数理工学PBL

『スパコンを用いたGPU スクラッチプログラミング』

前半：GPUアーキテクチャとGPUプログラミング

一関高専 未来創造工学科 情報・ソフトウェア系

小池　敦

GPUプログラミング入門

- 目標
  - GPUアーキテクチャの特性を理解した上で
    GPUを用いた並列化ができる
- 対象者
  - C言語等でプログラミングをしたことがあり，
    GPUを用いた高速化に興味がある
  - GPUによる並列計算の概要を知りたい
- 必要な予備知識
  - C言語を触ったことがある

GPUプログラミング入門

- 講習会の内容
  - 前半：GPUアーキテクチャとGPUプログラミング（座学中心）
  - 後半：GPUプログラミング（実習中心）
- 必要な事前準備
  - 大阪大学サイバーメディアセンターの大型計算機試用アカウントを取得し，OCTOPUSにログインできるようにしておく
    http://www.hpc.cmc.osaka-u.ac.jp/service/intro/shiyo/

初日の内容

- GPUアーキテクチャ
  - GPUアーキテクチャの概要とその長所
  - GPUアーキテクチャは世代により大きく変わるが共通のもの＋最新の動向を紹介する
- GPUプログラミング
  - GPUで汎用計算を行う方法を説明する（グラフィックの話はしない）
  - 1つのプログラムを複数GPUアーキテクチャで動作させる仕組み（CUDA）
- 計算機環境お試し

GPU

- Graphics Processing Unit
- 元々はグラフィック処理の専用プロセッサ

Introducing The GeForce GTX 1080 Ti, The World's Fastest Gaming GPU
https://www.geforce.com/whats-new/articles/nvidia-geforce-gtx-1080-ti

GPU

- 高い並列性能を持つため，
  グラフィック処理以外の汎用計算にも
  使われている
  - 最新モデル（GV100）では1デバイスで5120コア
  - 科学技術計算，深層学習などによく使われる
  - NVIDIA社はGPUコンピューティングと呼んでいる
    （元々はGPGPUと呼んでいた）

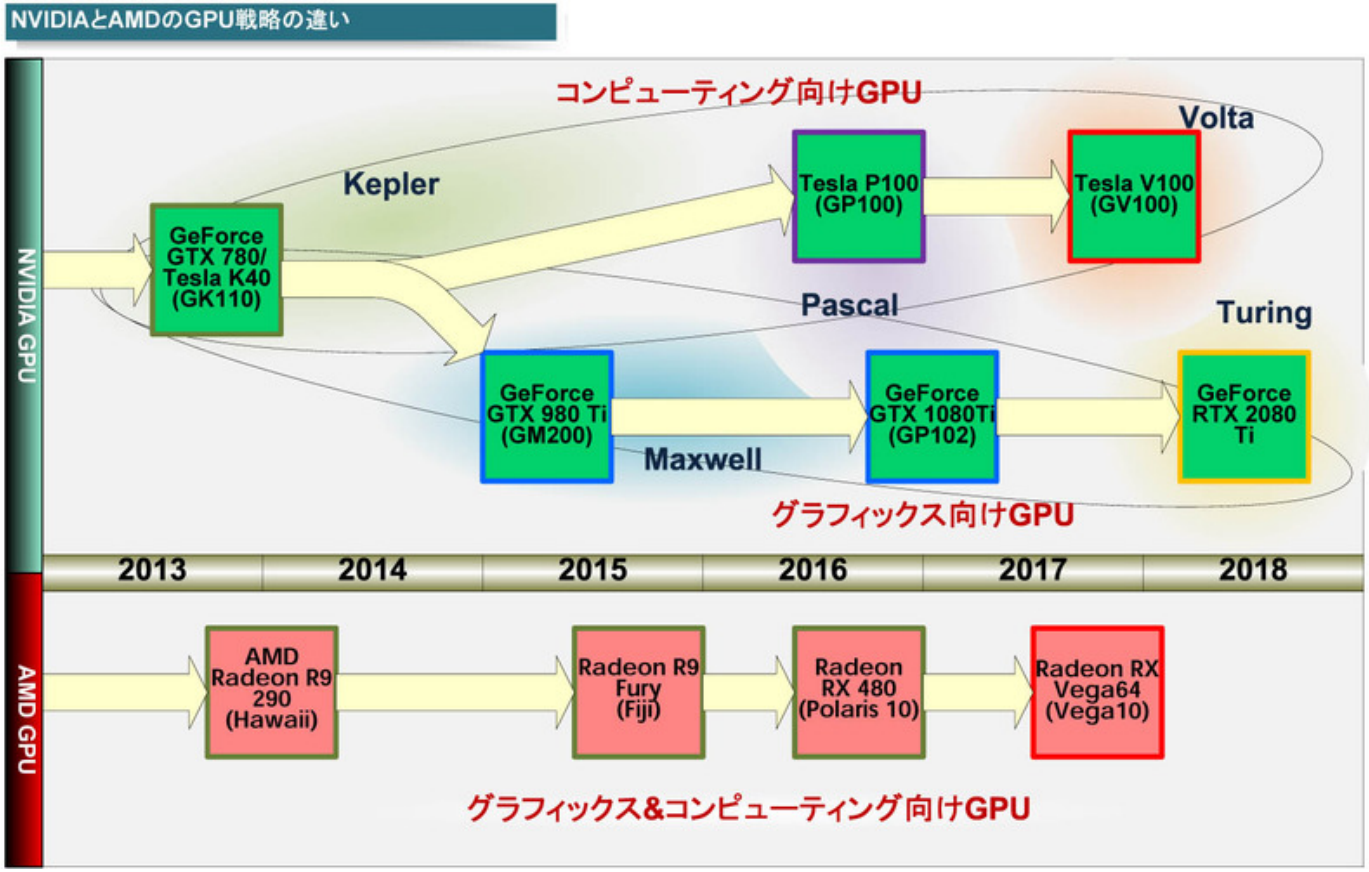
NVIDIA TESLA V100 GPUアーキテクチャ
https://images.nvidia.com/content/pdf/tesla/Volta-Architecture-Whitepaper-v1.1-jp.pdf

