大島聡史
（並列計算分科会 主査、東京大学 情報基盤センター 助教）

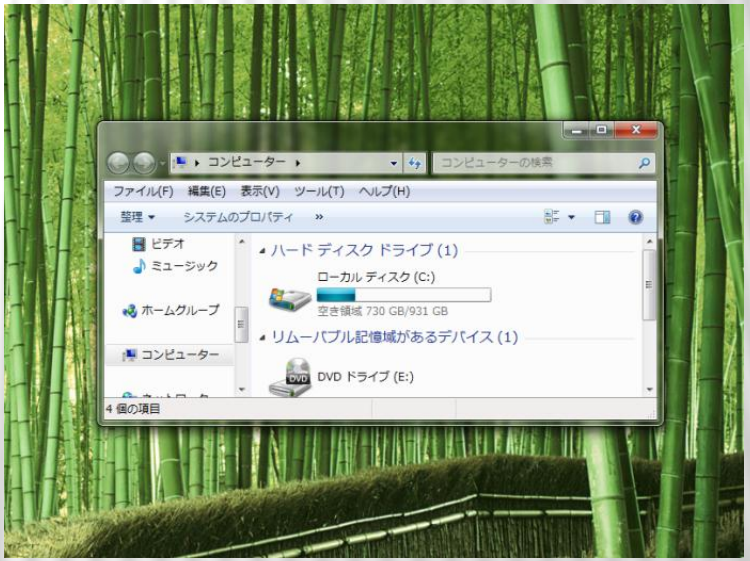
GPGPUイントロダクション

目的

- 昨今注目を集めているGPGPU（GPUコンピューティング）について紹介する
  - GPGPUとは何か？
  - 成り立ち
  - 特徴
  - 用途（ソフトウェアや研究例の紹介）
  - 使い方（ライブラリ・言語）
  - CUDA
  - GPGPUにおける課題

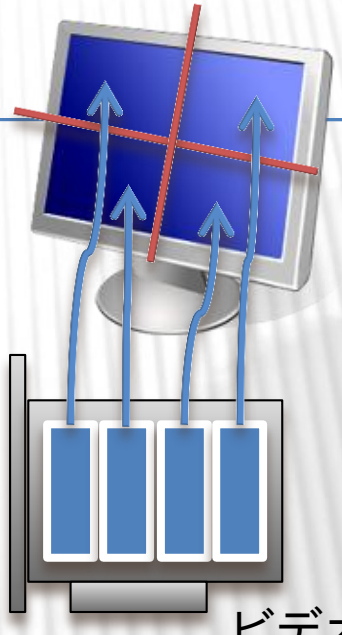
GPGPUとは何か？

- GPGPU
  - General-Purpose computation using GPUs
  - GPUを用いた汎用計算
- GPU
  - Graphics Processing Unit
  - 画像処理用のハードウェア、ビデオカード（上のLSI）
    - NVIDIA: GeForce, Quadro, Tesla
    - AMD(ATI): Radeon, FireGL, FireStream
    - Intel: Intel HD Graphics
  - 役割：3D描画、HD出力、動画エンコード・デコード、etc.



