14：30　OpenACC入門 + HA-PACSログイン
- 14：45 – 16：15　OpenACC最適化入門と演習
- 16：30 – 18：00　CUDA最適化入門と演習

# はじめに

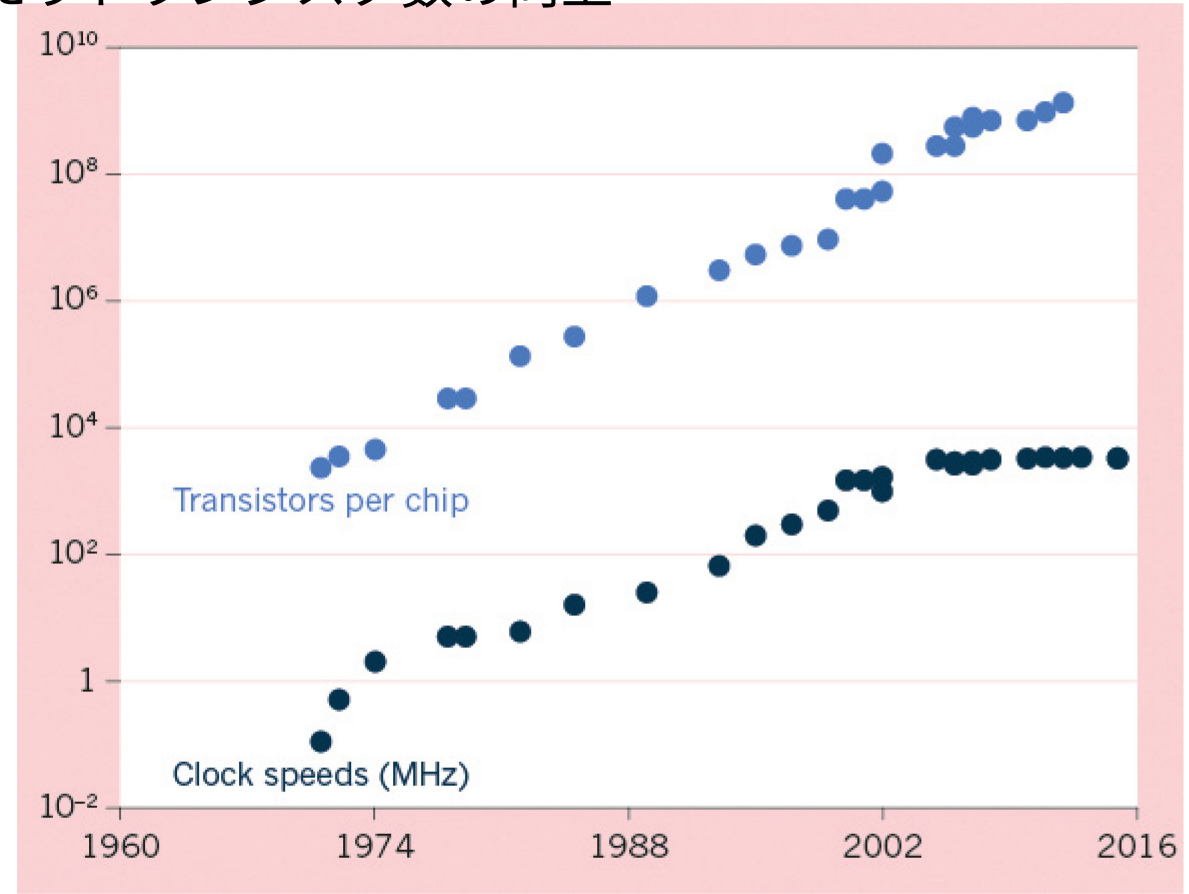
GPUについて
GPUスパコン事情
Reedbushシステムの紹介

# 多様化する並列計算環境

- 現在のHPC・計算機科学・計算科学分野では様々な並列計算ハードウェアが利用されている
  - マルチコアCPU：複数の計算コアを1つのチップにまとめたCPU
    - 代表例：Intel Xeon / Core, AMD Opteron/FX, IBM POWER, FUJITSU SPARC64, ARM Cortex
      - サーバ向けでは1999年POWER4、PC向けでは2005用Dual-Core Opteron/AthlonX2が初出と言われている
  - メニーコアプロセッサ：マルチコアCPUよりも多数の計算コアを搭載
    - 代表例：Intel Xeon Phi, Sun Niagara, PEZY PEZY-1/SC
      - 明確に何コア以上がメニーコアという定義が有るわけではない
  - GPU：画像処理用HWに端を発するメニーコアプロセッサ
    - 代表例：NVIDIA Tesla/GeForce, AMD FirePro/Radeon
  - FPGA：プログラミングにより回路構成を変更可能なプロセッサ
    - 代表例：Xilinx Virtex, Altera Stratix

# 背景：計算ハードウェアの性能向上

- ムーアの法則に支えられたCPUの性能向上が終わりつつある
  - 微細化によるチップあたりトランジスタ数の向上
  - クロック周波数の向上
    ↓
  - 消費電力や発熱が
    問題となり頭打ち
    ↓
  - マルチコア化、
    メニーコア化による
    並列演算性能の向上へ




出展 : The chips are down for Moore's law : Nature News & Comment
http://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338

# GPU (Graphics Processing Unit)

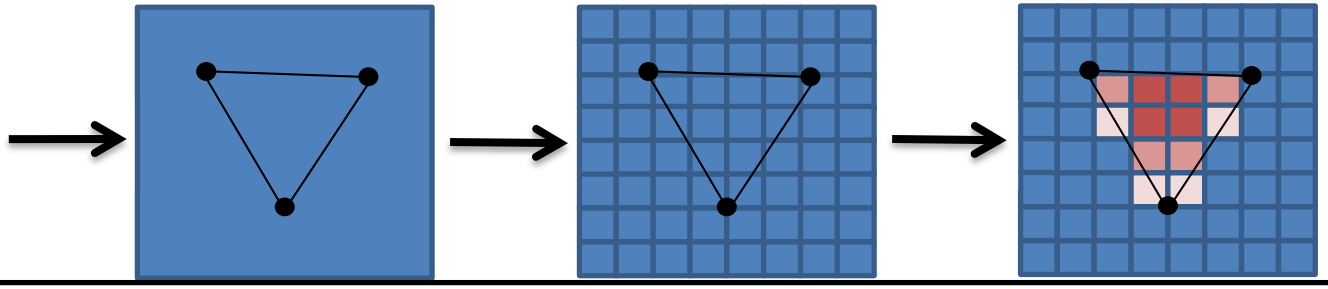
- 画像処理用のハードウェア
  - 高速・高解像度描画、3D描画処理（透視変換、陰影・照明）、画面出力
- CPUやマザーボードに組み込まれたチップとして、また拡張スロットに搭載するビデオカードとして広く利用される
- GPUに求められる処理が並列計算に適した処理であったため、CPUに先んじて並列化による高性能化が進んだ
- 性能・機能の向上に伴い2000年代後半から汎用演算への活用が進み、GPGPUやGPUコンピューティングと呼ばれる

General-Purpose computation on GPUs)

※参考

3次元画像描画の手順

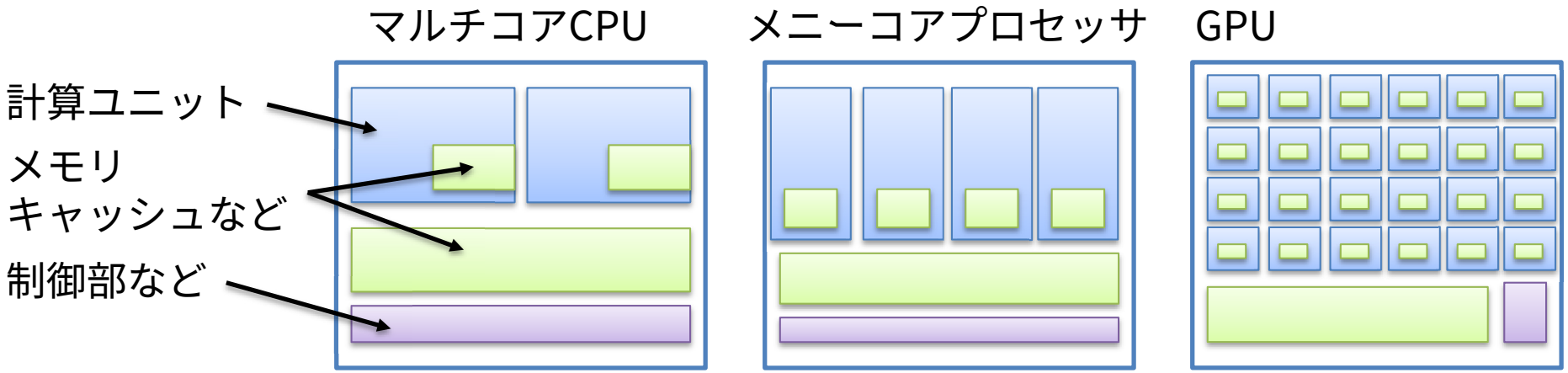
① (2, 2)
② (8, 3)
③ (5, 7)



オブジェクト単位、頂点単位、ピクセル単位で並列処理が可能→並列化により高速化しやすい

# GPUの特徴（1/2）

- ハードウェアの構成バランスの違い（イメージ）
  - 限られたトランジスタを主に何に用いるか




  - 多数の計算ユニットを搭載し全体として高性能を得ることを重視
  - （この図ではわからないが）総メモリ転送性能も重視している

# GPUの特徴（2/2）

- CPUとは異なる特徴を持つ
  - 非常に多くの（1000以上）の計算ユニットを搭載、計算ユニット単体の性能は低い
    - 動作周波数、キャッシュ、分岐、……
  - 計算コアが完全に個別には動けない
    - 32個などの単位でスケジューリング、SIMD演算器が大量に搭載されたイメージ
  - 浅めのキャッシュ階層、複数階層のメモリ
- 特定のアプリケーションでは非常に高い性能
  - ビッグデータや機械学習の分野で有用なため、最近特に注目されている
- CPUとは異なるプログラミング・最適化の知識と技術が必要
  - 本講習会がその手助け・入り口となることを期待します

# GPUスパコン

- GPU等の「アクセラレータ」を搭載したスパコンの普及
  - TOP500 (2015.11) TOP20中8, TOP500中100以上がGPUスパコン

TOP 10 Sites for November 2015

For more information about the sites and systems in the list, click on the links or view the complete list.

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

# 東大情報基盤センター 現在運用中のスパコン

## Oakleaf-FX (通常ジョブ用) (Fujitsu PRIMEHPC FX10)

| | |
|---|---|
| Total Peak performance | : 1.13 PFLOPS |
| Total number of nodes | : 4800 |
| Total memory | : 150 TB |
| Peak performance / node | : 236.5 GFLOPS |
| Main memory per node | : 32 GB |
| Disk capacity | : 1.1 PB + 2.1 PB |
| **SPARC64 Ixfx 1.84GHz** | |

## Oakbridge-FX (長時間ジョブ用) (Fujitsu PRIMEHPC FX10 )

| | |
|---|---|
| Total Peak performance | : 136.2 TFLOPS |
| Total number of nodes | : 576 |
| Total memory | : 18 TB |
| Peak performance / node | : 236.5 GFLOPS |
| Main memory per node | : 32 GB |
| Disk capacity | : 147TB + 295TB |
| **SPARC64 Ixfx 1.84GHz** | |

## Yayoi (Hitachi SR16000/M1)

| | |
|---|---|
| Total Peak performance | : 54.9 TFLOPS |
| Total number of nodes | : 56 |
| Total memory | : 11200 GB |
| Peak performance / node | : 980.48 GFLOPS |
| Main memory per node | : 200 GB |
| Disk capacity | : 556 TB |
| **IBM POWER 7 3.83GHz** | |

**Total Users > 2,000**

# 東大情報基盤センター 今後導入予定のスパコン

## Reedbush
(データ解析・シミュレーション融合スーパーコンピュータシステム)

- Reedbush-U (CPU only) と Reedbush-H (with GPU) からなる
- Reedbush-U
  - 508.03 TFlops
  - 2016/7/1 試験運用開始
- Reedbush-H
  - 1297.15-1417.15 TFlops
  - 2017/3/1 試験運用開始

## Oakforest-PACS

- 最先端共同HPC基盤施設(JCAHPC)により導入
  - JCAHPCは東大-筑波大の共同組織
- ピーク性能:25PFFLOPS
  - 8,208 Intel Xeon Phi (KNL)
  - 日本最速になる予定
- 2016/12/1 試験運用開始

JCAHPC

# Reedbush (Mini PostT2K改め) (1/2)

- システム構成・運用：SGI
- Reedbush-U (CPU only)
  - Intel Xeon E5-2695v4 (Broadwell-EP, 2.1GHz 18core,) x 2ソケット (1.210 TF), 256 GiB (153.6GB/sec)
  - InfiniBand EDR, Full bisection BW Fat-tree
  - システム全系: 420 ノード, 508.0 TF
- Reedbush-H (with GPU)
  - CPU・メモリ：Reedbush-U と同様
  - NVIDIA Tesla P100 (Pascal世代 GPU)
    - (4.8-5.3TF, 720GB/sec, 16GiB) x 2 / ノード
  - InfiniBand FDR x 2ch, Full bisection BW Fat-tree
  - 120 ノード, 145.2 TF(CPU)+ 1.15~1.27 PF(GPU)= 1.30~1.42 PF

# “Reedbush”って何？

Blaise Pascal
(1623–1662)

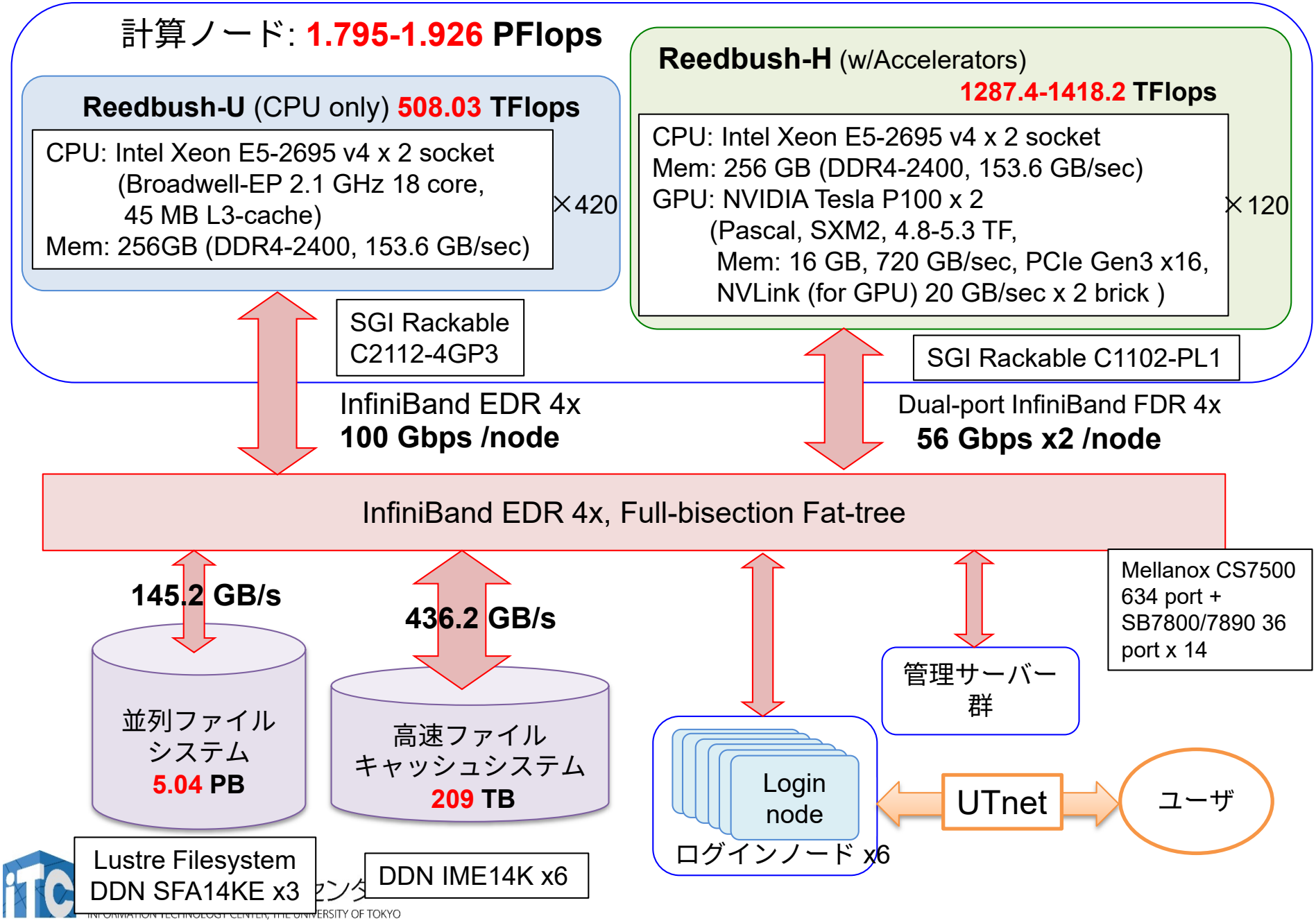
- L'homme est un roseau pensant.
- Man is a thinking reed.
- 人間は考える葦である

Pensées (Blaise Pascal)