GPU

- 主な開発企業
  - NVIDIA
  - AMD
- 汎用計算においては，NVIDIAの1強
  - CUDAと呼ばれるNVIDIA社GPU向けのプログラミング環境（後述）が整備されていることが大きい
    - 高速化しやすい，そこそこプログラミングしやすい
  - 本講習会でもNVIDIA社GPUをターゲットとする

NVIDIA GPU

- 今回はTesla P100を使用(Pascalアーキテクチャ)



後藤弘茂のWeekly海外ニュース
https://pc.watch.impress.co.jp/docs/column/kaigai/1139009.html

GPUプログラミングの情報源

- NVIDIA社ドキュメント
  - CUDA C Programming Guide
    https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
  - NVIDIAのWhite Paper（ハードウェア説明書）
    https://images.nvidia.com/content/pdf/tesla/Volta-Architecture-Whitepaper-v1.1-jp.pdf
  - NVIDIA社（日本）の各種資料（検索する）
    https://www.nvidia.com/content/apac/gtc/ja/pdf/2017/1041.pdf
    http://on-demand.gputechconf.com/gtc/2013/jp/sessions/8002.pdf

GPUプログラミングの情報源

- NVIDIA Developer
  - CUDA Education & Training
    https://developer.nvidia.com/cuda-education-training
  - GPU Accelerated Computing with C and C++
    https://developer.nvidia.com/how-to-cuda-c-cpp
  - Training Material and Code Samples
    https://developer.nvidia.com/cuda-education
- NVIDIA Developer Blog （一番詳細）
  https://devblogs.nvidia.com
  - Mark Harris氏の記事（下記は入門編）
    https://devblogs.nvidia.com/even-easier-introduction-cuda/

GPUプログラミングの情報源

- 後藤弘茂氏のニュース記事（幅広いネタ）
  - NVIDIAのマルチGPU戦略とインターコネクト帯域
    https://pc.watch.impress.co.jp/docs/column/kaigai/1122287.html
  - AMD、7nmで最大64コアの「ZEN2」とNVIDIA Voltaを上回る「Radeon Instinct M60」
    https://pc.watch.impress.co.jp/docs/column/kaigai/1151995.html
  - NVIDIAの巨大GPUを支えるTSMCのインタポーザ技術
    https://pc.watch.impress.co.jp/docs/column/kaigai/1064109.html