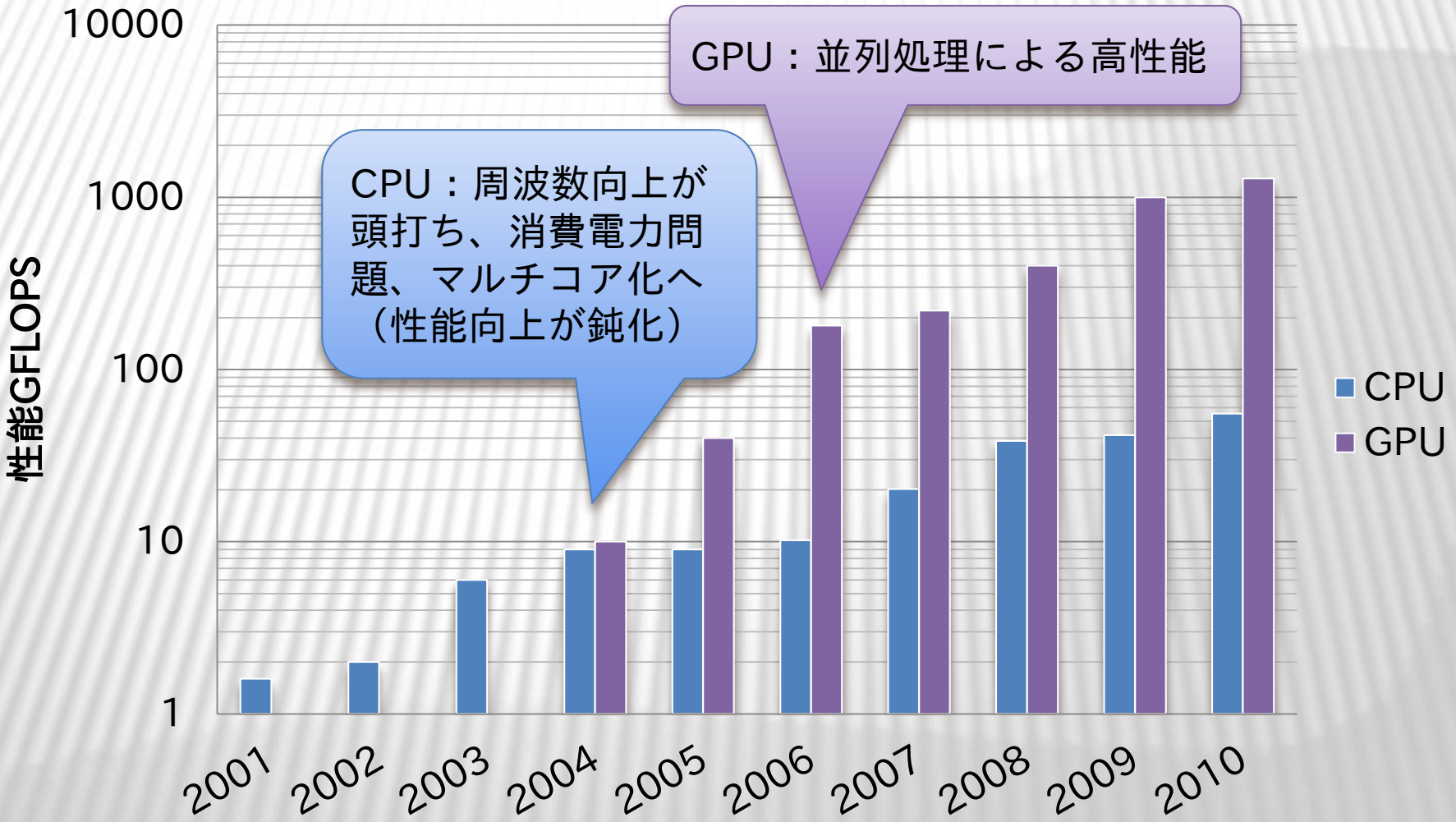
GPGPUの成り立ち

- 背景
  - 高速・複雑な三次元画像処理への要求
  - GPUハードウェアの並列化とプログラマブル化
  - 描画処理プロセスにおける汎用数値演算の導入
  - 汎用演算への利用可能性



ビデオカード

GPUとCPUの性能


GPU：並列処理による高性能
CPU：周波数向上が頭打ち、消費電力問題、マルチコア化へ（性能向上が鈍化）
性能GFLOPS
10000
1000
100
10
1
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
CPU
GPU

GPGPUの特徴（GPUとCPUのHWの違い）

- GPU
  - 並列度：高（100～）
  - クロック：低（～2GHz）
  - 計算コア：単純
  - ストリーミング性能重視
  - （戦略的に）隠されたアーキテクチャ
- CPU
  - 並列度：低（～8）
  - クロック：高（2GHz～）
  - 計算コア：複雑
  - キャッシュ性能重視
  - （比較的）透明性のあるアーキテクチャ