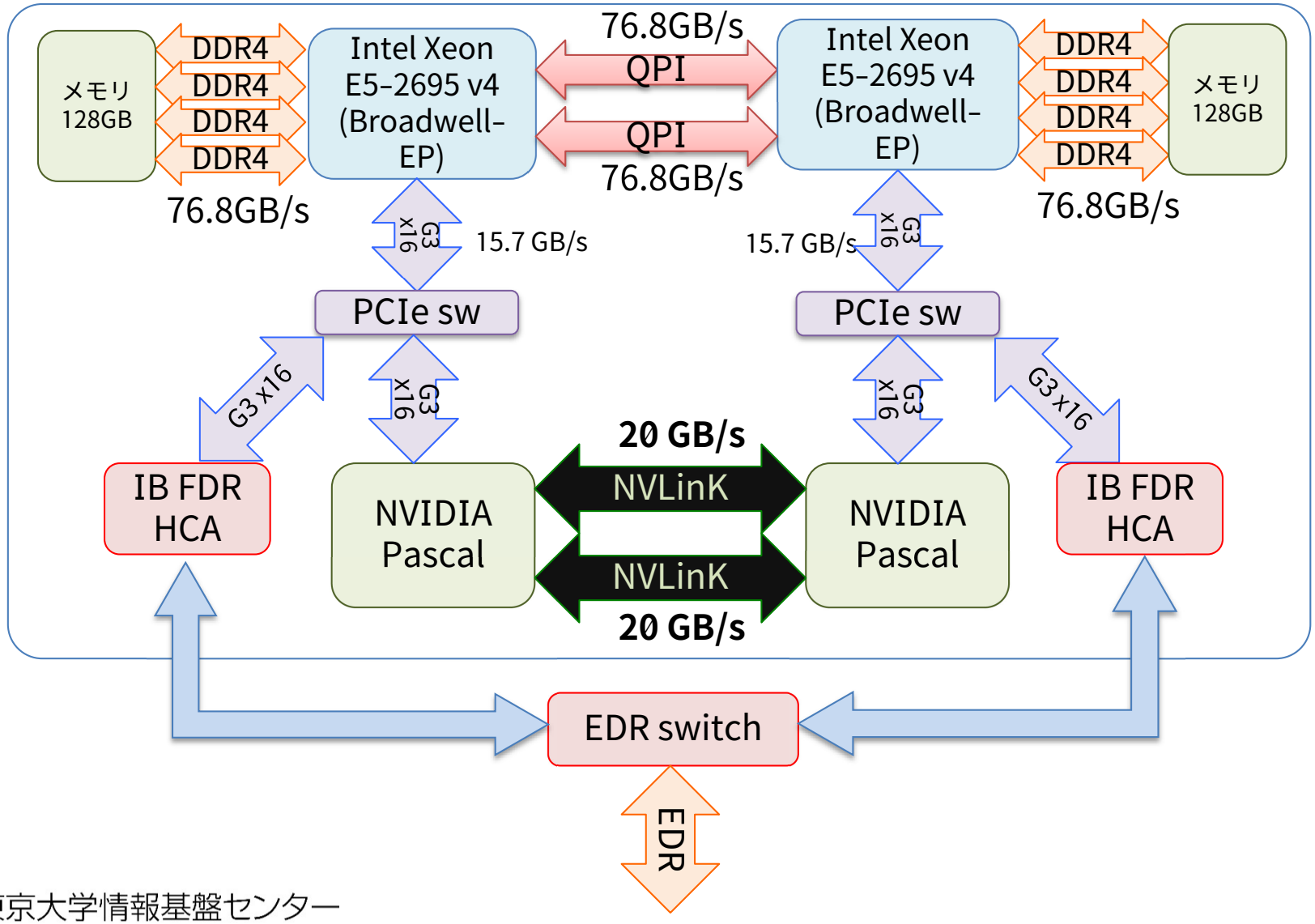
# Reedbush (Mini PostT2K改め) (2/2)

- ストレージ/ファイルシステム
  - 並列ファイルシステム (Lustre)
    - 5.04 PB, 145.2 GB/sec
  - 高速ファイルキャッシュシステム: Burst Buffer (DDN IME (Infinite Memory Engine))
    - SSD: 209.5 TB, 450 GB/sec
- 電力, 冷却, 設置面積
  - 空冷, 378 kVA(冷却除く)、< 90 $m^2$
- データ解析、ディープラーニング向けソフトウェア・ツールキット
  - OpenCV, Theano, Anaconda, ROOT, TensorFlow, Torch, Caffe, Chainer, GEANT4

利用申込み受付中、詳しくはWebをご参照ください
http://www.cc.u-tokyo.ac.jp/system/reedbush/


計算ノード: 1.795-1.926 PFlops
Reedbush-U (CPU only) 508.03 TFlops
CPU: Intel Xeon E5-2695 v4 x 2 socket
(Broadwell-EP 2.1 GHz 18 core,
45 MB L3-cache)
Mem: 256GB (DDR4-2400, 153.6 GB/sec)
×420
Reedbush-H (w/Accelerators)
1287.4-1418.2 TFlops
CPU: Intel Xeon E5-2695 v4 x 2 socket
Mem: 256 GB (DDR4-2400, 153.6 GB/sec)
GPU: NVIDIA Tesla P100 x 2
(Pascal, SXM2, 4.8-5.3 TF,
Mem: 16 GB, 720 GB/sec, PCIe Gen3 x16,
NVLink (for GPU) 20 GB/sec x 2 brick )
×120
SGI Rackable C2112-4GP3
SGI Rackable C1102-PL1
InfiniBand EDR 4x
100 Gbps /node
Dual-port InfiniBand FDR 4x
56 Gbps x2 /node
InfiniBand EDR 4x, Full-bisection Fat-tree
145.2 GB/s
436.2 GB/s
Mellanox CS7500
634 port +
SB7800/7890 36
port x 14
並列ファイル
システム
5.04 PB
高速ファイル
キャッシュシステム
209 TB
管理サーバー
群
Login
node
ログインノード x6
UTnet
ユーザ
Lustre Filesystem
DDN SFA14KE x3
DDN IME14K x6

# Reedbush-Hノードのブロック図



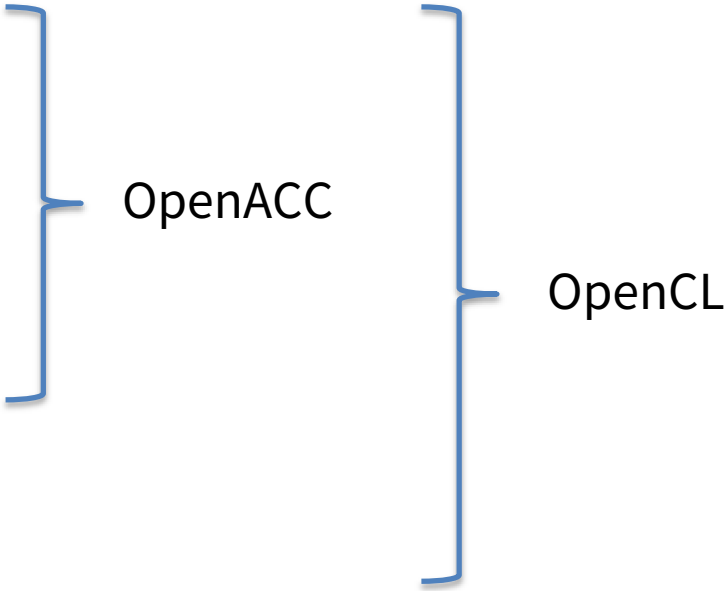
# GPUを利用する方法

1. GPUに対応したソフトウェア（アプリケーション）を使う
   - GPU上で行われる計算自体は実装しない、基本的にGPUの知識は不要
   - 存在するものしか使えない、手持ちのプログラムには適用不能
2. （GPUに対応していないプログラムから）GPUに対応したライブラリやフレームワークを使う
   - GPU上で行われる計算自体は実装しない、基本的にGPUの知識は不要
   - 対象分野における共通のAPIが存在しGPU化されていれば恩恵は大
     - BLASなどの数値計算ライブラリ、ビッグデータ・機械学習系のライブラリ・フレームワークなど
3. GPU上で行われる計算そのものを実装する
   - １や２で用いるソフトウェア・ライブラリ等そのものを作る
   - GPUに関する知識が必要
   - 手持ちのプログラム・独自のプログラムをGPU化できる

本講習会の対象

# 並列計算環境を利用するためのソフトウェア

- 主な開発環境（プログラミング言語など、特に並列化に用いるもの）
  - CPU/MIC
    - MPI, OpenMP
    - (pthread, Cilk+, TBB, …)
  - GPU
    - CUDA, DirectCompute
  - FPGA
    - Verilog HDL




  - 従来は個別のものが使われていたが、近年では共通化も進みつつある
    - 習得が大変、移植が大変という利用者の声が反映されている

# 本講習会で用いるハードウェアと開発環境

- 対象とするGPU：NVIDIA Tesla M2090
  - Tesla：NVIDIA社が開発しているGPUシリーズの1つ、HPC向け
    - コンシューマ向けのGeForceシリーズと比べて、倍精度演算が高速、ECC対応メモリを搭載、などの違いがある
    - M2090は2011年に発売されたGPUであり、アーキテクチャ名はFermi。現行のGPUと比べると古いが、GPUを用いた最適化プログラミングの基礎を学ぶには十分なもの。

- 対象とするGPUプログラミング開発環境：CUDAとOpenACC
  - CUDA (Compute Unified Device Architecture)：NVIDIAのGPU向け開発環境。C言語版はCUDA CとしてNVIDIAから、Fortran版はCUDA FortranとしてPGI(現在はNVIDIAの子会社)から提供されている。
  - OpenACC：指示文を用いて並列化を行うプログラミング環境。C言語とFortranの両方の仕様が定められている。PGIコンパイラなど幾つかのコンパイラが対応。（GPUが主なターゲットだが）GPU専用言語ではない。

# 参考：GPUプログラミングの歴史

- 2004年頃：GPU上である程度プログラミングが可能となった
  - 「プログラマブルシェーダ」が登場
  - それ以前は機能の切替程度しかできなかった
  - 主に画像処理のためのプログラミングであり、様々なアルゴリズムを実装するのに十分なものとは言えなかった
- 2006年頃：CUDAが登場、様々な制限はありつつも「普通のプログラム」が利用可能に
  - 様々なアルゴリズムが実装された、科学技術計算への応用も活発化
  - GPUスパコンの誕生
  - バージョンアップ（最新は7.5）により高機能化、制限の撤廃
- 2011年頃：OpenACCが提案される
  - CUDAより容易で汎用性のある（NVIDIA GPUに縛られない）プログラミング環境に対する要求の高まり
  - 最新仕様は2.5、実装されているのは2.1程度まで

# NVIDIA社のGPUラインナップについて（1/3）

- GeForce
  - コンシューマ向けグラフィックスカード
  - 主にゲーミングPCで使われる（＋最近は機械学習、VR？）
  - 単精度演算性能を重視（倍精度演算用のHWをあまり搭載していない）、クロック周波数が高めの傾向
  - 安価
- Quadro
  - ワークステーション用グラフィックスカード
  - （GeForceやTeslaと比べると注目されていない？）
- Tesla
  - HPC（科学技術計算、スパコン）向け
  - 画面出力できないモデルも多い（”Graphics” Processing Unit……？）
  - 倍精度演算性能も重視、クロック周波数が低めの傾向、ECCメモリ対応
  - 安価とは言えない

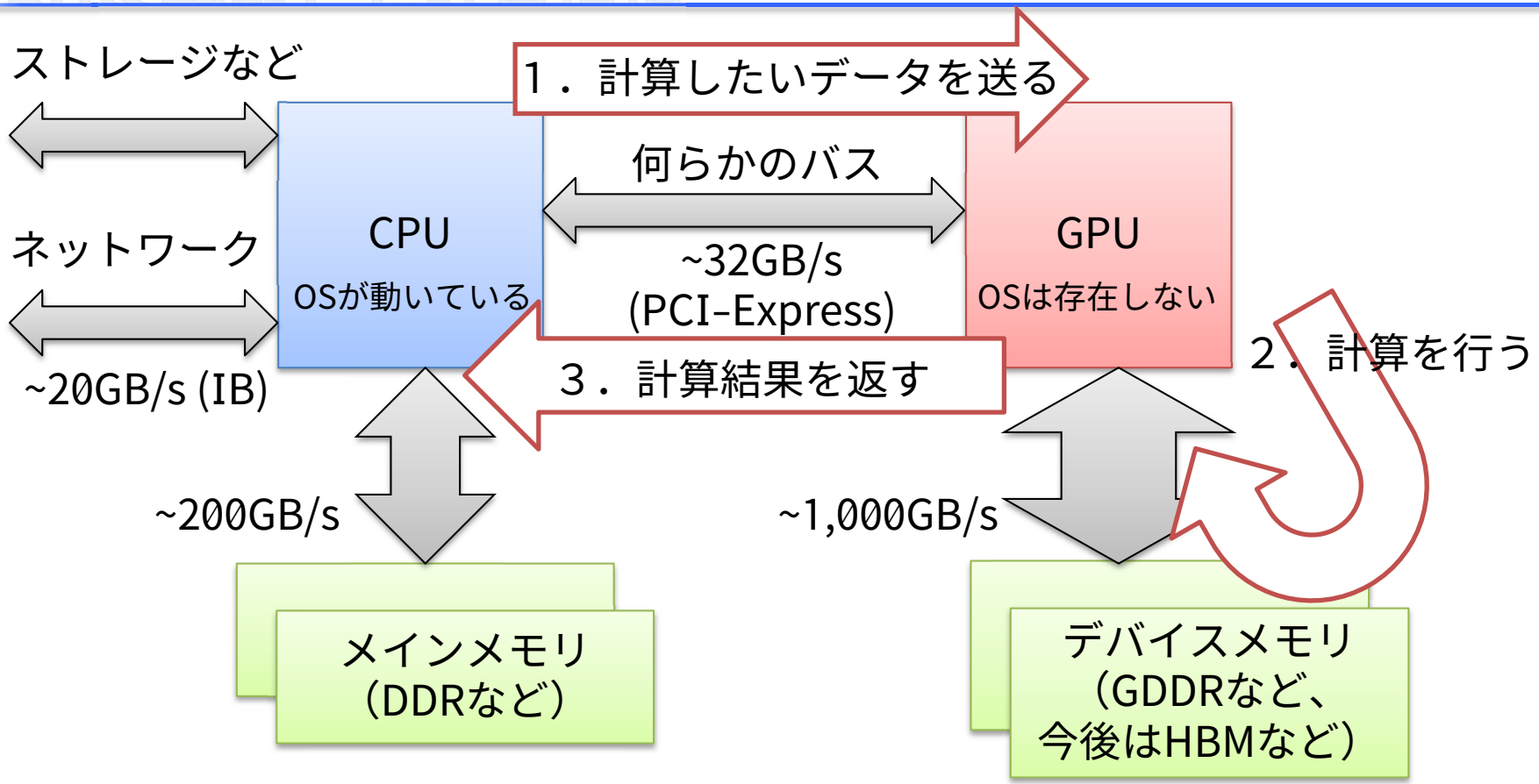
# NVIDIA社のGPUラインナップについて（2/3）

- アーキテクチャ（世代）と特徴・新機能
  - Tesla：最初のHPC向けGPU
  - Fermi：本講習会で用いるGPU
    - ECCメモリ対応、FMA演算、atomic演算
  - Kepler：現行のHPC向けGPU
    - コア群を構成するコア数の増加、動的な並列処理（GPUカーネルからGPUカーネルの起動）、Hyper-Q（複数CPUコアによるGPU共有）、シャッフル命令、読み込み専用データキャッシュ、Unifiedメモリ、PCI-Express 3.0
  - Maxwell：コンシューマ向けGPU
    - 電力あたり性能の向上、Teslaとしての製品は存在しない
  - Pascal：まだ販売されていないGPU、Reedbushに搭載
    - HBM2（高速メモリ）、NVLink（高速バス）

# NVIDIA社のGPUラインナップについて（3/3）

- 「現行GPUではできるが、講習会で使うGPUではできないこと」もあるが、最適化を行ううえで基本となる点は共通している
  →Reedbushでも活用できる
  - そもそもHPC向けのプログラミングには不要に近い機能も多い
- 世代毎に色々な制限等に違いがあるため、細かい最適化パラメタについては都度考える必要がある
  - 最大並列度、レジスタ数、共有メモリ容量、命令実行サイクル数、etc.

# GPUを搭載した計算環境




- GPUを使う為には１．２．３．を考える（実装する）必要がある
- デバイス内外のデータ転送速度差が大きいことから、対象とするプロセッサ内で計算が完結していることが望ましいことがわかる

# GPU入門・CUDA入門

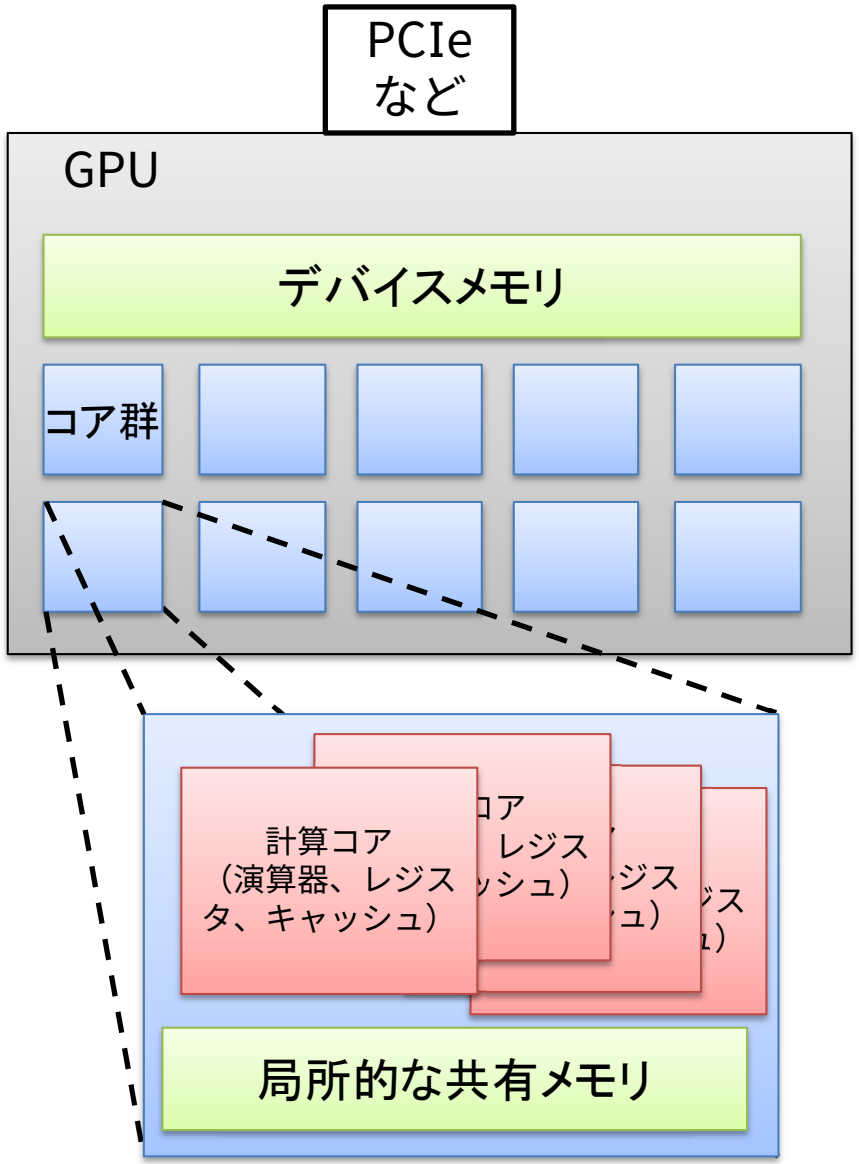
- GPUの構造とCUDAを用いたプログラミングの方法を学ぶ
- 最適化を行ううえで考えるべきこと（概要）を学ぶ