GPUプログラミングの情報源

- CUDA関連の教科書
  - CUDA C プロフェッショナル プログラミング
    https://www.amazon.co.jp/dp/B015R0M8TS/ref=dp-kindle-redirect?_encoding=UTF8&btkr=1
    - 詳しい．なんでも書いてある．初心者には詳しすぎるかも
  - ＧＰＵプログラミング入門－ＣＵＤＡ５による実装
    https://www.amazon.co.jp/dp/B00KCL6PFG/ref=dp-kindle-redirect?_encoding=UTF8&btkr=1
    - CUDAの説明はあっさり
    - 数値計算の例が多く載っているのでその分野の人はいいかも
  - はじめてのCUDAプログラミング
    https://www.amazon.co.jp/はじめてのCUDAプログラミング―驚異の開発環境-GPU-CUDA-を使いこなす-BOOKS/dp/4777514773
    - よくまとまっているが，内容が古い

GPUプログラミングの情報源

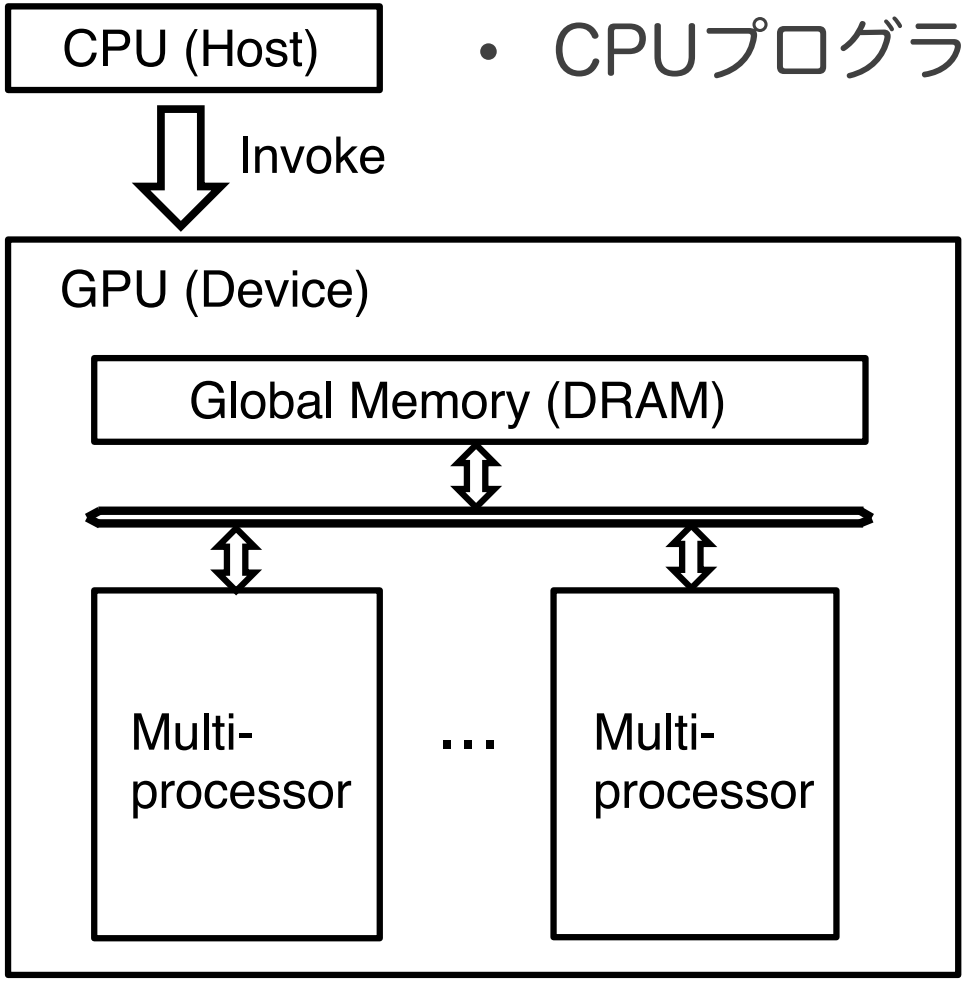
- アーキテクチャの教科書
  - 「ヘネ＆パタ」：ヘネシー＆パターソン コンピュータアーキテクチャ 定量的アプローチ第5版 4.4節
    https://www.shoeisha.co.jp/book/detail/9784798126234
  - 「パタ＆ヘネ」：コンピュータの構成と設計 第5版 ダウンロード教材 Appendix C
    http://ec.nikkeibp.co.jp/nsp/dl/09842/index.shtml
    - 無料で読めるが内容が古い