ハードウェアの特徴＝性能の特性

- GPUの得意とする計算
  - 並列度が高い
  - 計算内容が単純（分岐が少ない）
  - 連続メモリアクセスが多い
- アーキテクチャの改良によって不得意な計算は減りつつある
  - 分岐や不連続メモリアクセスがあっても性能が落ちにくくなってきている

使い方（ライブラリ・言語）

- GPGPU黎明期
  - グラフィックスプログラミング
    - DirectX + HLSL、OpenGL + GLSL
  - 描画処理を用いて計算
    - 画像処理の知識が必要（習得が大変）
    - 画像処理と直接関係のない情報が提供されない（最適化が非常に困難）
- 現在
  - CUDA, OpenCL, etc.
    - CPU向けのプログラミングに近づいた
  - GPUに対応したライブラリや言語
    - 数値計算ライブラリ、CUDA対応言語

グラフィックスプログラミングの例

- グラフィックスAPI（DirectX, OpenGL）による描画処理＋シェーダ言語（HLSL, GLSL）による演算

```
void gpumain(){
    vec4 ColorA = vec4(0.0, 0.0, 0.0, 0.0); vec4 ColorB = vec4(0.0, 0.0, 0.0, 0.0);
    vec2 TexA = vec2(0.0, 0.0); vec2 TexB = vec2(0.0, 0.0);
    TexA.x = gl_FragCoord.x; TexA.y = gl_FragCoord.y;
    TexB.x = gl_FragCoord.x; TexB.y = gl_FragCoord.y;
    ColorA = texRECT( texUnit0, TexA );
    ColorB = texRECT( texUnit1, TexB );
    gl_FragColor = F_ALPHA*ColorA + F_BETA*ColorB;
}
```

GPUの処理
(GLSL)
各ピクセルに対して実行される

シェーダ言語を用いた配列加算
（vc=a*va + b*vb）の例

```
void main(){
    glutInit( &argc, argv );
    glutInitWindowSize(64,64);glutCreateWindow("GpgpuHelloWorld");
    glGenFramebuffersEXT(1, &g_fb);
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, g_fb);
    glGenTextures(4, g_nTexID); // create (reference to) a new texture
    glBindTexture(opt1, texid);
    glTexParameteri(opt1, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(.....);
    glTexImage2D(opt1, 0, opt2, width, height, 0, GL_RGBA, GL_FLOAT, 0);
    ……（以下省略）
```

CPUの処理
(OpenGL)

グラフィックスプログラミングの例

- グラフィックスAPI（DirectX, OpenGL）による描画処理＋シェーダ言語（HLSL, GLSL）による演算

```
void gpumain(){
    vec4
    vec2
    TexA
    TexB
    Colo
    Colo
    gl_Fr
}
```

```
void ma
    glutI
    glutI
    glGe
    glBin
    glGe
    glBin
    glTex
    glTexParameteri(.....);
    glTexImage2D(opt1, 0, opt2, width, height, 0, GL_RGBA, GL_FLOAT, 0);
    ……（以下省略）
```