# ハードウェアスペックの確認

| | SPARC64 IXfx | Xeon E5-2670 (Sandy Bridge-EP) HA-PACS ホストCPU | Tesla M2090 (Fermi) HA-PACS GPU | Tesla K40 (Kepler) |
|---|---|---|---|---|
| コア数 | 16 | 8 (HT 16) | 512 (32*16) | 2880 (192*15) |
| クロック周波数 | 1.848 GHz | 2.60 GHz | 1.3 GHz | 745 MHz |
| 搭載メモリ種別 | DDR3 32GB | DDR3 最大384GB (HA-PACS 64GB/socket) | GDDR5 6GB | GDDR5 12GB |
| Peak FLOPS [GFLOPS] (SP/DP) | 236.5 | 332.8/166.4 | 1330/665 | 4295/1430 |
| Peak B/W [GB/s] | 85 | 51.4 | 178 (ECC off) | 288 |
| TDP [W] | 110 | 115 | 225 | 235 |

# GPUハードウェア：「超」概要

- ホスト(CPU)とデバイス(GPU)はPCI-Expressなどで接続されている
- GPU上にはいくつかのコア群とデバイスメモリが搭載されている
- コア群にはいくつかの計算コアと局所的な共有メモリが搭載されている
- 局所的な共有メモリはデバイスメモリと比べて高速だが小容量



# CUDA「超」入門：C言語版

- CUDA C (RuntimeAPI)　（より細かくCUDAを制御可能なDriverAPIもあるが、あまり使う必要は無いため割愛）
  - GPUが処理を行う単位は関数
    - CPUがGPUに関数を実行させるための記述が用意されている
      - gpufuncname<<<並列実行形状の指定>>>(引数);　並列実行形状については後述
  - 接頭辞を用いて関数の実行方法とメモリ配置を指定
    - 実行対象指定（組み合わせ可能）
      - __global__　CPUから呼び出し、GPU上で実行
      - __device__　GPUから呼び出し、GPU上で実行
      - __host__　CPUから呼び出し、CPU上で実行
    - 配置指定
      - __device__　GlobalMemory：GPU全体で共有するデバイスメモリ
      - __shared__　SharedMemory：局所的な高速共有メモリ
      - __constant__　ConstantMemory：読み出し専用に使う特殊なメモリ
      - （専用のクラスを用いて扱うTextureMemory）

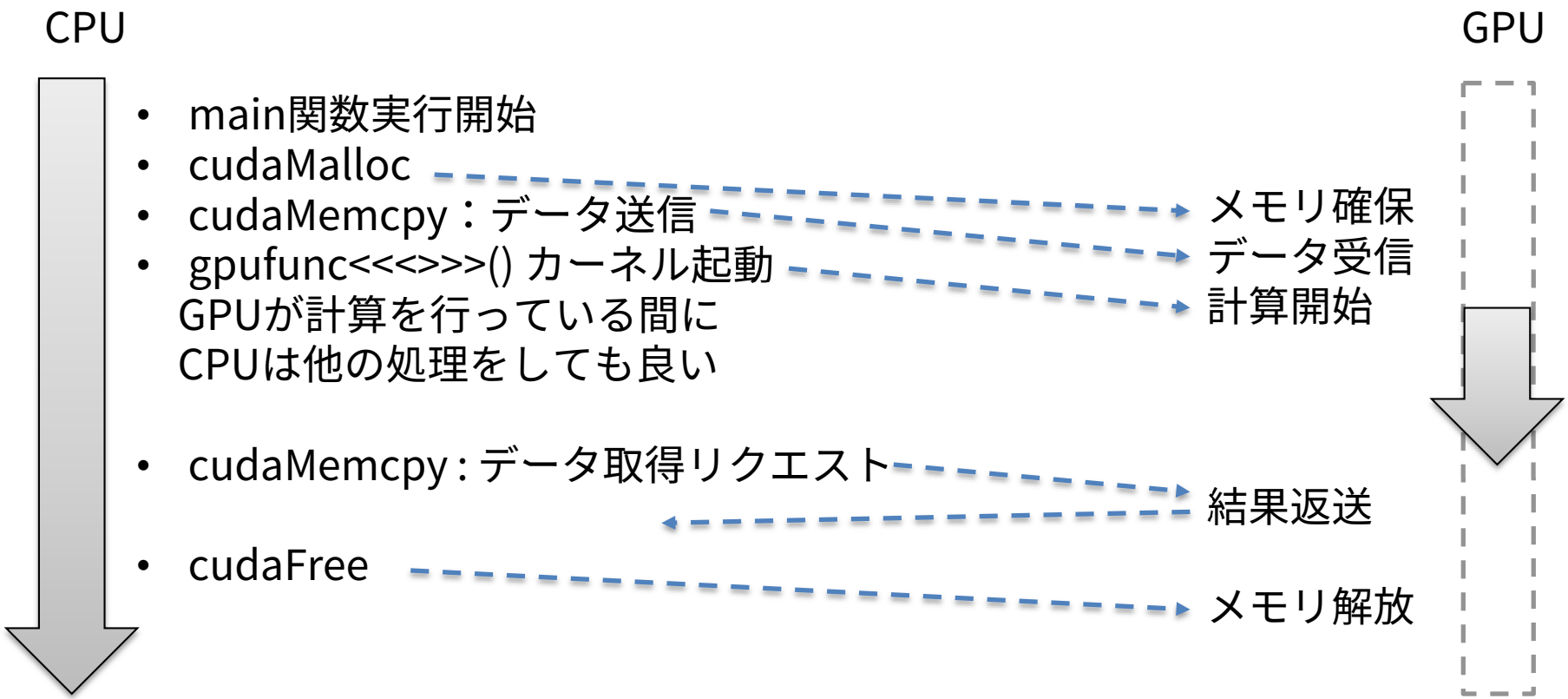
# CUDA「超」入門：C言語版（続き）

- 主なAPI関数
  - cudaMalloc
    - GPU上のメモリを確保する、GPU版malloc
  - cudaFree
    - cudaMallocで確保したメモリを解放する、GPU版free
  - cudaMemcpy
    - CPU-GPU間のデータ転送を行う
    - データ転送方向は引数で指定する

- Fortran版ではどうか？
  - 概念・考えるべきことは同様
  - 言語仕様の違いがあるため具体的な記述の仕方には違いがある
    - 配列等の宣言時にメモリ配置等を指定することで、専用のAPIを使わずにGPUを利用可能

# プログラムの動作イメージ

- CPUからの指示に従ってGPUが動作する



# CUDA C コード例1: シンプルな配列コピーの例

目標：どのような情報を書く必要があるのかを把握する

simple1.cu（CUDA Cプログラムの拡張子は.cu）

```
#define N 100000
__device__ float d_A[N], d_C[N];

__global__ void gpukernel()
{
  for(int i=0; i<N; i++){
    d_C[i] = d_A[i];
  }
}
```

GPU上のメモリ（配列）

単純な配列のコピー

GPU上で行われる処理（GPUカーネル）

```
float A[N], C[N];
int main(int argc, char **argv){
  cudaMemcpy(d_A, A, sizeof(float)*N, cudaMemcpyHostToDevice);

  gpukernel<<<1,1>>>();

  cudaMemcpy(C, d_C, sizeof(float)*N, cudaMemcpyDeviceToHost);
  return 0;
}
```

ホスト上のメモリ（配列）

cudaMemcpy CPU-GPU間のコピー

<<<>>> GPUカーネルの実行
※1,1なので逐次実行

CPU上で行われる処理

※各種API関数の細かい説明は後述（午後）

※配列A,Cの値は適当に初期化されていると仮定

# CUDA C コード例1a: 配列サイズを実行時に決める例

simple1a.cu

```
__global__ void gpukernel
(int N, float* C, float* A){
  for(int i=0; i<N; i++){
    C[i] = A[i];
  }
}
```

サイズや配列を引数として受け取る

GPU上で行われる処理（GPUカーネル）

```
int main(int argc, char **argv){
  int N = 100000;
  float *A, *C;
  float *d_A, *d_C;
  A = (float*)malloc(sizeof(float)*N);
  cudaMalloc((void**)&d_A, sizeof(float)*N);
  cudaMemcpy(d_A, A, sizeof(float)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(d_C, C, sizeof(float)*N, cudaMemcpyHostToDevice);
  gpukernel<<<1,1>>>(N, d_C, d_A);
  cudaMemcpy(C, d_C, sizeof(float)*N, cudaMemcpyDeviceToHost);
  cudaFree(d_A);
  return 0;
}
```

cudaMalloc GPU上のメモリを確保する
cudaFree GPU上のメモリを解放する

※C, d_Cのメモリ確保も必要だが、スペースの都合で省略

CPU上で行われる処理

※残りのメモリ解放はスペースの都合で省略

# CUDA C コード例1b: 並列計算を行う

simple1b.cu

```
__global__ void gpukernel
(int N, float* C, float* A){
  int tid = blockIdx.x*blockDim.x + threadIdx.x;
  int nt = gridDimd.x * blockDim.x;
  for(int i=tid; i<N; i+=nt){
    C[i] = A[i];
  }
}
```

CUDAにおける並列計算の基本
GPUカーネル内で自分のIDを取得し、
計算するべき範囲を特定する

GPUカーネル関数が
16個同時に起動する
と思えば良い

GPU上で行われる
処理
（GPUカーネル）

```
int main(int argc, char **argv){
  // simple1a.cu とほぼ同様

  gpukernel<<<4,4>>>(N, d_C, d_A);

  return 0;
}
```

並列実行形状を与える、ここでは4*4=16並列
での実行（のようなものだと思えば良い）

CPU上で行われる
処理

# CUDA C: プログラムのコンパイルと実行

- 通常のCプログラムと同様にコンパイル・実行が可能
  - nvccを使う
    - nvcc simple.cu
    - ./a.out
- nvccがGPUカーネルを分離し、CPU部とGPU部をそれぞれコンパイルし、単一の実行ファイルを生成する
  - CPU部またはGPU部のみをコンパイルしたり、中間表現ファイル（PTX≒アセンブラ）を出力して解析することも可能

# CUDA Fortranの場合

- 配列に属性を付加しておけば、確保や代入などの処理がGPUに対して行われる
  - device, global など
  - CUDA Cよりも簡単
  - コンパイル例

    pgf90 –Mcuda arraytest.cuf

```
program arraytest
  use cudafor     ←CUDA Fortranを使う為に必要
  use cudamod     ←GPUカーネルを含むモジュール
  integer, parameter :: N=10
  integer i
  real, allocatable, dimension(:) :: A, C
  real, device, allocatable, dimension(:) :: d_A, d_C
  allocate(A(N))
  allocate(C(N))
  A = …
  C = …
  allocate(d_A(N))
  allocate(d_C(N))          } GPUに対して行われる
  d_A = A
  d_C = C                   } CPU-GPU間でコピーされる
  call gpukernel<<<1,1>>>(N, d_C, d_A)
  C = d_C
  deallocate(d_A)
  deallocate(d_C)
  deallocate(A)
  deallocate(C)
end program arraytest
```

CPU側

```
module cudamod
  use cudafor
  contains
  attributes(global) subroutine gpukernel(N, C, A)
  integer, value :: N
  real, device :: C(N), A(N)
  C = A
  end subroutine gpukernel
end module cudamod
```

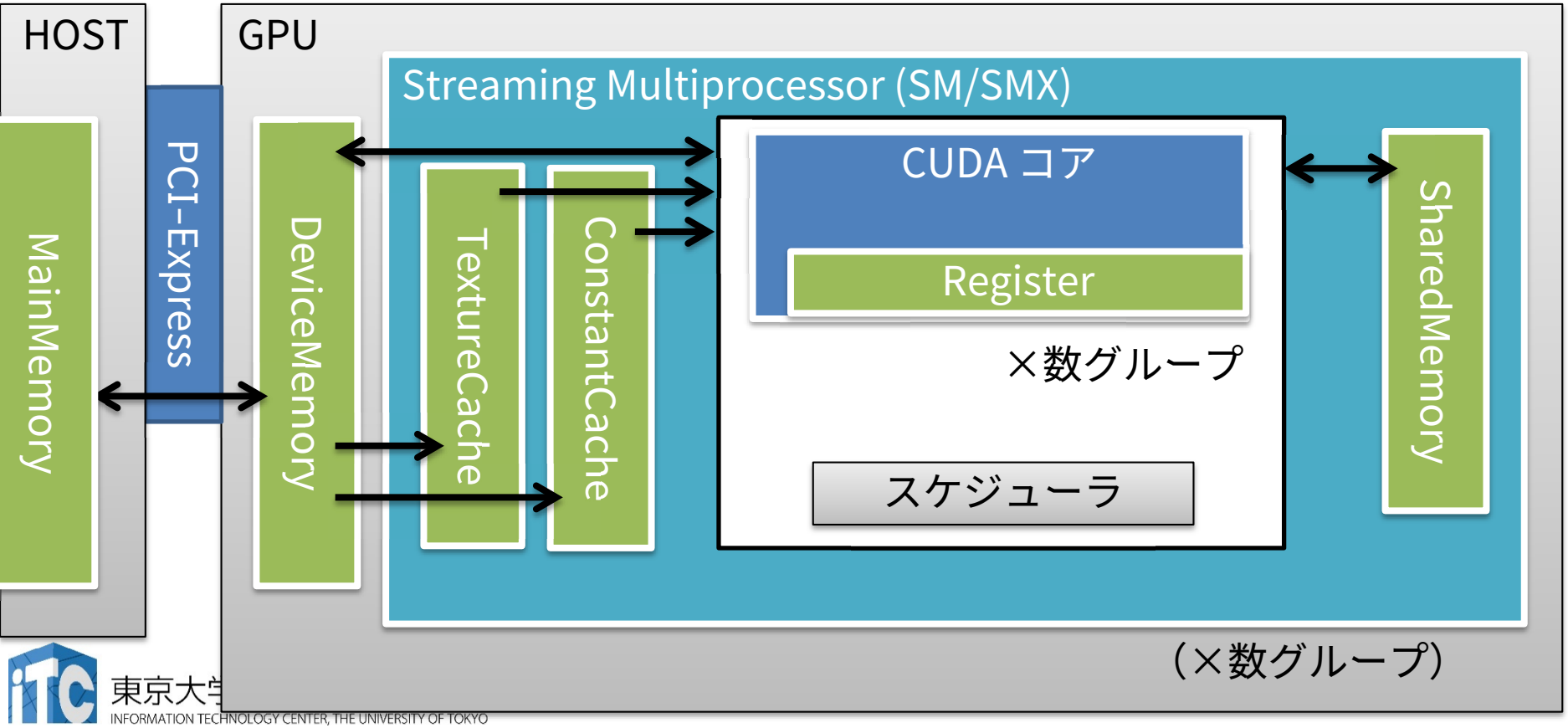
GPU側

GPUカーネル内で配列のコピーを行う例

# CUDAアーキテクチャ（物理的な構成）

※以下、SM/SMXをSMxと表記する

- 物理的な構成の概要
  - SM/SMXはGPUあたり1～30（GPUのグレードに依存）
  - CUDAコアはSM/SMXあたり8~192（GPU世代に依存）

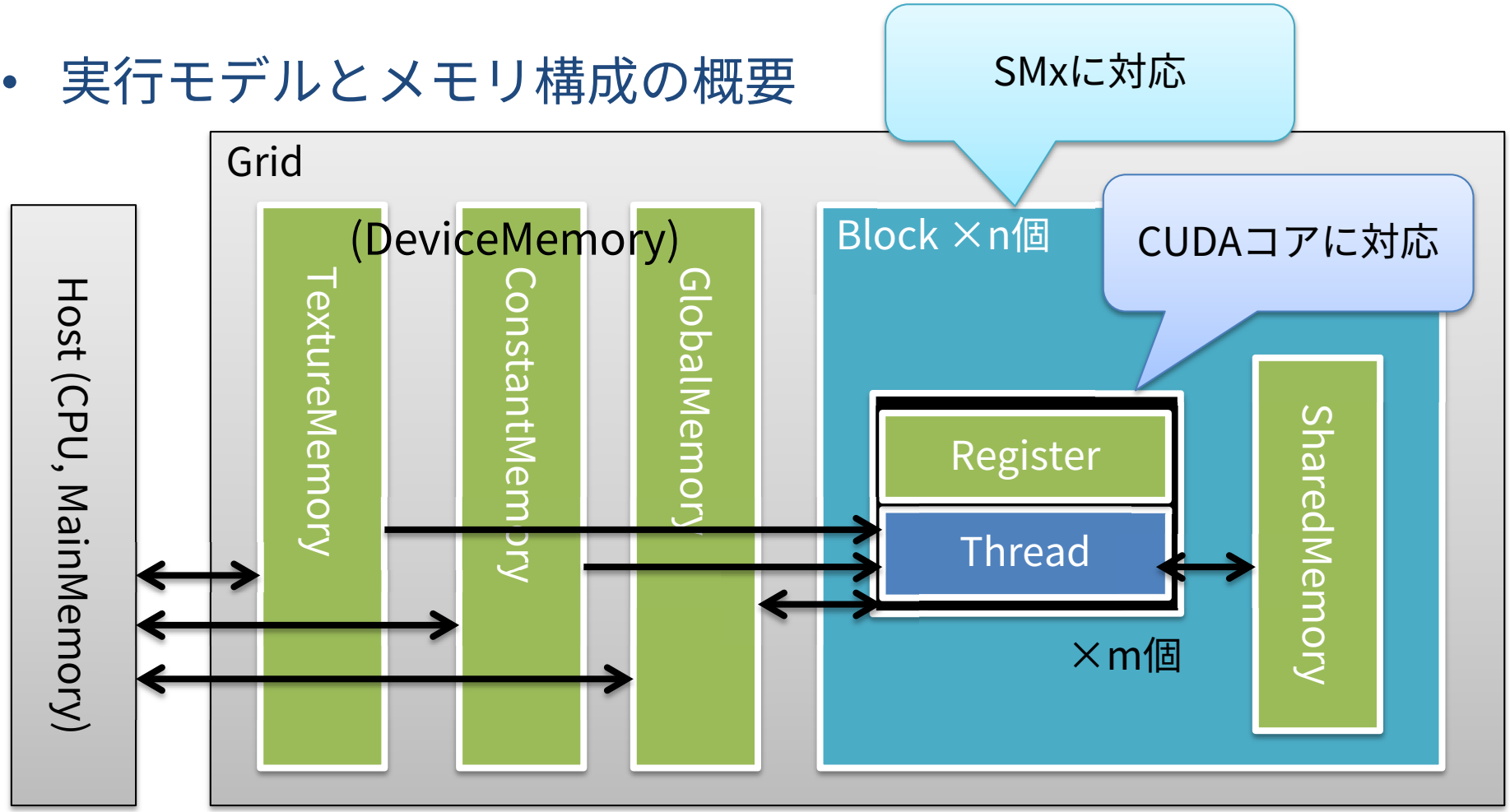


# ハードウェアの特徴

- 階層性のあるハードウェア構成
  - 演算器の構成
    - 階層性のある演算器配置（CUDAコア*m × SMx*n）
    - 幾つかの計算コアがグループを構成、同一グループ内のコアは同時に同じ演算のみ可能（SIMD的な構成）
      - CPUのコアのように独立して動作できず、分岐方向が違う場合にはマスク処理される
      - NVIDIAはSIMTと呼んでいる
  - メモリの構成
    - 階層性と局所性のあるメモリ配置
      - 全体的な共有メモリ ＋ 部分的な共有メモリ ＋ ローカルメモリ
      - GPU上に搭載された大容量でグローバルなメモリ：DeviceMemory
      - 局所的に共有される小容量高速共有メモリ：SharedMemory
      - コア毎に持つレジスタ