GPUの準備

- 汎用計算をするには，データセンター用GPUを使うのが一般的（画像用も使える）
  - 自分で買うと15万〜100万以上
- クラウド環境
  - 阪大サイバーメディアセンターOCTOPUS（本ワークショップ）
  - Amazon Web Services (AWS) EC2
    - P3インスタンス（旧世代でよければ，P2, G3）
    - 1時間 3USD〜（P2なら 0.9USD〜）

Amazon EC2 P3 インスタンス
https://aws.amazon.com/jp/ec2/instance-types/p3/

GPUアーキテクチャ（全体像）

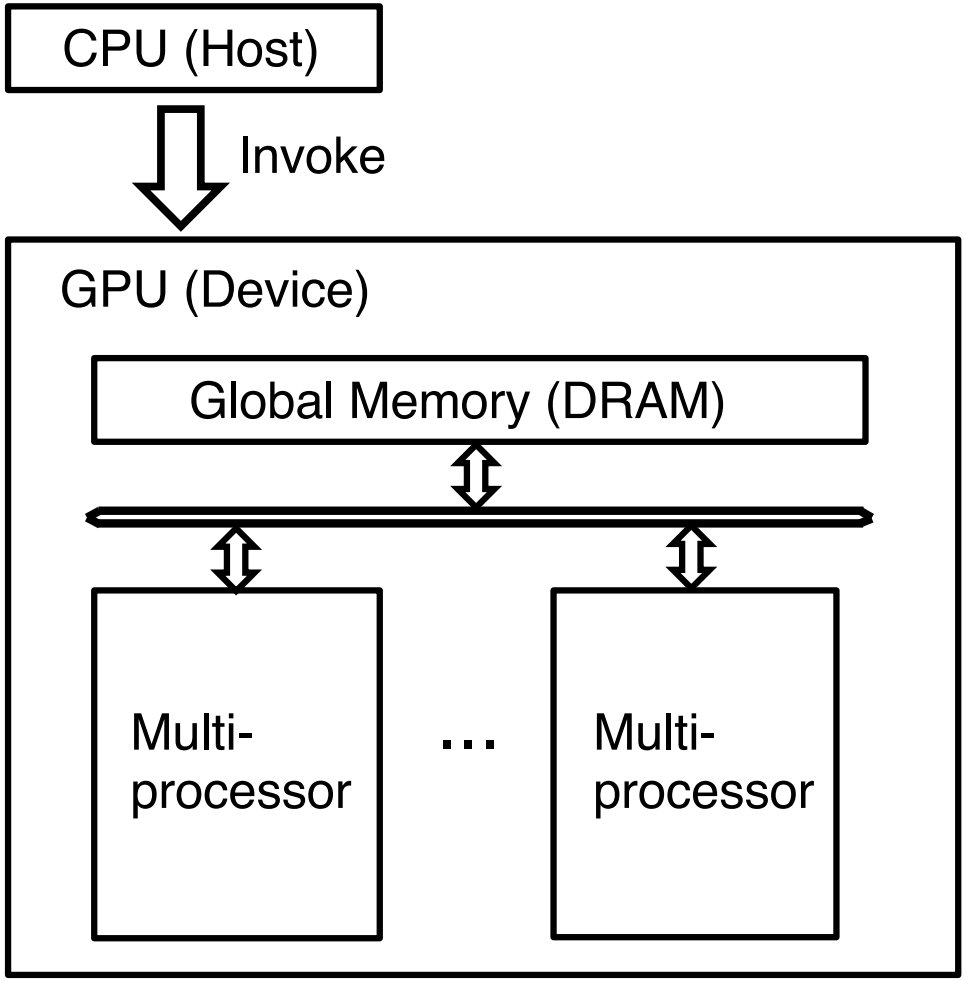


- CPUプログラムがGPUを制御する

- CPUプログラムは
  グローバルメモリとの
  データ転送ができる
- マルチプロセッサは
  個別に動作する
  - 非同期
  - 互いにデータ通信
    不可