「**プログラミングの難しさ**」が
GPGPUの最大の課題（普及の障害）
と言っても過言ではなかった

GPGPU用の言語（ストリーミング言語など）の研究なども行われたが、普及はしなかった
※ここでは割愛

CUDA

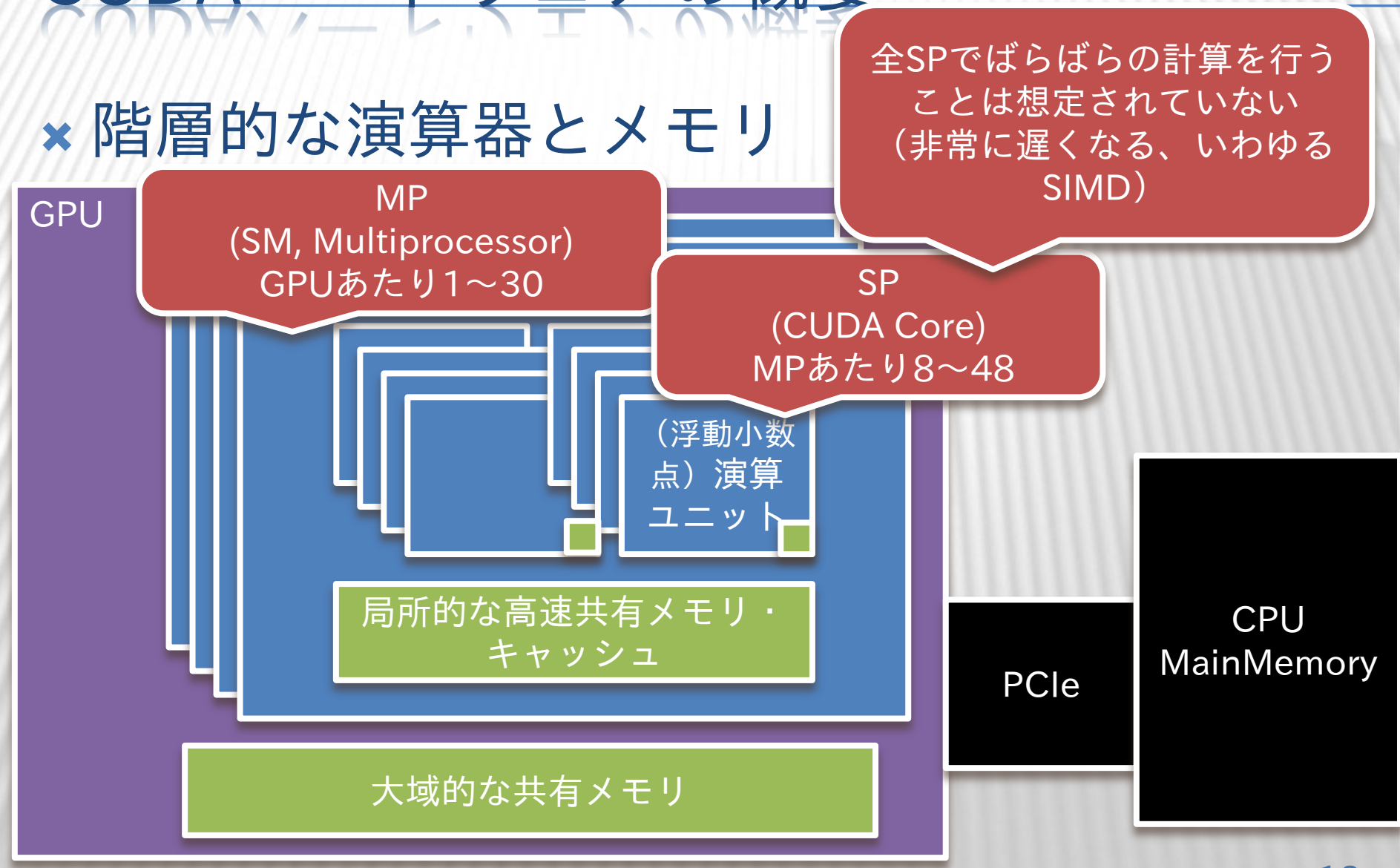
- NVIDIA社製GPUのアーキテクチャ・開発環境（2007年に一般公開）
  - 「GPUの内部」をある程度公開
  - 開発環境を無料で提供
  - 画像処理の知識がなくても利用可能
  - （まともな）最適化が可能