# CUDAアーキテクチャ（実行モデルとメモリ）

- 実行モデルとメモリ構成の概要




- CPUのプロセスやスレッド同様に、Block・Threadは物理的な数以上に生成可能、GPUカーネル起動時に<<<,>>>で指定するのはこの値
- 特にThreadは物理的な数を超えて作成した方が良い（後述）

# 詳細なメモリ構成

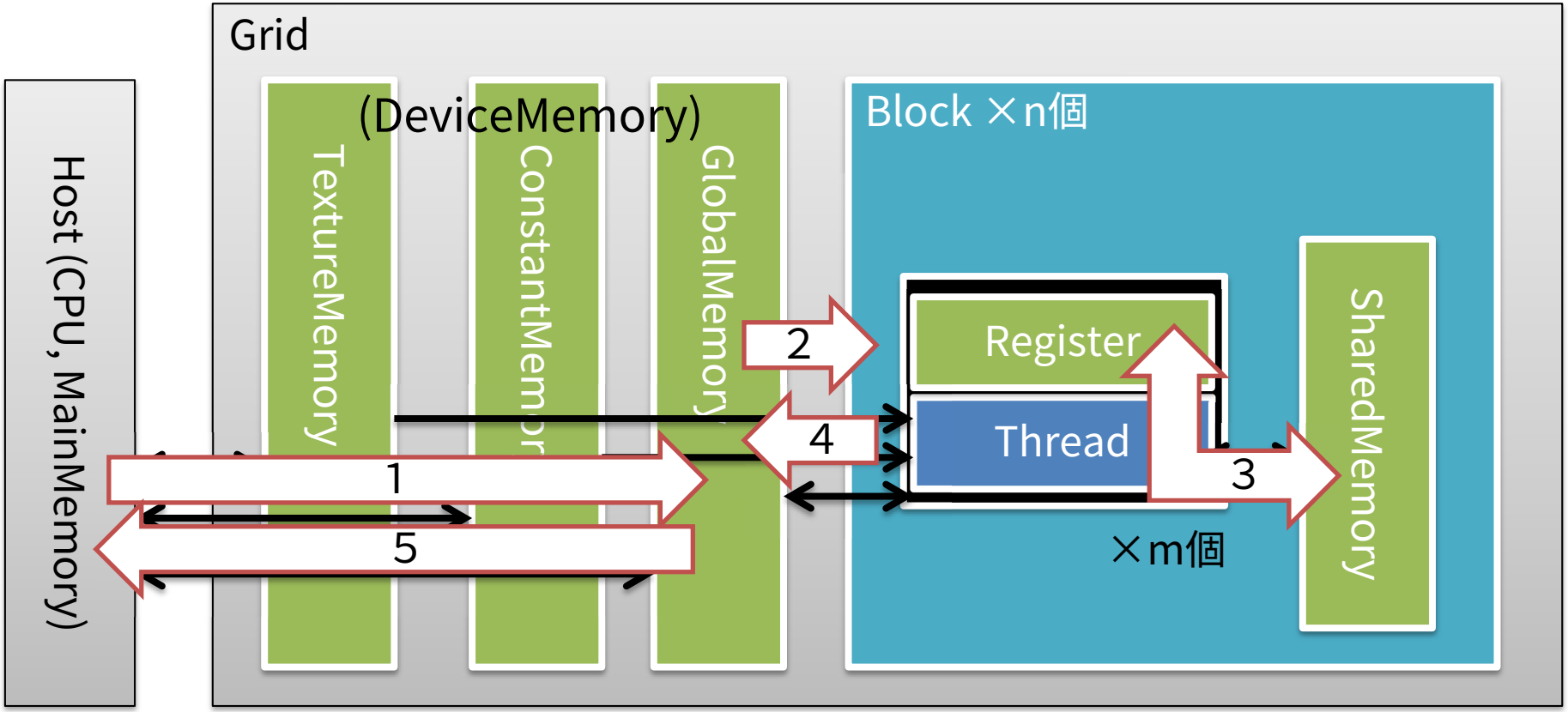
- 特徴の異なる複数種類のメモリ
  - 必ずしも全てのメモリを使う必要はない

| 名称 | Lifetime | 共有範囲 | 速度 | 容量 |
|---|---|---|---|---|
| GlobalMemory | プログラム | GPU全体 | 高速・高レイテンシ | ～GB |
| ConstantMemory | プログラム | GPU全体 | 高速・高レイテンシ<br>＋キャッシュ | 64KB |
| TextureMemory | プログラム | GPU全体 | 高速・高レイテンシ<br>＋キャッシュ | GlobalMemory<br>と共用 |
| SharedMemory | Grid | SMx単位 | 超高速・低レイテンシ | ~112KB/SMx * |
| Register | Grid | 非共有 | 超高速・低レイテンシ | ~64KB/SMx * |
| LocalMemory ** | Grid | 非共有 | 高速・高レイテンシ | - |

* GPUの世代により異なる
** 実体はGlobalMemory、レジスタを使いすぎると
LocalMemoryに配置されてしまう

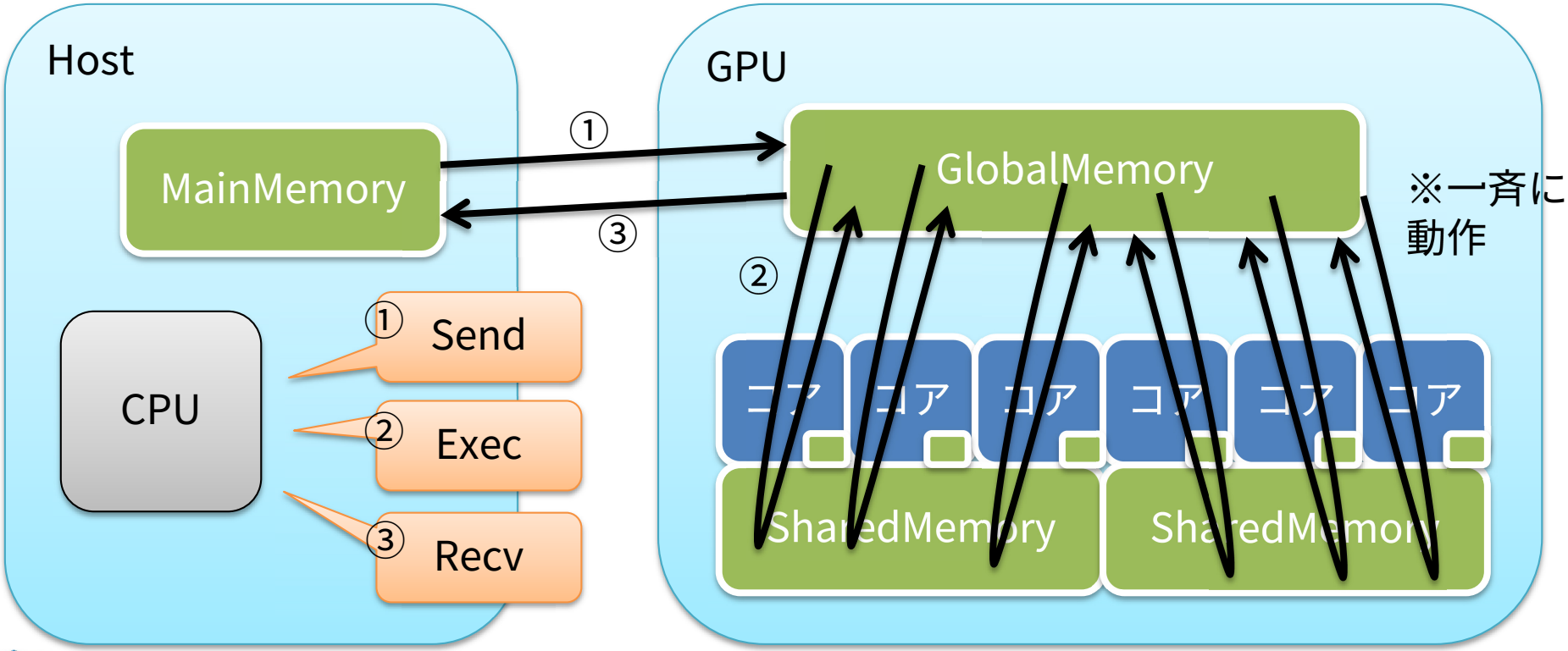
# CUDAアーキテクチャ（データの流れ）

- 計算時の データの流れ



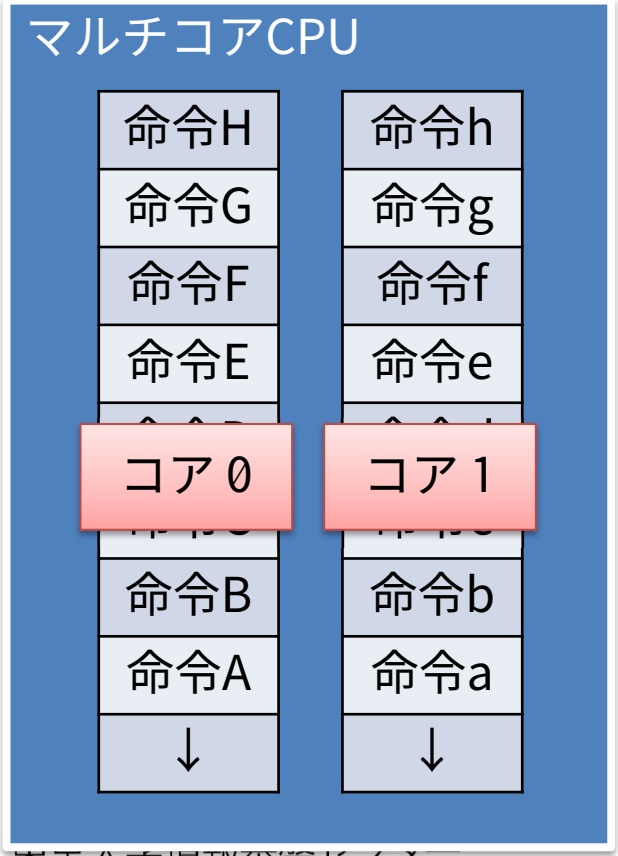
# CUDAアーキテクチャ（GPUの制御）

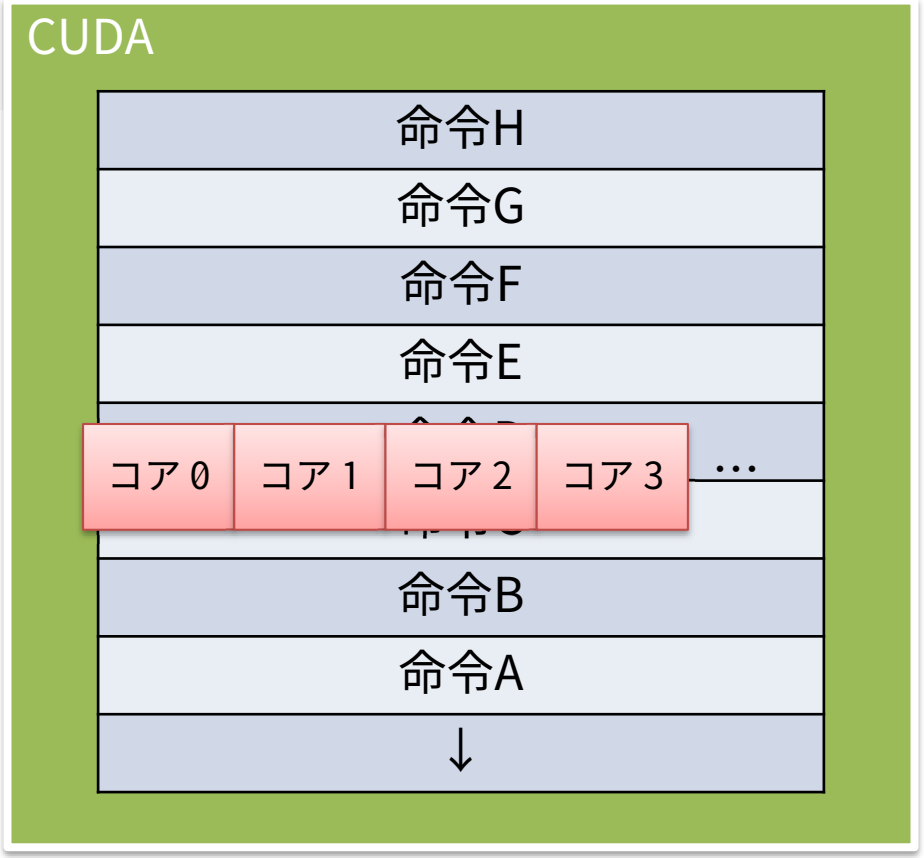
- もう少し詳しい実行モデル解説
  - CPUによるGPU制御、GPU上のコアの一斉動作



# CUDAアーキテクチャ（命令の流れ）

- もう少し詳しい実行モデルのイメージ
  - 各コアが流れてくる命令を処理していくようなイメージで考える
  - GPU上のコア群は同時に同じ命令を実行している（全体で、ではない）



# さらに詳しい実行モデルの解説

- 実際のスケジューリングは32スレッド単位（＝WARP単位）で実行される
- 異なるデータに対して同時に同じ演算を行う
  - 実行時に取得できるスレッドIDを用いて各自の計算対象（配列インデックス）を算出すれば良い
- WARP内のスレッド毎に分岐方向が異なるプログラムを実行する場合は、分岐方向の異なるスレッドは待たされる
  - divergent warp、重要な性能低下要因
  - スレッドIDが連続する32個のスレッド毎に分岐方向が揃うようなプログラムを作成すれば、divergent warpによるペナルティが発生しない

# CUDA最適化プログラミングの考え方

- どのようなプログラムに対して高性能が得られるか
  - 大量のスレッドを生成する
    - 理想的なBlockあたりスレッド数は64~256程度
    - GPUの世代やプログラムの複雑度などにも影響を受ける
  - GlobalMemoryのコアレスアクセスを行う
    - メモリアクセスをまとめる機能がある
  - SharedMemoryのバンクコンフリクトを回避する
    - SharedMemoryを利用する際に同じメモリバンクにアクセスすると性能が低下する
  - ループアンローリング
    - 分岐回数を減らす、GPUは分岐処理に弱いので重要
- 以下、各手法の概要について説明する
  - 実例や対策は後述（CUDA・OpenACCの最適化の中で扱う）
  - 最適化の際には各手法が衝突することもあるので注意が必要

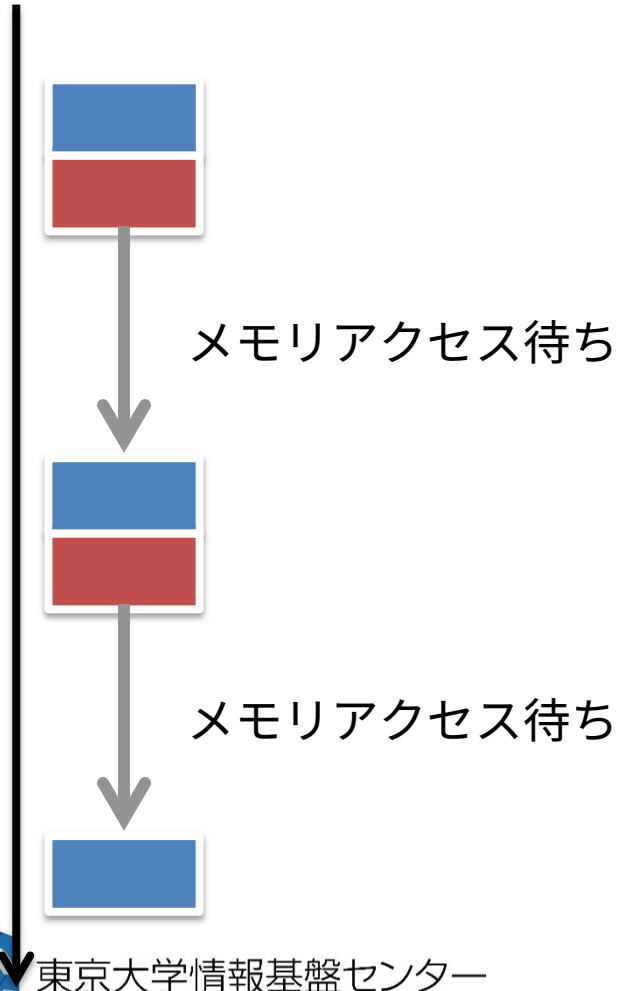
# 大量のスレッドを生成する

- スレッドのコンテキスト切り替えがとても速いため、メモリアクセスを待つよりコンテキストを切り替えて別のスレッドを処理した方が速い
- 逆に言えば大量のスレッドでGlobalMemoryに対するメモリアクセスのレイテンシを隠蔽しないと高い性能が得られない
- ただし、レジスタや共有メモリの使用量が多すぎると多数のスレッドを実行できない
  - 同時に実行できるスレッドやブロックの数は色々な資源の使用量によって決まる