※ 他のメモリは省略

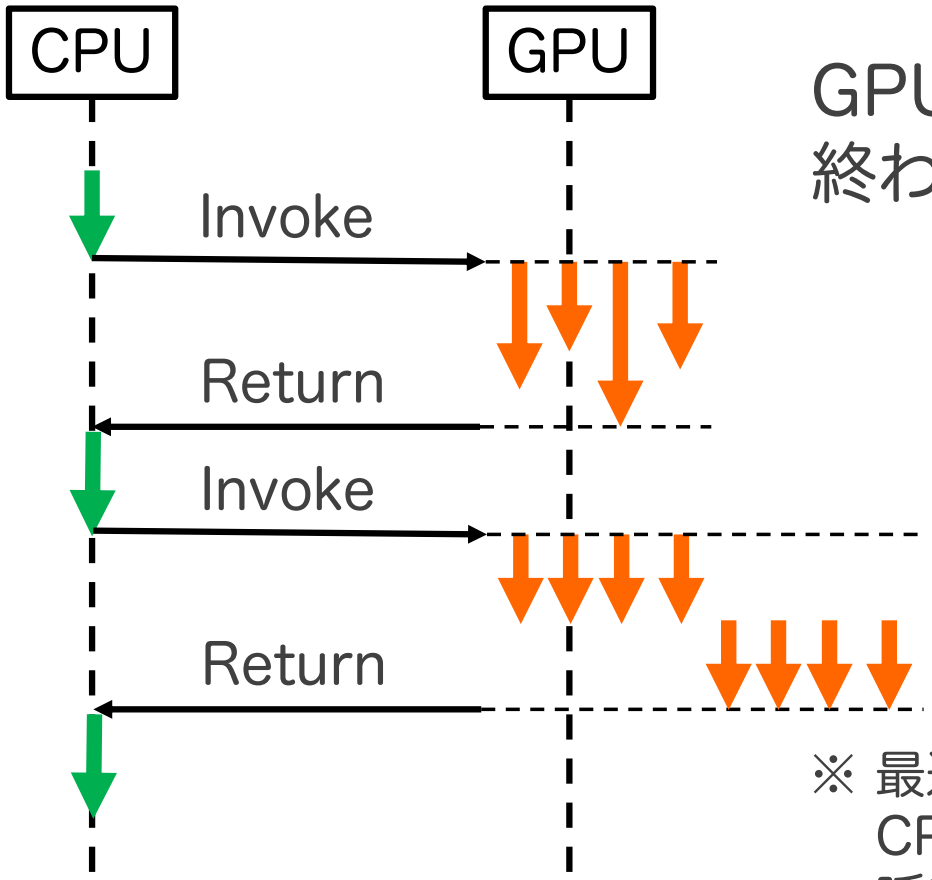
GPUプログラミングの処理概要



1. グローバルメモリに
   入力データ配置
2. GPU上でユーザ定義
   関数を実行
   （カーネル関数と呼ぶ）
3. グローバルメモリから
   出力データ取得
4. 必要に応じて
   1～3を繰り返し