5 分でわかる（かもしれない）CUDA

- ハードウェアの概要
- ソフトウェアの概要
- 最適化の基本戦略

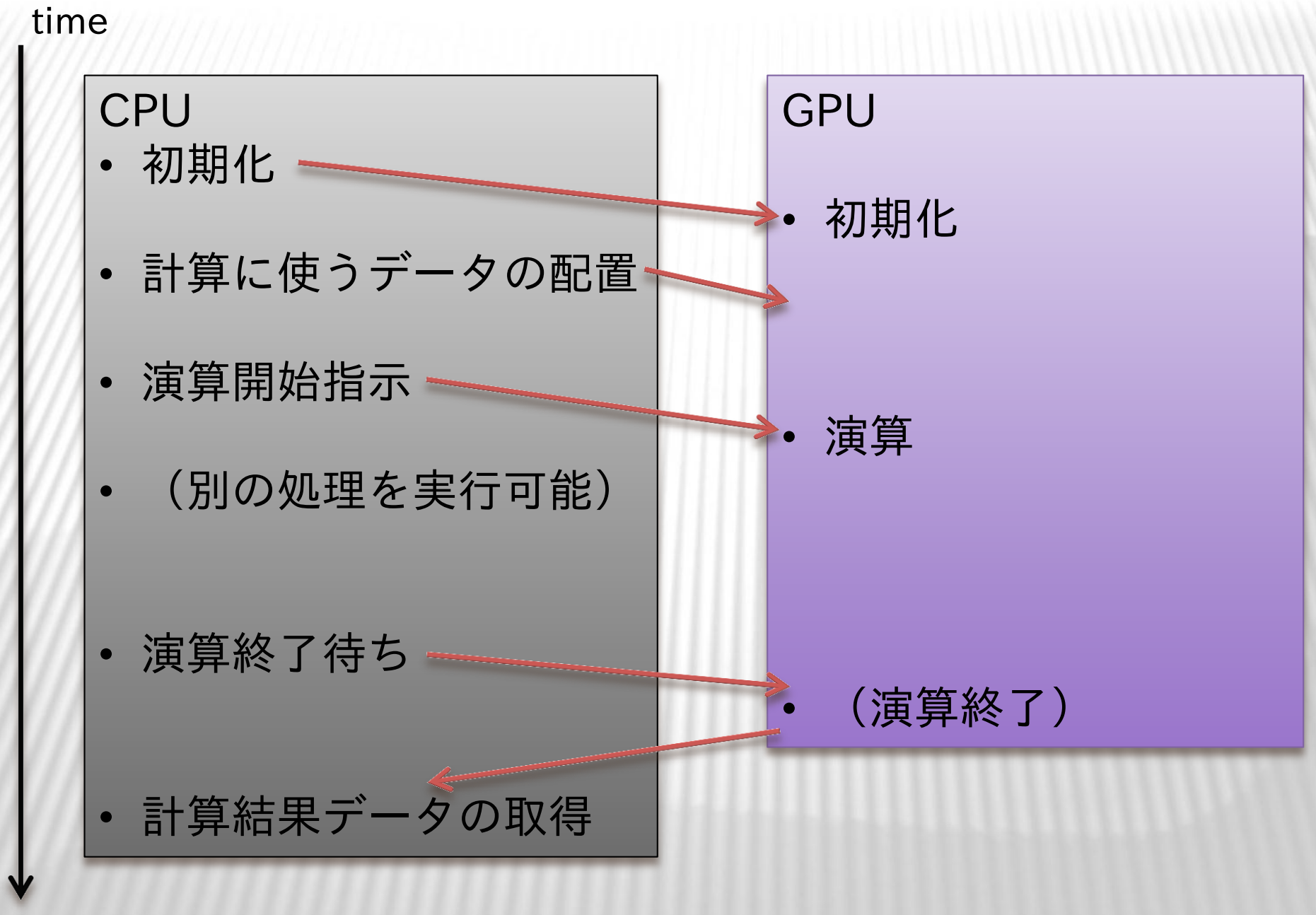
CUDAハードウェアの概要

- 階層的な演算器とメモリ



実行イメージ

- ポイント
  - GPUはCPUからの指示で動く
  - GPUの実行単位は関数
  - 関数呼び出し時に並列度を指定
    - Block、Thread


time
CPU
• 初期化
• 計算に使うデータの配置
• 演算開始指示
• （別の処理を実行可能）
• 演算終了待ち
• 計算結果データの取得
GPU
• 初期化
• 演算
• （演算終了）

CUDAプログラムの概要

- CUDA C
  - C/C++を若干拡張した言語
  - 関数指示詞：GPU上で実行する関数を明示
  - メモリ指示詞：GPU上のメモリ配置を明示
  - GPU制御記述：GPUに処理を行わせる記法
  - 専用のAPI：GPUの制御用