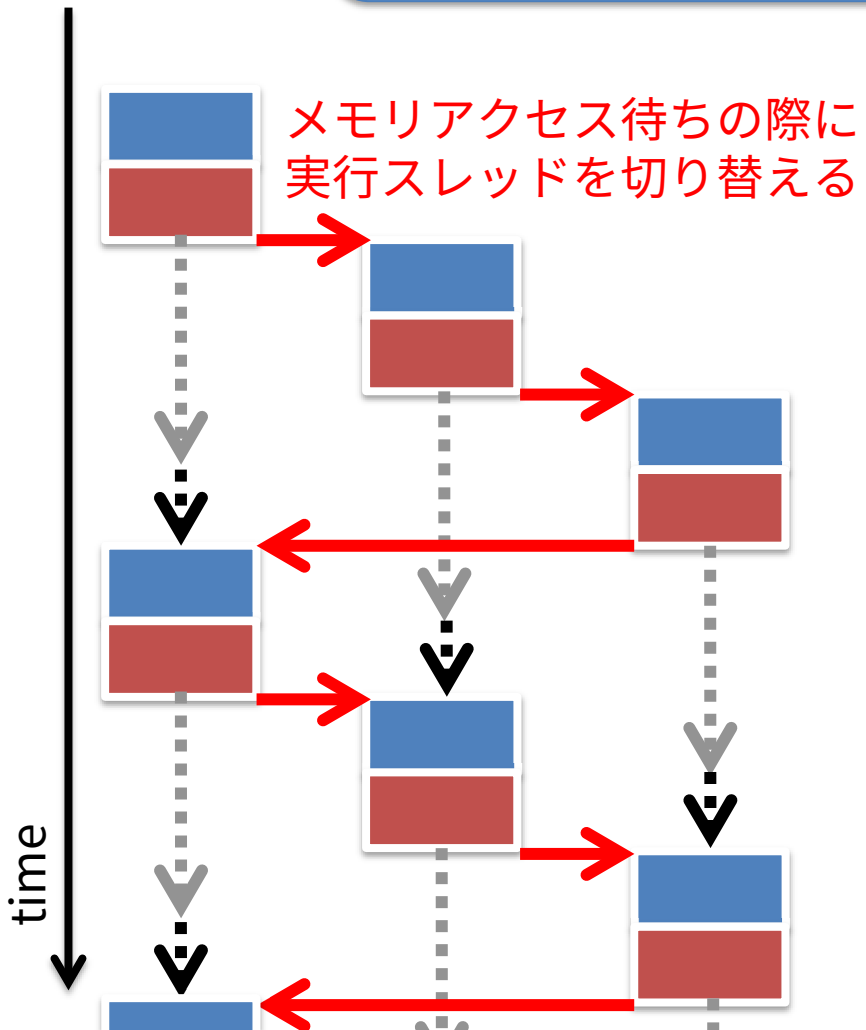
並列度の高いシンプルなGPUカーネルが望ましい

# 高速なスレッド切り替えのイメージ


- CPU

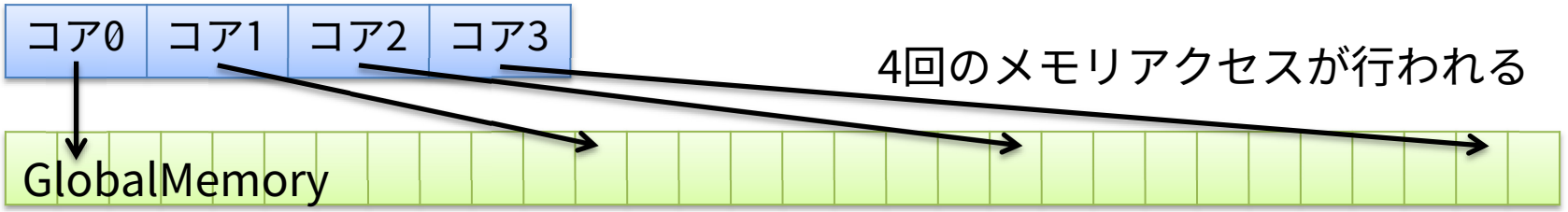



- CUDA

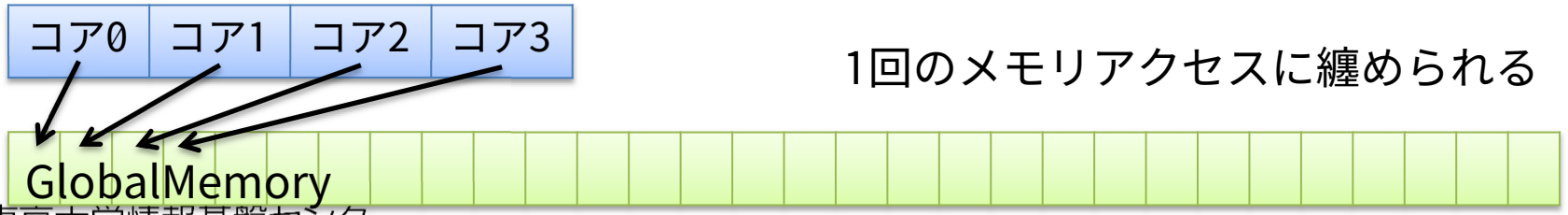


# GlobalMemoryのコアレスアクセス

- 同一SMx内の複数CUDAコアによるメモリアクセスが近い場合にまとめてアクセスできる
  - 詳細な条件はGPUの世代によって異なる
    - 最新世代ほど条件が緩い
- アクセスがバラバラな（遠い）場合

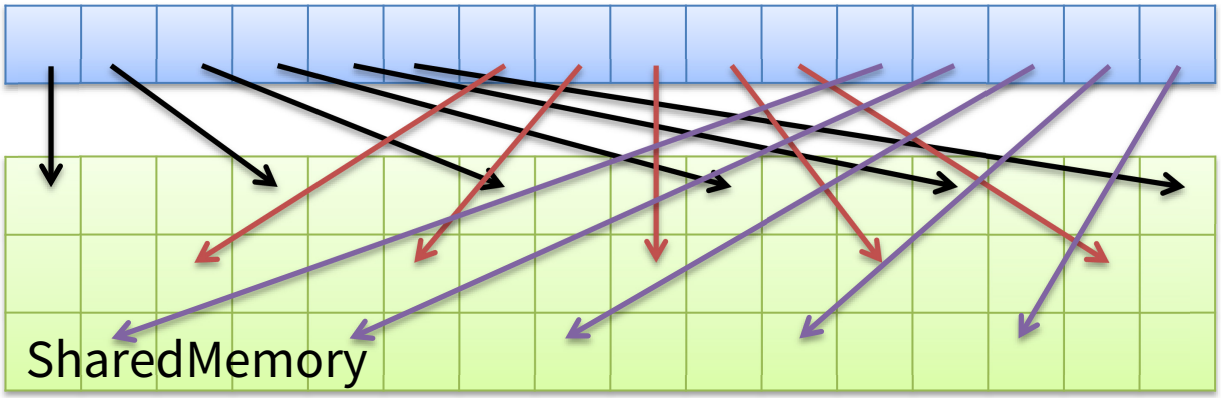



- アクセスが揃っている（近い）場合

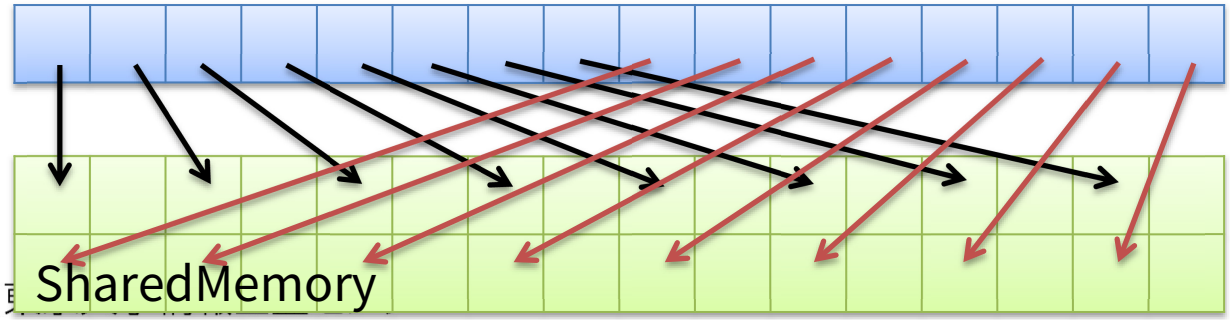


# SharedMemoryのバンクコンフリクト回避

- 高速共有メモリは16個or32個ずつのバンクにより構成
  - 同一バンクへのアクセスが集中すると性能低下
- 均等なアクセス＝性能低下しない




- アクセスが集中＝性能低下する




2-way
バンクコンフリクトの例

# CUDA最適化入門と演習

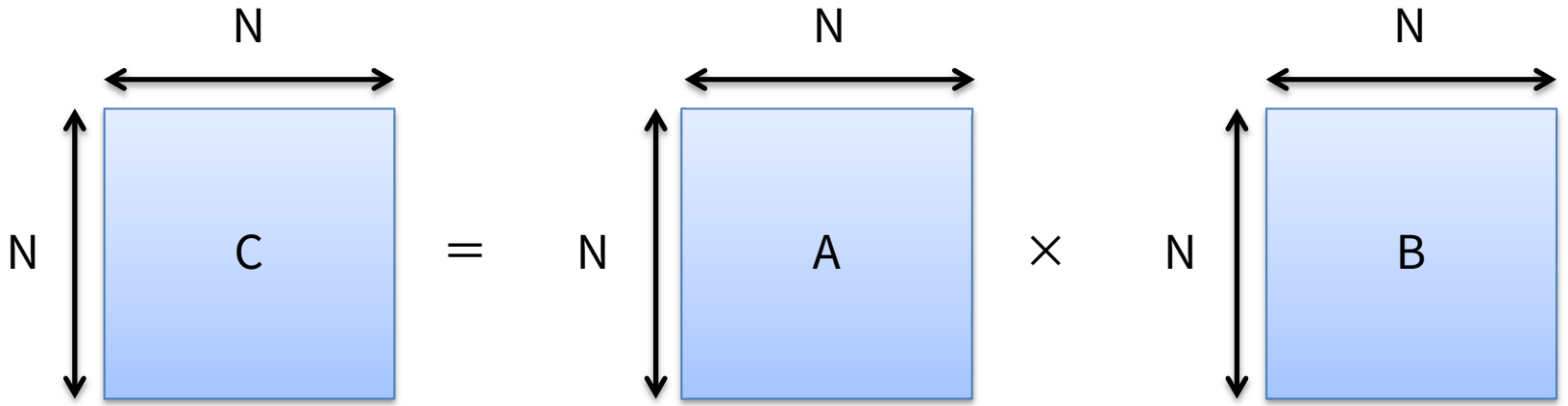
簡単な数値計算プログラム（行列積、行列ベクトル積）の最適化を題材に、CUDAプログラムの最適化の基本について学ぶ

# 問題設定

- 問題設定
  - 行列積　C=A×B
  - データ型：単精度浮動小数点型
    - CUDA C : float型
    - CUDA Fortran : real型

※倍精度浮動小数点型でも
特に考え方は変わりません

  - 話を簡単にするため、N×Nサイズの正方行列を対象として、主に行列Aの参照を並列高速化することについて考える



# 行列積を実装する前に……

- まずは一次元配列を処理する簡単なプログラムを作成し、CUDAプログラムの作成方法を理解する
- 用意するもの：CUDAプログラム arraytest.cu/arraytest.cuf
- プログラム内で行うこと、書かねばならないこと
  - CPU側（ホスト側処理）
    - 配列を確保する：CPU用、GPU用
    - CPUからGPUへデータを送る
    - GPUカーネルを起動する
    - GPUからCPUへデータを書き戻す
  - GPU側（カーネル関数）
    - 引数として問題サイズと計算対象配列を受け取る
    - 配列を処理（コピーor加算）する

# 配列をコピーするテスト：arraytest.cu（CUDA C）

- 配列Aと配列Cを用意し、配列Cに配列Aの内容をコピーする

main関数内

```
const int N = 1000;
float *A, *C;
A = (float*)malloc(sizeof(float)*N);
C = (float*)malloc(sizeof(float)*N);
for(i=0; i<N; i++){A[i]=(float)i; C[i] = 0.0f;}
float *d_A, *d_C;
cudaMalloc((void**) &d_A, sizeof(float)*N);
cudaMalloc((void**) &d_C, sizeof(float)*N);
cudaMemcpy(d_A, A, sizeof(float)*N, cudaMemcpyHostToDevice);
cudaMemcpy(d_C, C, sizeof(float)*N, cudaMemcpyHostToDevice);
dim3 blocks(1, 1, 1);
dim3 threads(1, 1, 1);
gpukernel<<< blocks, threads >>>(N, d_C, d_A);
cudaMemcpy(C, d_C, sizeof(float)*N, cudaMemcpyDeviceToHost);
cudaFree(d_A);
cudaFree(d_C);
free(A); free(C);
```

CPUメモリ確保、初期化

GPUメモリ確保

CPUからGPUへのデータ転送

並列度の指定（１：逐次）

GPUカーネル実行開始

演算終了待ち、結果取得

GPUメモリ破棄

CPUメモリ破棄

# API関数解説：配列の確保と破棄、データ転送

- 配列の確保：cudaMalloc
  - 第一引数：確保対象
  - 第二引数：サイズ（バイト数）
- 配列の破棄：cudaFree
  - 引数：破棄対象
  - プログラム終了時に破棄されていなくても特にペナルティは無い
- CPU-GPU間のデータ転送：cudaMemcpy
  - 第一引数：転送先
  - 第二引数：転送元
  - 第三引数：転送サイズ（バイト数）
  - 第四引数：転送方向
    - cudaMemcpyDeviceToHost、cudaMemcpyHostToDevice
  - 第五引数：ストリーム（省略可能、今回は扱わない）

# GPU側の記述とコンパイル

GPUカーネル関数

```
__global__ void gpukernel
(int N, float *C, float *A)
{
  int i;
  for(i=0; i<N; i++){
    C[i] = A[i];
  }
}
```

コンパイル

```
$ nvcc arraytest.cu
```

配列と変数の扱い

- 引数として与えられた配列はGlobalMemory上に配置される
- 引数として与えられた変数はレジスタ上に配置される
- カーネル内で宣言された変数や配列はレジスタ上に配置される
  - 容量が大きすぎるとLocalMemory扱いにされる（GlobalMemoryに配置される）ので注意

# CUDA Fortranの場合（CPU側）

```
program arraytest
  use cudafor
  use cudamod
  integer, parameter :: N=10
  integer i
  real, allocatable, dimension(:) :: A, C
  real, device, allocatable, dimension(:) :: d_A, d_C
  allocate(A(N))
  allocate(C(N))
  A = …              ! 適当に配列の
  C = …              ! 初期化を行う
  allocate(d_A(N)) ! GPU上での
  allocate(d_C(N)) ! メモリ確保
  d_A = A          ! CPUからGPUへの
  d_C = C          ! データ転送
  call gpukernel<<<1,1>>>(N, d_C, d_A)
  C = d_C          ! 結果の取得
  deallocate(d_A)
  deallocate(d_C)
  deallocate(A)
  deallocate(C)
end program arraytest
```

- CUDA Fortanを使うにはuse cudaforが必要
- GPU上に置かれるデータにはdevice属性を付加
- device属性を持つ配列に対してallocateや配列コピーをするとGPU用の処理が行われる
- GPUカーネルの起動はCUDA Cと同様に <<< >>> を使って行う

# GPU側の記述とコンパイル

GPUカーネル関数

```
module cudamod
  use cudafor
  contains
  attributes(global) subroutine gpukernel(N, C, A)
  integer, value :: N
  real, device :: C(N), A(N)
  C = A
  end subroutine gpukernel
end module cudamod
```

コンパイル

```
$ pgf90 –Mcuda  arraytest.cuf
```

さらに-Minfoを追加指定すると様々な情報が表示される（ことがある）

注意

この資料は基本的にCUDA Cを使う前提で書いているが、CUDA FortranでもGPUカーネルの中身はほとんど同じである。適宜読み替えること。

# 演習

- arraytest.cu/arraytest.cufを完成させ、計算後の配列Cを表示して正しく動作していることを確認する
  - 問題サイズや初期データは自由に決めて良い
  - GlobalMemory上のデータは次にGPUカーネルを実行するときも引き継がれる
    - GPUカーネルを複数回実行したり、送受信する配列を増やしたりして動作を確認してみよう

# Tips：エラー処理

- 多くのAPI関数は返値を見れば関数の成否が確認できる
  - 成功時はcudaSuccessという値が得られる
- 問題があった場合はその内容を専用の関数で取得できる
  - cudaGetLastError関数とcudaGetErrorStringを使う

```
cudaError ret = cudaMalloc(…);
if(ret!=cudaSuccess){
  cudaError _err = cudaGetLastError();
  if(cudaSuccess!=_err){
    printf("%s￥n” , cudaGetErrorString(_err));
  }
}
```

# 行列積プログラムの実装

- 行列積GPUカーネルを実装し性能を確認する
- 段階的に最適化を適用して性能の差を確認する

# CUDA C：CPU側

- CPU側は共通のプログラムを使用
  - 並列度は必要に応じ変更する

```
cudaMalloc((void**) &d_A, sizeof(float)*N*N);
cudaMalloc((void**) &d_B, sizeof(float)*N*N);
cudaMalloc((void**) &d_C, sizeof(float)*N*N);
cudaMemcpy(d_A, A, sizeof(float)*N*N, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, sizeof(float)*N*N, cudaMemcpyHostToDevice);
cudaMemcpy(d_C, C, sizeof(float)*N*N, cudaMemcpyHostToDevice);
dim3 threads(TX, TY, 1);
dim3 grid(GX, GY, 1);
gpukernel<<< grid, threads >>>(N, d_C, d_A, d_B);
cudaMemcpy(C, d_C, sizeof(float)*N*N, cudaMemcpyDeviceToHost);
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

GPUメモリ確保

データ転送

並列度の指定（必要に応じて変更する）

演算開始

演算終了待ちと
結果の取得

# 行列の確保：配列の扱いに関する注意

- CUDA Cでは多次元配列は扱いにくい
  - cudaMalloc, cudaMemcpyは一次元配列のみを対象としている
- 解決策はいくつかあるが……
  - 多次元配列を扱うための関数を使う
    - 専用の関数・手順が必要、めんどくさい
  - __device__接頭辞をつけた固定長の配列を確保する
    - 簡単だが扱いにくい（問題サイズの変更などがしにくい）
  - 一次元配列に置き換えて考える
    - プログラムが若干複雑になるが使い方自体は簡単で汎用的
- 本資料ではCPU上でもGPU上でも全て一次元配列を用いる
  - 問題サイズを可変にするため、ポインタを宣言しておいて動的に確保する