※ 最近(Pascal以降)のGPUでは
CPU,GPU間のメモリ同期を
オート（呼び出し不要）にできる
（ユニファイドメモリ）

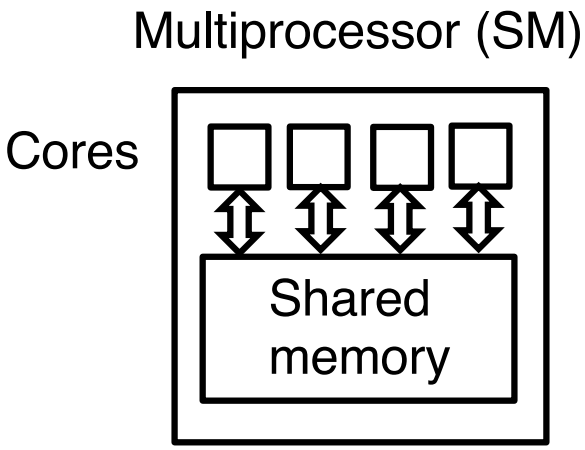
基本的な処理の流れ



GPU側のすべての処理が
終わるとCPUにリターン

※ 最近(Kepler以降)のGPUでは
CPUから複数のGPUカーネルを
呼び出したり，GPUからGPU
カーネルを呼び出したりできる

マルチプロセッサ（Streaming Multiprocessor; SM）



- 複数のコアと共有メモリからなる
  - 32～192コア/ SM
- コアは（基本的には）同期しながら処理を行う
  - 複数のコアが同一命令を実行
  - 一部のコアのみが命令実行することも可能

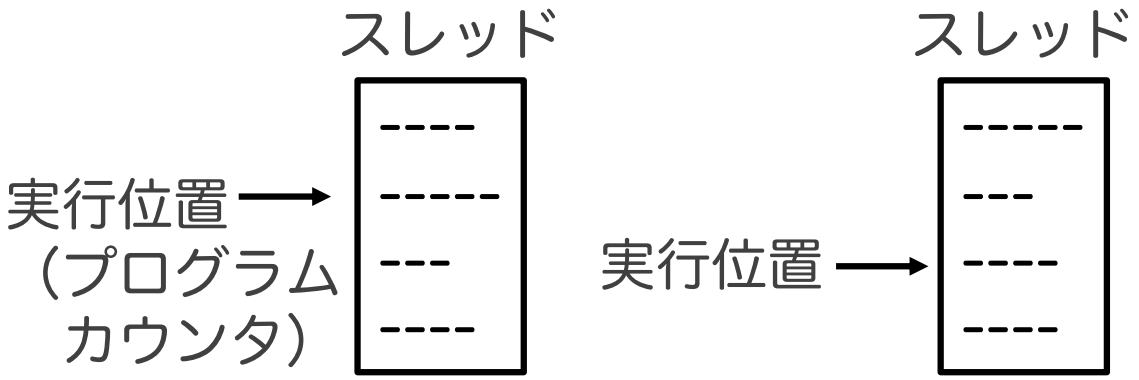
※ 最新GPU(Volta) では
より柔軟なコアの制御ができる

これから説明すること

- GPUアーキテクチャはなぜ優れているのか？
  - そのために並列/並行処理により処理が高速化される仕組みを説明する
- 並列/並行処理による高速化の仕組み（概要）
  - 処理を複数コアに割り当て，並列実行することで処理時間が短くなる
  - メモリアクセス時間が低減・隠蔽される

【用語】スレッド

- プログラム実行の単位
  - 1CPU上で複数スレッドを実行可能
  - 複数スレッドは代わりばんこにCPUを利用する
    ⇒ 通常はOSがスレッドのスケジューリングを行う
  - GPUではハードウェアがスレッド管理を行う