※PGIが提供するFortran版もあります

- 必要なもの
  - toolkit + sdk
    - Windows,Linux,MaxOSX用が公開されている
    - デバッガや性能解析ツールも公開されている
    - GCCやVisualStudioも必要

CUDAプログラムの例 1/2

- シンプルな配列加算の例

```
__global__ void gpumain
(float* vc, float* va, float* vb, int nSize, float a, float b){
  for(int i=0; i<nSize; i++){
    vc[i] = a*va[i] + b*vb[i];
  }
}
```

（CPUからGPU上のプロセッサを個別に操作しない）

GPUの処理
blocks,threadsで
指定された数だけ
実行される

CUDAを用いた配列加算
（vc=a*va + b*vb）の例

```
void main(){
  CUT_DEVICE_INIT();
  cudaMalloc((void**)&d_va, n);
  cudaMalloc((void**)&d_vb, n);
  cudaMalloc((void**)&d_vc, n);
  cudaMemcpy(d_va, h_va, n, cudaMemcpyHostToDevice);
  cudaMemcpy(d_vb, h_vb, n, cudaMemcpyHostToDevice);
  gpumain<<< blocks, threads >>>(d_vc, d_va, d_vb, nSize, alpha, beta);
  cudaMemcpy(h_vc, d_vc, n, cudaMemcpyDeviceToHost);
}
```

CPUの処理

CUDAプログラムの例 2/2

シンプルな配列加算の例（並列計算版）

```
__global__ void gpumain
(float* vc, float* va, float* vb, int nSize, float a, float b){
  int begin = blockIdx.x*blockDim.x + threadIdx.x;
  int step = gridDim.x*blockDim.x;
  for(int i=begin; i<nSize; i+=step){
    vc[i] = a*va[i] + b*vb[i];
  }
}
```

GPUの処理
blocks,threadsで
指定された数だけ
実行される

- （シェーダと比べて）非常にわかりやすくなった
- CUDAに関する理解は必要

を用いた配列加算
（vc=a*va + b*vb）の例

```
void main(){
  CUT_DEVICE_INIT();
  cudaMalloc((void**)&d_va, n);
  cudaMalloc((void**)&d_vb, n);
  cudaMalloc((void**)&d_vc, n);
  cudaMemcpy(d_va, h_va, n, cudaMemcpyHostToDevice);
  cudaMemcpy(d_vb, h_vb, n, cudaMemcpyHostToDevice);
  gpumain<<< blocks, threads >>>(d_vc, d_va, d_vb, nSize, alpha, beta);
  cudaMemcpy(h_vc, d_vc, n, cudaMemcpyDeviceToHost);
}
```

CPUの処理

BlockとThread

- イメージとしては、CPUコアとプロセスおよびスレッドとの関係
- BlockはSMに、ThreadはSPに割り当てられて実行される
  - 物理数を超えたら時分割実行
  - 基本的に全Blockの全Threadが同一の関数を実行する（最近のGPUでは緩和された）
  - IDを利用すれば計算対象データを選択可能
  - 同一Block内のThreadに同じ処理をさせると高性能（正確には32個単位）

最適なBlock・Thread数

- CPUプログラミング
  - コア数を超えないプロセス・スレッド数が常識
- CUDAプログラミング
  - SP数を大きく超えたThread数が常識
  - SPの切り替えが高速なため、メモリアクセス待ちの間に切り替えて実行することが可能

特性の異なる複数種類のメモリ

- Register
  - SP毎に持つレジスタ（高速）
- GlobalMemory（__device__ 変数）
  - GPU全体で持つ共有メモリ（高速、高レイテンシ、ランダムアクセスが遅い、～１GB程度）
- SharedMemory（__shared__ 変数）
  - MP毎に持つ共有メモリ（高速、低レイテンシ、ランダムアクセスも高速、16KB/48KB）関数単位
- TextureMemory
  - GPU全体で持つ読み取り専用メモリ（≒キャッシュ効果や補完機能のあるGlobalMemory）
- ConstantMemory（__constant__ 変数）
  - GPU全体で持つ読み取り専用メモリ（≒キャッシュ効果のあるGlobalMemory、64KB）