```
float *d_A, *d_B, *d_C; // GPU
float *A, *B, *C; // CPU
```

- CUDA Fortranでも一次元配列を使う→同様のGPUカーネル

# CUDA C：最適化前のGPUカーネル関数（逐次実行）

- 特に最適化を行っていない逐次実行カーネル

```
__global__ void gpukernel
( int N, float *C, float *A, float *B ){
  int i, j, k;
  for(k=0; k<N; k++){
    for(j=0; j<N; j++){
      for(i=0; i<N; i++){
        C[k*N+j] += A[k*N+i] * B[i*N+j];
      }
    }
  }
}
```

  - 単純な３重ループ
  - 遅い
    - 並列演算していない
      - 計算コア単体の性能は同世代のCPU未満
    - GlobalMemoryアクセスばかりしている

# GlobalMemoryアクセスの削減

- GlobalMemory上の配列を毎回書き換えるのをやめるだけでもそれなりに影響がある

```
__global__ void gpukernel
( int N, float *C, float *A, float *B ){
  int i, j, k;
  float tmp;
  for(k=0; k<N; k++){
    for(j=0; j<N; j++){
      tmp = 0.0f;
      for(i=0; i<N; i++){
        tmp += A[k*N+i] * B[i*N+j];
      }
      C[k*N+j] = tmp;
    }
  }
}
```

演習

- 実装し、実行して比較してみる
  →実行時間はどのように測定するべきか？

mm2

# 実行時間の測定方法

- 汎用のタイマー関数、OpenMPやMPIの提供する測定関数
  - gettimeofday, omp_get_wtime, MPI_Wtime
  - もちろん、これらを使っても良い、非同期関数には注意（次頁）
- CUDAに用意されているもの：cudaEvent、プロファイラ
  - cudaEvent

```
float elapsedTime;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
ここに測定対象の処理を入れる
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

# 実行時間の測定に関する注意点

- CUDAの提供する関数（API）には非同期な関数が多い
- （CUDAにおける）非同期な関数とは？
  - GPUに対して処理内容を伝えた時点でCPUに制御が返ってくる関数
  - 「CPUからは処理が終わっているかのように見えるが、GPUは動作している」状態がありえる
  - 単純にAPI関数の実行時間を測定すると、正しい実行時間にならない
    - 大きな行列に対する行列積を逐次実行するとわかりやすい
- 正しく測定する方法
  - GPUが処理を終えるのを待つ関数を実行し、終了を保証する
    - cudaThreadSynchronize(引数無し);
  - プロファイラを使う（次頁）

# コマンドラインプロファイラの使い方１

- 環境変数 COMPUTE_PROFILEに１をセットしてCUDAプログラムを実行すれば実行情報を取得できる

```
COMPUTE_PROFILE=1 ./a.out
もしくは
export COMPUTE_PROFILE=1 してから
./a.out
```

※ジョブスクリプトに書き足す

  - cuda_profile_0.log のようなファイルが作られる
  - gputimeの項を見ると時間がわかる

```
method=[ memcpyHtoD ] gputime=[ 0.736 ] cputime=[ 8.020 ]
method=[ _Z9gpukerneliPfS_ ] gputime=[ 3.968 ] cputime=[ 11.833 ] occupancy=[ 0.167 ]
method=[ memcpyDtoH ] gputime=[ 1.632 ] cputime=[ 15.579 ]
```

  - さらに色々な情報を得たい場合には設定を追加する
    - COMPUTE_PROFILE_CONFIGなどの設定を利用する

……が、いずれ廃止される予定であり、
現在はnvprofの使用が推奨されている模様

# コマンドラインプロファイラの使い方２

- 実行ファイルを与えるだけで良い：nvprof ./a.out

```
==203057== NVPROF is profiling process 203057, command: ./mm.out
==203057== Profiling application: ./mm.out
==203057== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 96.56%  114.94us         1  114.94us  114.94us  114.94us  gpukernel(int, float*, float*, float*)
  2.12%  2.5280us         3     842ns     768ns     992ns  [CUDA memcpy HtoD]
  1.32%  1.5680us         1  1.5680us  1.5680us  1.5680us  [CUDA memcpy DtoH]

==203057== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.76%  826.60ms         3  275.53ms  4.7720us  826.59ms  cudaMalloc
  0.16%  1.3396ms       332  4.0340us     120ns  155.75us  cuDeviceGetAttribute
  0.02%  164.30us         4  41.074us  6.0780us  122.88us  cudaMemcpy
  0.02%  152.15us         4  38.036us  36.221us  40.665us  cuDeviceTotalMem
  0.02%  147.76us         3  49.253us  5.2430us  126.78us  cudaFree
  0.02%  146.48us         4  36.620us  30.233us  54.212us  cuDeviceGetName
  0.00%  33.160us         1  33.160us  33.160us  33.160us  cudaLaunch
  0.00%  5.2890us         2  2.6440us     180ns  5.1090us  cuDeviceGetCount
  0.00%  4.2620us         4  1.0650us     148ns  3.5240us  cudaSetupArgument
  0.00%  2.0040us         1  2.0040us  2.0040us  2.0040us  cudaConfigureCall
  0.00%  1.8070us         8     225ns     123ns     785ns  cuDeviceGet
```



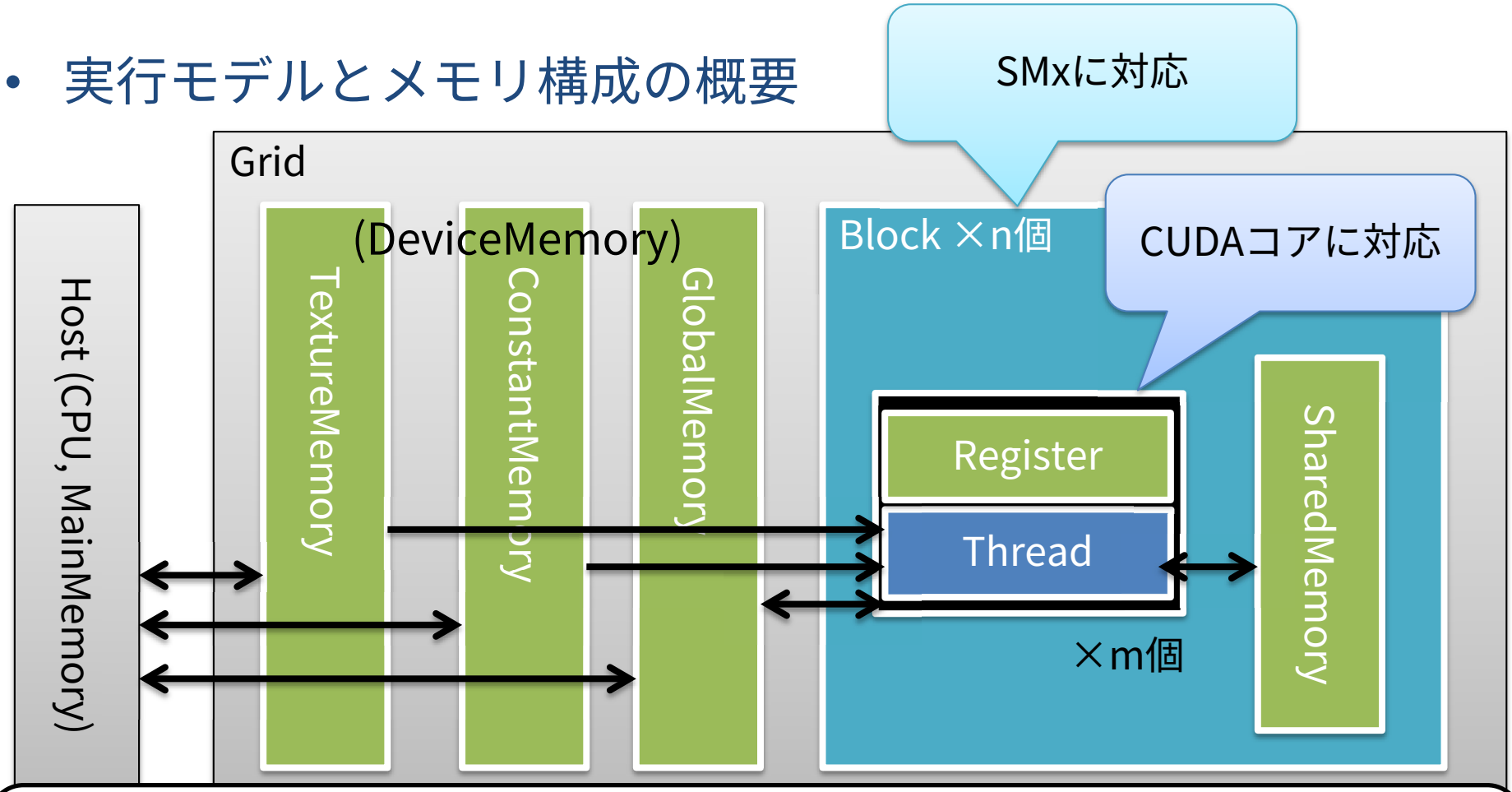
各演習において測定してみよう

# 並列度の指定について

- GPUカーネルを起動する際にブロック（Block）とスレッド（Thread）の数を指定する
  - <<<グリッドあたりブロックサイズ, ブロックあたりスレッドサイズ>>>
  - 各値の乗算分のスレッドがGPU上で動作する
- それぞれ三次元の値を指定可能
    - dim3 block; block.x = 32; block.y = 4; block.z = 2;
    - dim3 thread(32,16,2); のように宣言時に指定しても良い
    - <<<32,2>>>のようにスカラー値を直接与えても良い：(32,1,1), (2,1,1)扱い
  - 最大並列度
    - グリッドあたりブロックサイズ
      - 公式ドキュメントにおけるMaximum ～ of a grid of thread blocks
      - x次元：Fermiでは65535, Kepler以降では$2^{31}$-1、y,z次元：65535
    - ブロックあたりスレッドサイズ
      - 公式ドキュメントにおけるMaximum ～ of a block
      - x,y次元：1024、z次元：64

# CUDAアーキテクチャ（実行モデルとメモリ）

- 実行モデルとメモリ構成の概要




- GPUカーネルは１つのグリッド（Grid）としてGPU上で実行される
- スレッドの集合がブロック（Block）、ブロックの集合がグリッド（Grid）
- ブロックとスレッドは物理的な数以上に生成可能（時分割実行される）
- 生成する数量はGPUカーネル起動時に<<<,>>>で指定する

# 並列度指定の際に考えること（性能への影響）

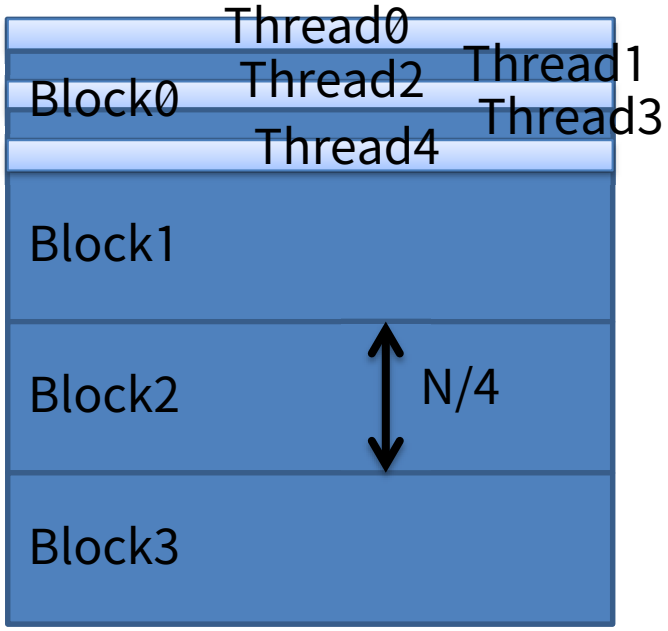
- メモリアクセスパターンとの対応づけ
  - 同一ブロック内の近いIDを持つスレッド群はコアレスなメモリアクセスが行える
  - 同一ブロック内のスレッド群は高速共有メモリ（SharedMemory）を共有する
    - データの使い回しを考える必要がある
- いくつくらいの値を与えるのが妥当なのか？
  - 不足するとGPUに仕事が行き渡らない、多すぎる方がマシ
  - ブロックあたりスレッドサイズ
    - （細かい話を省くと）128~256程度、32の倍数で試すのが良い
  - グリッドあたりブロックサイズ
    - GPUに搭載されているSMxの数に応じて指定
    - Tesla M2090は16ユニット搭載のため、16またはその倍数が妥当？（実際にはそれほどこだわらなくても良い）

# WARPサイズに合わせた最適化

- GPU内部での命令割り当ては32スレッド単位（WARP単位）で行われている
- 分岐処理の単位もWARP
- （Fermiでは使えないが）WARP内でデータをやりとりする命令（シャッフル命令）も存在する
- 常に32スレッド単位での動作を意識しておくと良い

# 単純な行単位での並列化

```
__global__ void gpukernel
(int N, float *C, float *A, float *B){
  int i, j, k;
  float tmp;
  k = blockIdx.x*blockDim.x + threadIdx.x;
  for(j=0; j<N; j++){
    tmp = 0.0f;
    for(i=0; i<N; i++){
      tmp += A[k*N+i] * B[i*N+j];
    }
    C[k*N+j] = tmp;
  }
}
```

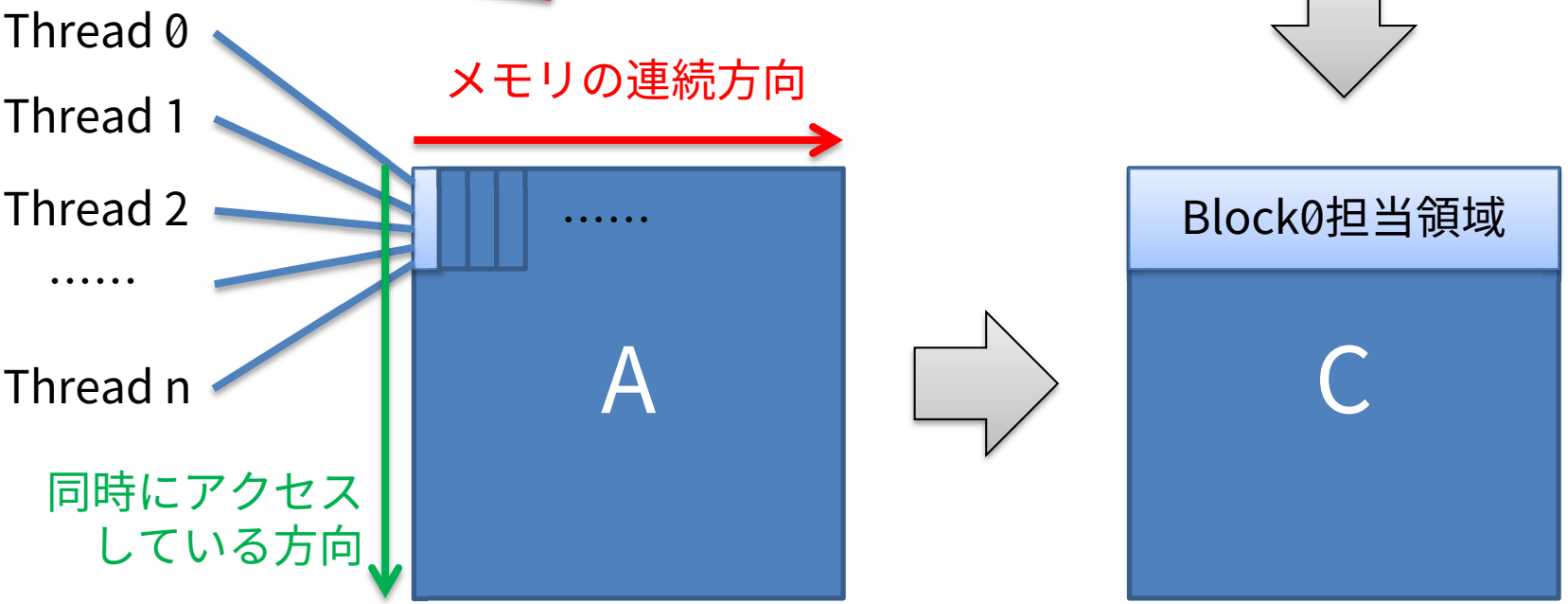



5Thread*4Blockの場合の担当範囲例

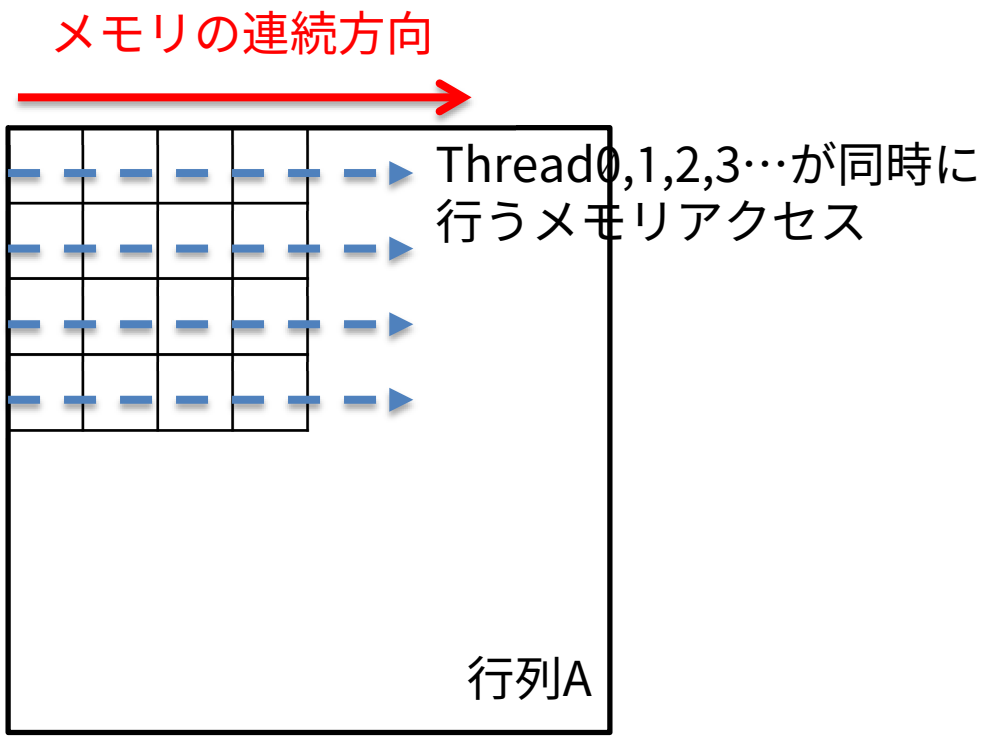
- 最外ループを並列化
- blockIdx, blockDim, threadIdxを使ってIDを得る
  - この例ではブロック・スレッドともに一次元を想定している
  - ～IdxでID、～Dimで総数を取得できる
- 各スレッドが行列の１行を担当するため、スレッド数×ブロック数＝Nである必要がある
- 並列化により性能が向上するはず？

# メモリアクセスの確認

```
k = blockIdx.x*blockDim.x + threadIdx.x;
for(j=0; j<N; j++){
  tmp = 0.0f;
  for(i=0; i<N; i++){
    tmp += A[k*N+i] * B[i*N+j];
  }
  C[k*N+j] = tmp;
}
```

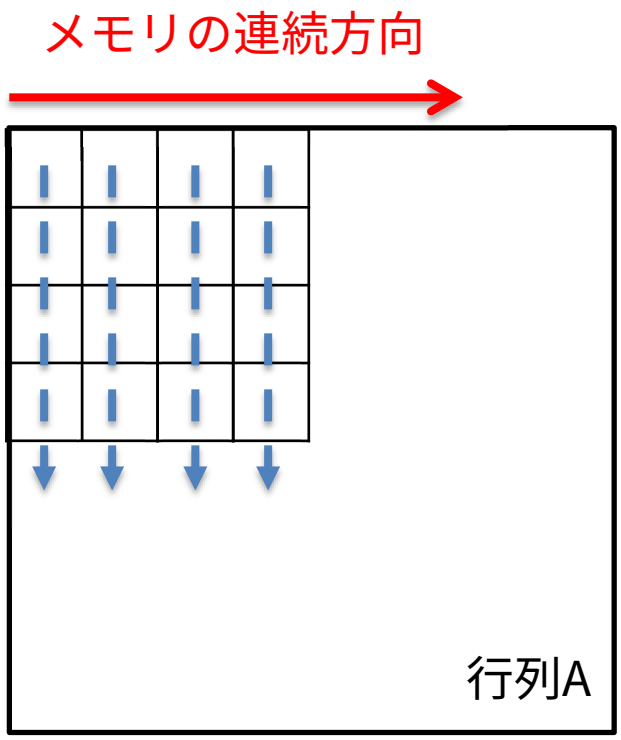


理想的なメモリアクセス




「コアレスなメモリアクセス」
が行われるため高速

現在のメモリアクセス

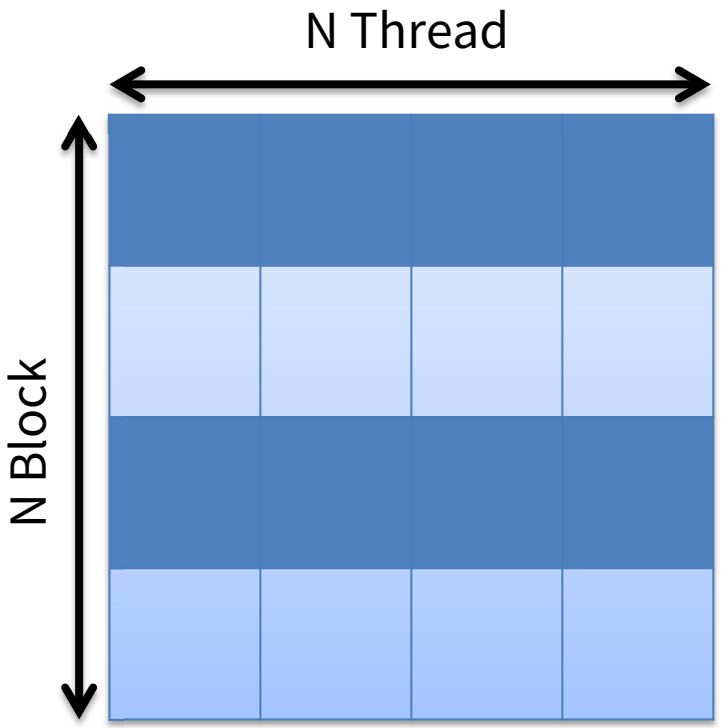



「コアレスなメモリアクセス」
が行われないため低速

- 並列化自体はできているため性能は向上する
- 不連続なメモリに一度にアクセスしているのを修正・解消すれば、もっと性能が向上するはず

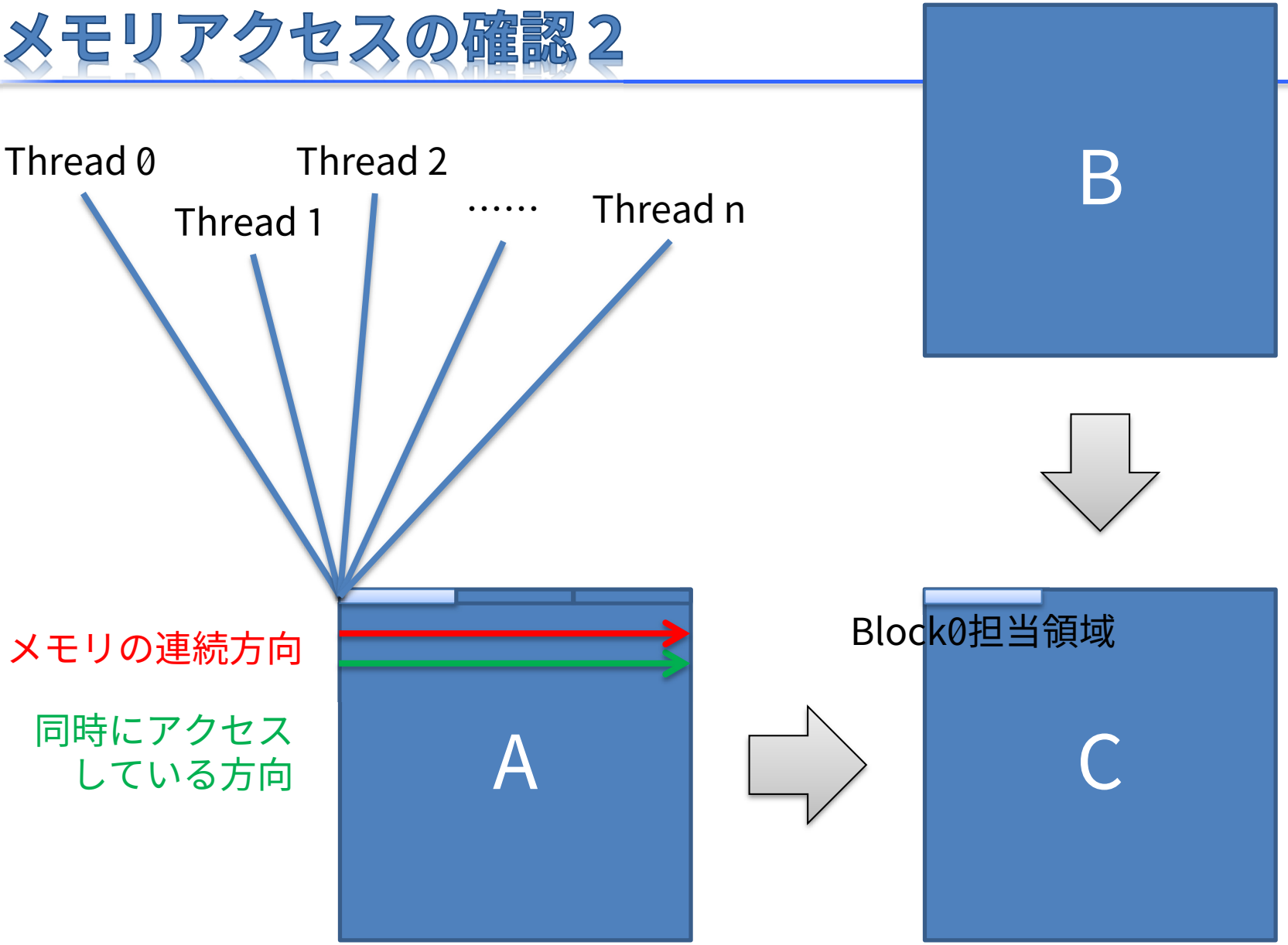
# 二次元的な並列化

```
__global__ void gpukernel
(int N, float *C, float *A, float *B){
  int i, j, k;
  float tmp;
  k = blockIdx.x;
  j = threadIdx.x;
  tmp = 0.0f;
  for(i=0; i<N; i++){
    tmp += A[k*N+i] * B[i*N+j];
  }
  C[k*N+j] = tmp;
}
```

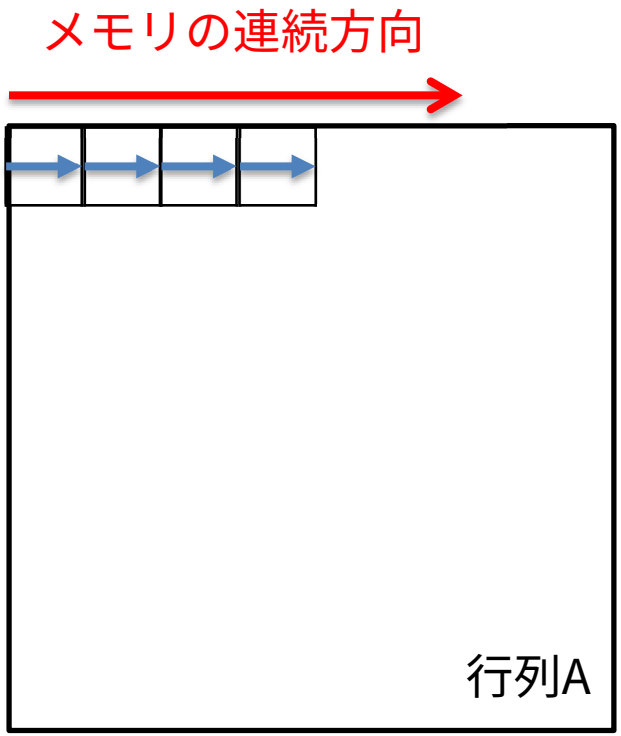



- 各スレッドが計算結果行列の１要素ずつを担当するイメージ
  （要素が多すぎる場合には複数要素を担当する、などの改善も可能）
- 同一Block内のThreadはメモリアクセス方向に並ぶ：メモリアクセスが不連続にならないため性能改善する？

# メモリアクセスの確認２



現在のメモリアクセス




Thread0,1,2,3…が同時に
行うメモリアクセス

- ちがう、そうじゃない
- 別のスレッドがGlobalMemoryから取得済の行列Aのデータを共有したい

# SharedMemory：局所的な共有メモリ

- 共有範囲
  - 同一ブロック内のスレッド群
- 利点
  - 高速（レジスタ並）
- 注意点
  - 小容量：FermiではSMxあたり48KB
  - バンクコンフリクト：メモリアクセスパターンによっては性能が低下
- 使い方
  - 記述：__shared__ 接頭辞
  - よくある使い方：GlobalMemoryからコアレスなメモリアクセスでデータを取得し、計算に使う
    - SharedeMemory内ではランダムなメモリアクセスでも高速

# 二次元的な並列化＋SharedMemoryの利用

```
#define MAX_SM 1024
__global__ void gpukernel
(int N, float *C, float *A, float *B){
  int i, j, k, ntx;
  float tmp;
  __shared__ float sA[MAX_SM];
  k = blockIdx.x;
  j = threadIdx.x;
  ntx = blockDim.x;
  tmp = 0.0f;
  for(i=j; i<N; i+=ntx){
    sA[i] = A[k*N+i];
  }
  __syncthreads();
  for(i=0; i<N; i++){
    tmp += sA[i] * B[i*N+j];
  }
  C[k*N+j] = tmp;
}
```

GlobalMemoryに連続読み込みアクセスしてSharedMemoryへデータを格納

SharedMemory格納済みのデータを用いて計算、Registerへ

※__syncthreadsはN>32の場合のみ必要

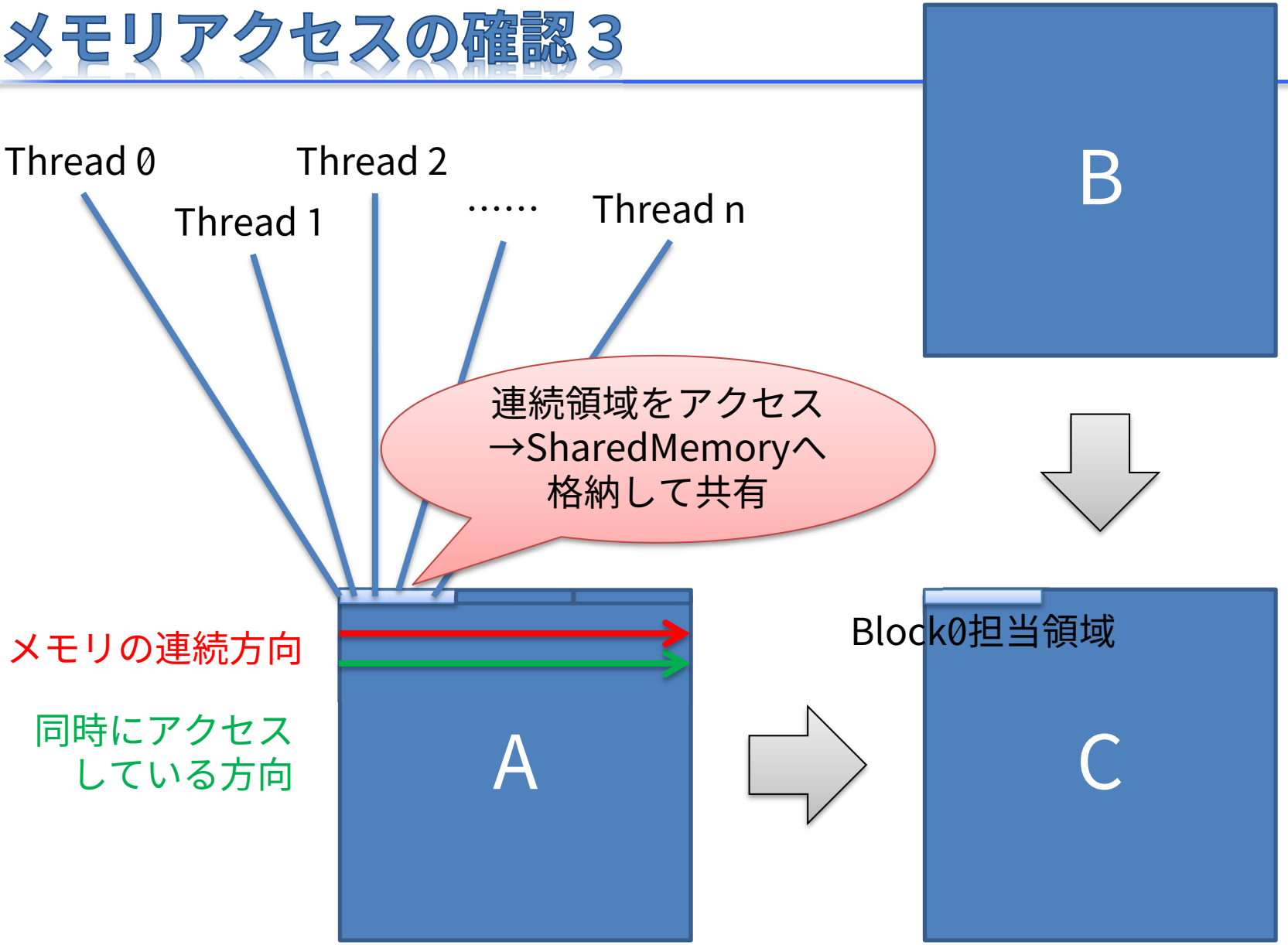
- SharedMemoryを用いてBlock内でデータを再利用
  - 共通してアクセスするデータをSharedMemoryに格納しておいて再利用する
  - 説明を簡単にするため固定長のSharedMemoryを用意したが、動的な確保も可能
    - サイズを未指定（[]）にしておき、<<< >>>の第3引数で指定
- 総命令実行回数自体は増加、問題サイズが大きくないとペイしない？

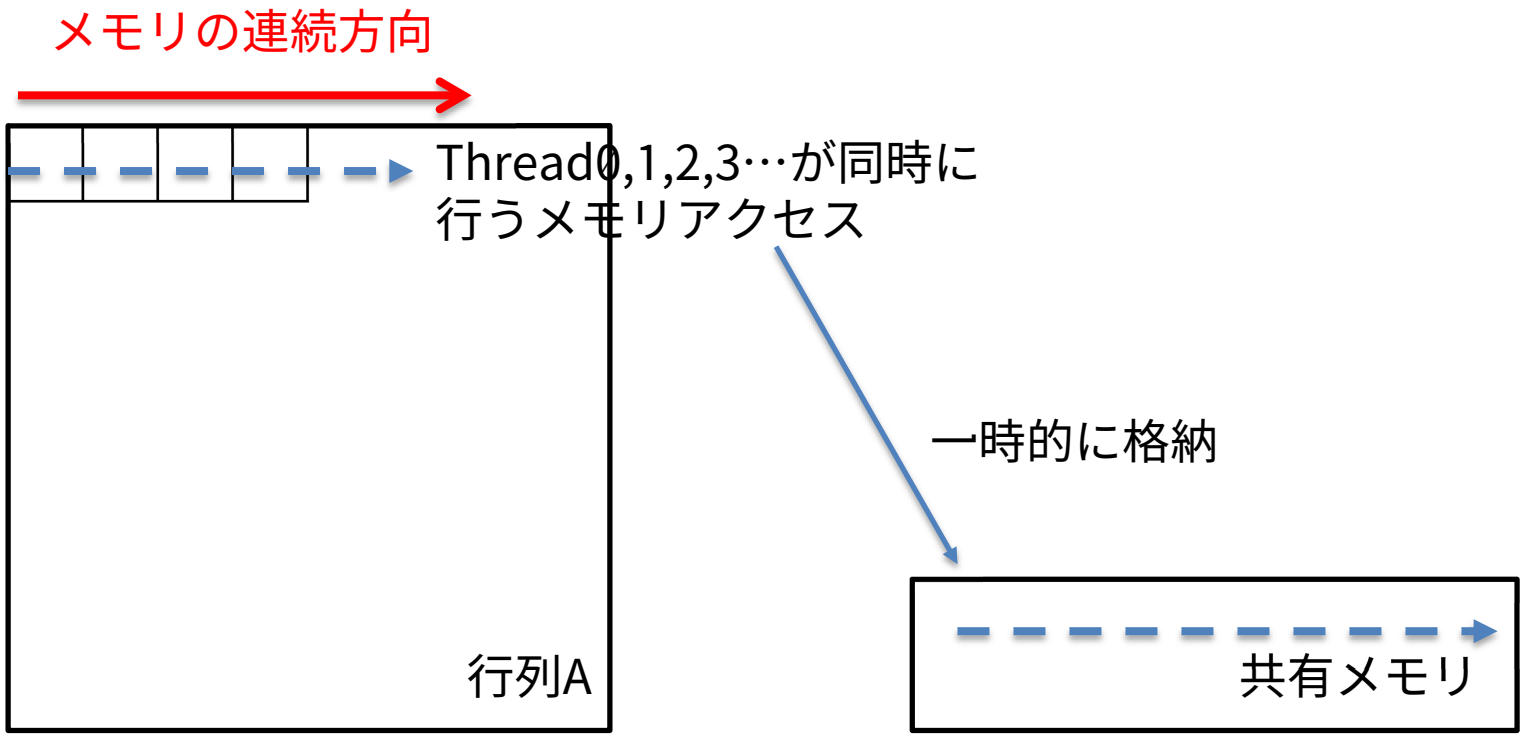
CUDA Fortranの場合は
real, shared :: sA(MAX_SM)
のように指定