Threadが同時に連続アドレスをアクセスすると、（GPU内部では）一命令でまとめてアクセス可能＝性能向上（コアレスなメモリアクセス）

- + SP毎に持つレジスタ（高速）
- GlobalMemory（__device__ 変数）
  - + GPU全体で持つ共有メモリ（高速、高レイテンシ、ランダムアクセスが遅い、～１GB程度）
- SharedMemory（__shared__ 変数）
  - + MP毎に持つ共有メモリ（高速、低レイテンシ、ランダムアク

（メモリがバンクに分かれているため）特定のアドレス幅で同時にアクセスすると衝突＝性能低下（バンクコンフリクト回避）

ッシュ効果や補

- ConstantMemory（__constant__ 変数）
  - + GPU全体で持つ読み取り専用メモリ（≒キャッシュ効果のあるGlobalMemory、64KB）

CUDA最適化プログラミングの基本戦略

1. 多数のBlock/Threadを密に動かす
   + 並列度を上げる、動作がばらつかないようにする
2. メモリを適切に利用する
   + コアレスメモリアクセス、バンクコンフリクト回避
3. 分岐をなくす、ループ展開する
   + GPUは分岐処理に弱い
4. データ転送の隠蔽・CPUの活用
   + データ転送と演算のオーバーラップが可能
   + 場合によってはCPUでの計算も行う
     × 基本的にはGPUで全ての演算を行えるようにするべきだが

◆ 計算とメモリのどちらが頭打ちになるか考える

５分でわかる（かもしれない）CUDA

- （おわかりいただけましたか？）

- 正しく動かすだけならそれほど難しくはない
- 性能最適化は大変
  - 特に最近はアーキテクチャの更新が頻繁で大変

- 無料で公開されている＋GPUが無くても試行は可能なので、是非使ってみてください

GPGPU対応ソフトウェアや研究の例

- GPGPU黎明期
  - 描画処理の中の計算、描画処理を用いた計算
    - 照明・陰影計算中の算術演算、衝突判定、etc.
    - 行列積、LU分解、etc.
- 現在（主にCUDA）
  - 動画編集ソフト（リアルタイム編集、形式変換）
  - BLAS、FFT、N体問題、etc.
  - CG法などのソルバー（アルゴリズム）
  - プログラミング言語、その他ライブラリ
  - （描画関連）

GPGPU(CUDA)の現状と将来

- CUDAのトレンド
  - 倍精度演算の性能向上、ECCメモリ対応、統合開発環境(nsight)の公開
    - ユーザのニーズに応えて性能・機能向上
    - GPUのCPU化？
  - スーパーコンピューターへの採用
    - 圧倒的な演算性能は必要不可欠
    - TSUBAME 2.0（東工大）、中国スパコン
- 課題
  - 複数GPU利用、実用アプリケーション、言語
    - PCI-Expressの帯域やメインメモリ容量の消費、プログラミングの難しさ
  - GPUクラスタの活用
    - 最適な問題分配、チェックポイント、メモリの最適利用

GPGPU(GPU)の現状と将来

- GPUの将来（予想）
  - ～５年後
    - さらなる性能向上・機能追加
    - CPUに近づく
      - GPUは汎用化しCPUに近づく
      - CPUはメニーコア化しGPUに近づく
    - GPUスパコンの増加
    - GPUアプリケーションの増加
  - ～10年後
    - 「GPU」ではなくなる？
      - グラフィックスハードウェアとしての要求とHPCハードウェアとしての要求が乖離→「GPUのような何か」

おわりに

- （非常に簡単ではありますが）GPUとGPGPUについて紹介しました
- GPGPUの魅力の一つは「試しやすさ」だと思うので、是非使ってみてください
  - 秋葉原で１万円未満から買えます
- 良い効果は得られないことも多いと思いますが、それが現在のGPUです
  - 特定の問題でしか良い性能は得にくいです
- まだまだ利用コストは高いですが、挑戦する価値はあると思います