# Tips：同期について

- 同一WARP内のスレッド群は常に同期して動作している
  - 乱すことは不可能
    - メモリアクセスの待ち時間が均一でないときは遅いスレッドが足を引っ張る
- 同一ブロック内のスレッド群（異なるWARPのスレッド同士）は同期を取ることができる
  - __syncthreads();
- 異なるブロックをまたいだ同期は不可能
  - （後述のatomic関数を使えば同期のようなこともできなくはないが、非推奨）

# メモリアクセスの確認３






「コアレスなメモリアクセス」
が行われるため高速

GlobalMemoryの代わりに
SharedMemoryを用いて計算
（その後、結果をコアレスな
メモリアクセスで行列Cに書
き戻す）

# 二次元的な並列化＋TextureMemoryの利用

```
texture<float, 1>texA;
__global__ void gpukernel
(int N, float *C, float *B){
  int i, j, k;
  float tmp;
  k = blockIdx.x;
  j = threadIdx.x;
  tmp = 0.0f;
  for(i=0; i<N; i++){
    tmp += tex1Dfetch(texA,k*N+i) * B[i*N+j];
  }
  C[k*N+j] = tmp;
}
```

CPU側コード

```
size_t offset = size_t(-1);
cudaMemcpy(
  d_A, A, sizeof(float)*N*N,
  cudaMemcpyHostToDevice
);
cudaBindTexture(&offset, texA, d_A);
```

- TextureMemoryでメモリアクセスを高速化
  - キャッシュ効果があるためSharedMemoryを使うのに似た効果が期待される
  - Kepler以降ではReadOnlyDataCacheを使うと良い
  - 本当は二次元空間的な補間が使えるときなどに有効な方法

# 発展課題：さらなる高速化に向けて

- 行列Bに関する最適化
  - 行列Aに関する最適化しか考えていない。行列Bについてはどうか？
  - ヒント：行列A（横方向）と行列B（縦方向）の両方でSharedMemoryを活用するにはどうすれば良いだろうか？
- 様々な数の最適化
  - スレッド数、ブロック数、SharedMemoryに格納する単位……
  - ベストな数を選ぶことで高い性能が得られるはずである

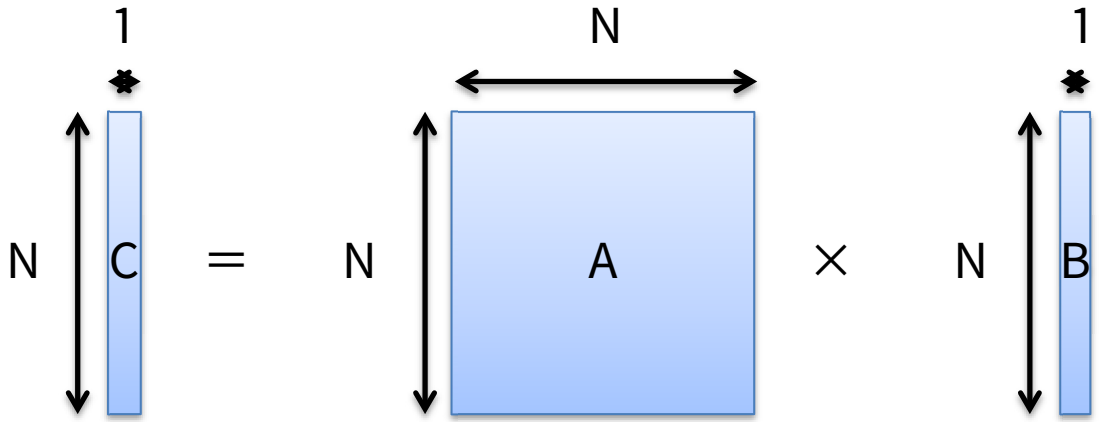
# nvvpを用いた性能の確認

- 可視化機能を持つプロファイラを用いて性能の差を視覚的に理解する

- デモ（実演）による紹介
  - 準備：HA-PACSにSSHログインする際に-Yオプションをつけておく
  - インタラクティブジョブを実行する
    `qsub_gpu -I -X -A GPUSEMINAR -q gpuseminar`
  - nvvpコマンドを実行して起動

# 行列ベクトル積の行単位並列化

- 行列ベクトル積の場合はどうだろうか？
- ブロックごとに1行の計算を担当することを考える
  - 行列データの再利用性がないため、行列積のような最適化の余地がない
  - コアレスなメモリアクセスは必須
    - 連続するスレッドが配列を順番にアクセスすれば良い、簡単
  - ブロック内での足しあわせ（リダクション）はどうする？



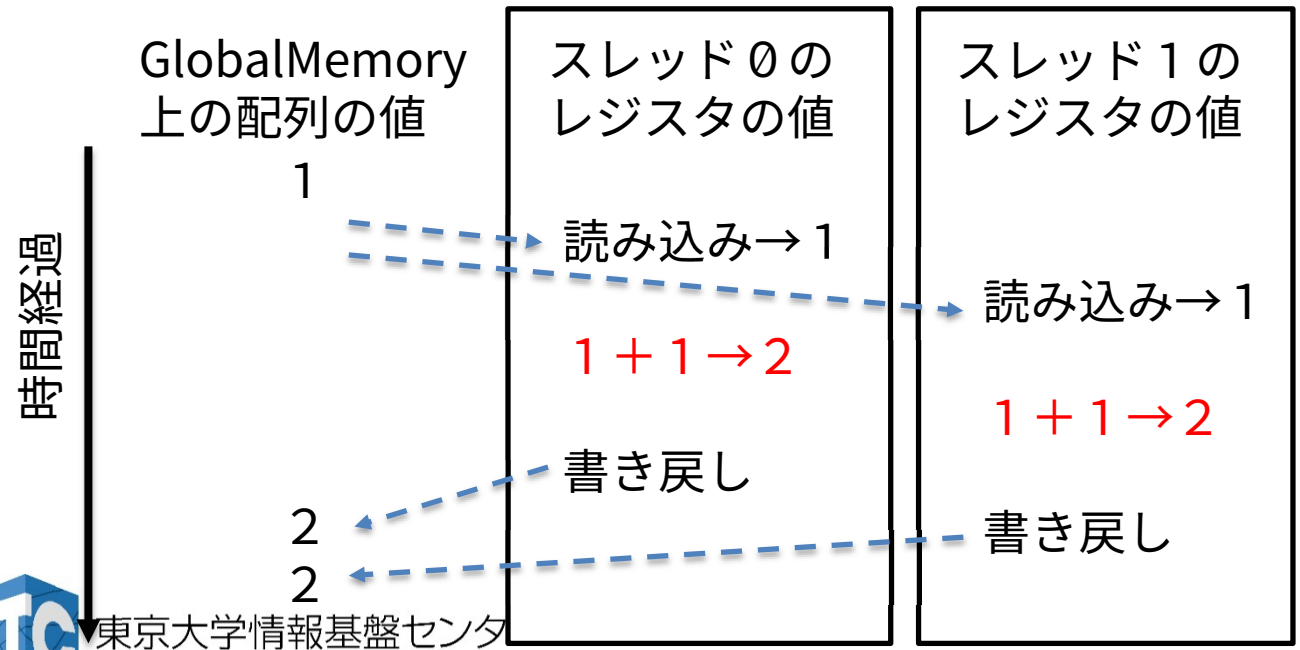
# GPUカーネルの実装

```
int tid = threadIdx.x;
int ntx = blockDim.x;
int bid = blockIdx.x;

float tmp = 0.0f;
for(j=tid; j<N; j+=ntx){
  tmp += A[bid*N + j] * B[j];
}
  // この時点で各スレッドは結果の一部を持った状態
result = ……;   // どうにかしてスレッド間で足しあわせたい
if(tid==0){     // スレッドID=0のスレッドが計算結果を書き戻して終了
  C[bid] = result;
}
```

- ブロックID＝行番号
- 行内のスレッド群が全体で一行を計算

# 並列リダクション演算（加算）を考える

- OpenMPやOpenACCでは指示文を一行入れるだけ
- CUDAではどのように行えば良いだろうか？
  - 何も考えずにGlobalMemoryに足し合わせると、タイミングによって値が変わってしまう
    - あるスレッドが値を読み込んで足して書き戻す間に、他のスレッドが割り込む可能性がある




「＋１」を2回行ったはずなのに、１しか増えていない……

# atomic演算

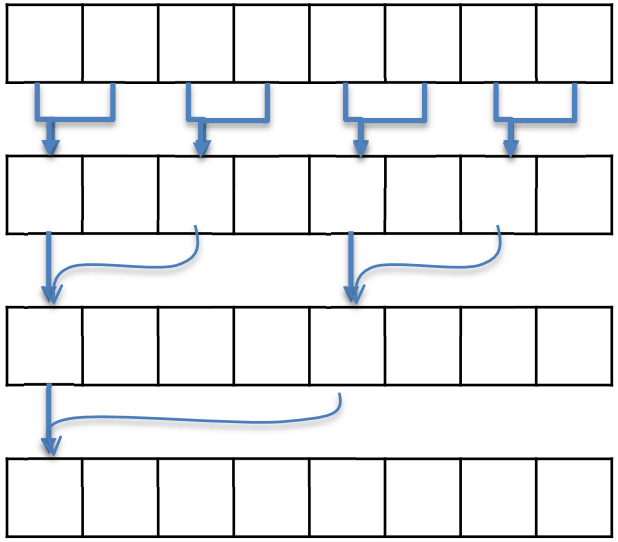
- 他のスレッドに割り込まれずにGlobalMemory上の値を更新するための関数群が提供されている
  - atomicAdd, atomicSub, atomicExch, atomicMin, atomicMax, atomicInc, atomicDec, atomicCAS, atomicAnd, atomicOr, atomicXor

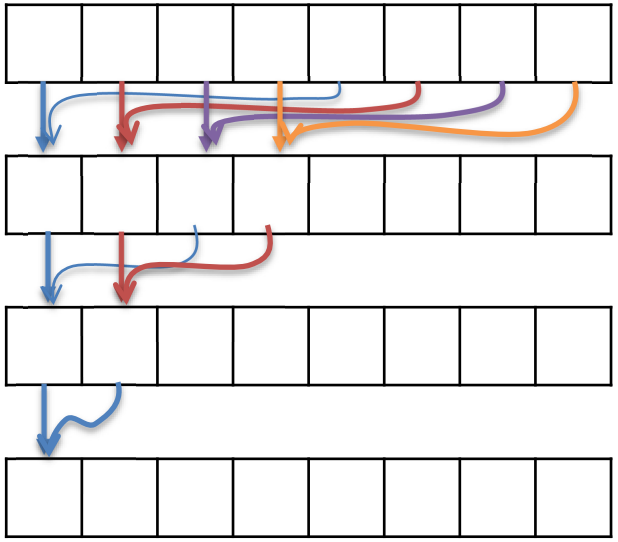
```
atomicAdd(&hoge, 1.0f);    atomicSub(&hoge, -1);
```

  - メリット：割り込まれる心配が不要になる
  - デメリット：性能低下要因、多用しすぎには注意
    - 新しい世代のGPUほどatomic演算も高性能、様々なデータ型に対応
  - GPU全体でのリダクション演算も不可能ではない
    - が、数千スレッドが同一の変数に対してリダクション演算を行うのは推奨できない
    - 同一ブロック内でのリダクション演算は別の方法で行い、ブロック間のリダクションにatomic演算を使うのが妥当

# WARP内での並列リダクション

- 色々な方法が考えられるが……

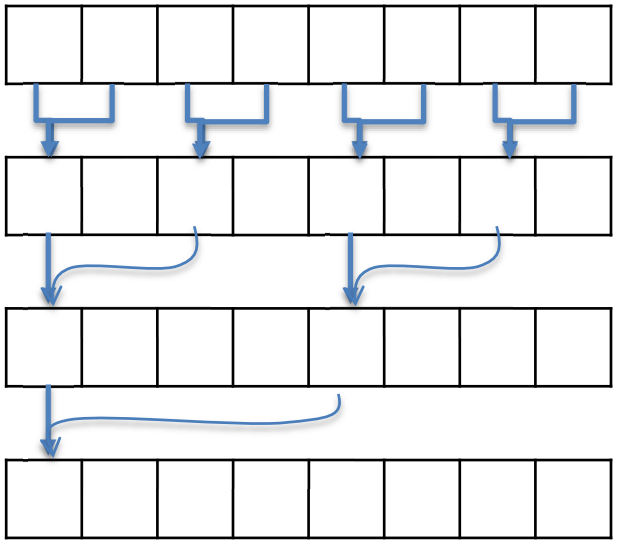




# リダクションの実装例１

CUDAサンプルの
6_Advanced/reduction/reduction_kernel.cu
より

```
template <class T>

T *sdata = SharedMemory<T>();
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

sdata[tid] = (i < n) ? g_idata[i] : 0;

__syncthreads();

for (unsigned int s=1; s < blockDim.x; s *= 2)
{
    int index = 2 * s * tid;

    if (index < blockDim.x)
    {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

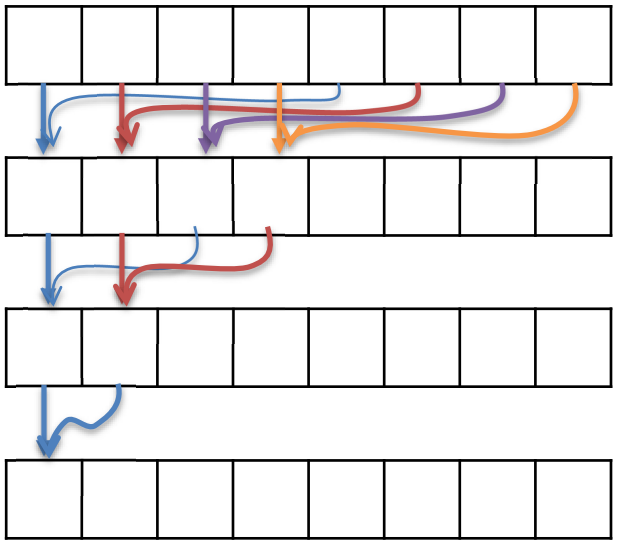


Divergent WARPだらけになり良い性能は得られない
（足し合わせる順序よりむしろ実装の仕方が悪い）

# リダクションの実装例２

CUDAサンプルの
6_Advanced/reduction/reduction_kernel.cu
より

```
template <class T, unsigned int blockSize>

T *sdata = SharedMemory<T>();
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;

T mySum = (i < n) ? g_idata[i] : 0;
if (i + blockSize < n)mySum += g_idata[i+blockSize];

sdata[tid] = mySum;
__syncthreads();
```



```
if((blockSize >= 512)&&(tid < 256)){ sdata[tid] = mySum = mySum + sdata[tid + 256];}__syncthreads();
if((blockSize >= 256)&&(tid < 128)){ sdata[tid] = mySum = mySum + sdata[tid + 128];}__syncthreads();
if((blockSize >= 128)&&(tid <  64)){ sdata[tid] = mySum = mySum + sdata[tid +  64];}__syncthreads();
if((blockSize >=  64)&&(tid <  32)){ sdata[tid] = mySum = mySum + sdata[tid +  32];}__syncthreads();
if((blockSize >=  32)&&(tid <  16)){ sdata[tid] = mySum = mySum + sdata[tid +  16];}__syncthreads();
if((blockSize >=  16)&&(tid <   8)){ sdata[tid] = mySum = mySum + sdata[tid +   8];}__syncthreads();
if((blockSize >=   8)&&(tid <   4)){ sdata[tid] = mySum = mySum + sdata[tid +   4];}__syncthreads();
if((blockSize >=   4)&&(tid <   2)){ sdata[tid] = mySum = mySum + sdata[tid +   2];}__syncthreads();
if((blockSize >=   2)&&(tid <   1)){ sdata[tid] = mySum = mySum + sdata[tid +   1];}
```

Divergent WARPが発生しない実装の工夫のおかげで高速
一見すると分岐だらけだが、テンプレート展開によって消滅する

# 演習

- 行列ベクトル積を作成する
  - リダクション方法を変えて性能を比較する
  - 並列化方法を変えて性能を比較する
    - 「１ブロックあたり１行」をやめる（WARPあたり１行など）
    - 行列サイズ・並列度・性能の関係を調べる

CUDAを用いた並列リダクションについては以下の資料に詳細に書かれているので参考にしてください
http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf
その他にも各種資料がオンラインで公開されています（CUDA Toolkitをインストールする際に入手することもできます）
http://docs.nvidia.com/cuda/cuda-samples/index.html

# まとめ

- この時間に扱ったこと
  - CUDA（CUDA C, CUDA Fortran）の基本的な使い方
  - CUDAプログラムの最適化方法のほんの一部
    - スレッドとブロックを使った並列化
    - コアレスなメモリアクセス
    - SharedMemory
    - TextureMemory
    - atomic演算
    - リダクション処理

「入門」であり、
いずれも触った程度。
GPUの持つ性能を引きだす
にはさらなる経験が必要。

- 扱っていないこと
  - Kepler以降の新機能
  - CPU-GPU間データ転送を考慮した最適化（ストリームなど）
  - 複数GPUの活用