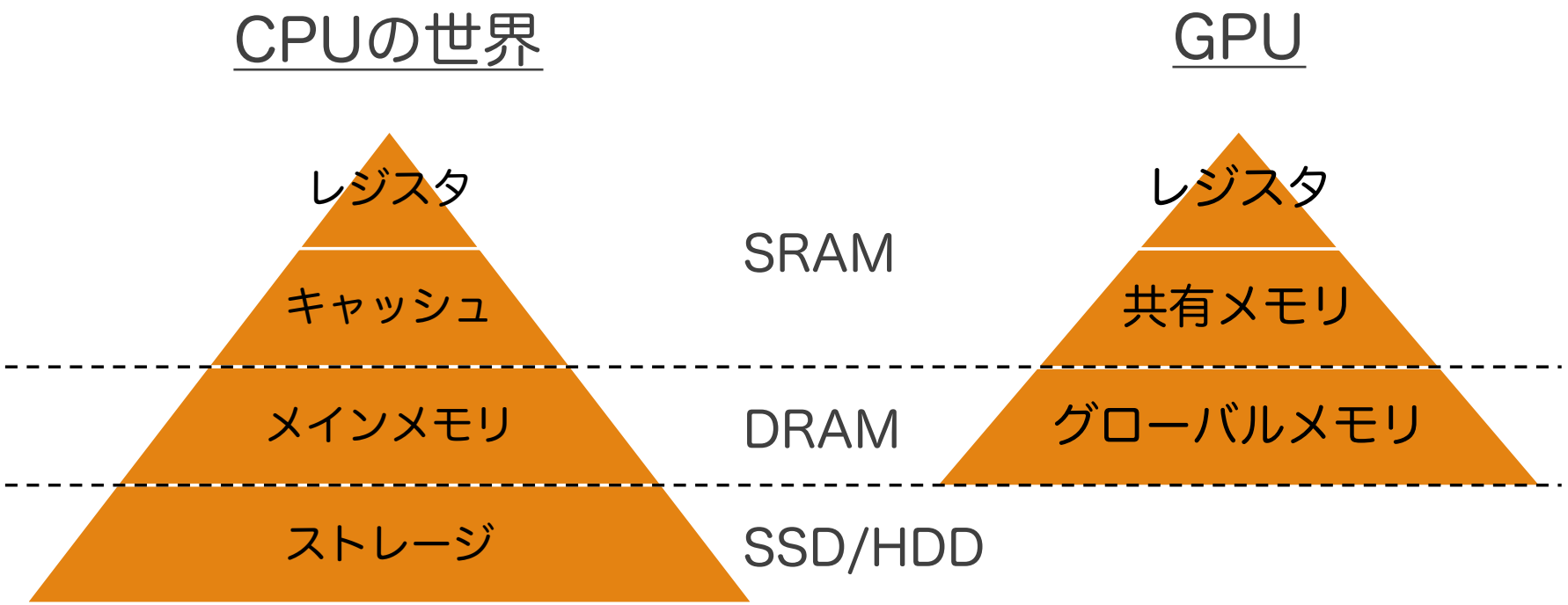
高速化の課題

- コアを増やせばその分速くなる
  というわけではない
- 速くならない主な要因
  - スレッド間の依存度が高い
    （他のスレッドの処理完了待ち時間が長い）
    ⇒ 主にソフトウェアの問題（二日目に扱う）
  - メモリアクセスに時間がかかる
    （多くのコアが一斉にアクセスして渋滞する）
    ⇒ アーキテクチャ+ソフトによる適切な活用が重要

メモリ

- メモリアクセスの効率化を議論する
  - メモリのハードウェアの特性を説明した後,
    各メモリへ効率的にアクセスする仕組みを説明する
- GPU
  - グローバルメモリ：DRAM
  - 共有メモリ：SRAM

メモリ階層



- 一般的には下の階層ほど，大容量だが遅くなる

DRAM

- キャパシタ（コンデンサ）で情報を記憶する
- 格子上にキャパシタを配置する
  - 先頭行から順にアドレスを配置し，埋まったら二行目

DRAM

- データ読み込み後，電荷の再注入（プリチャージ）が必要となる
- 何もしなくても一定時間毎に電荷の再注入（リフレッシュ）が必要となる
- これらを行単位で管理する

DRAMのデータ読み込み手順

1. 行のアクティベート
   - 信号線を指定の行に接続する
2. 指定列の転送
   - アクティベートした行の中から必要な列を転送する
3. プリチャージ
   - 他の行にアクセスする前に電荷のチャージを行う（ディアクティベート）

データ転送以外にも時間がかかる

DRAMへの効率的アクセス

- 一度，行にアクセスしたら，できるだけたくさんのデータを転送した方が有利
- DRAMはバーストリード（行の連続するデータの高速転送）に対応している

⇒ 連続するデータにアクセスした方が高速
⇒ CPUではキャッシュメモリが有効

ざっくりとした数字

- クロック周波数を1GHzとすると
  クロック周期は1ns
- データアクセスに必要な総時間は10〜
  100nsのオーダー
  - グローバルメモリアクセスのレイテンシは200〜
    400クロック（@CUDA Programming Guide）
- メモリバンド幅（最大データ転送レート）は
  672GB/sec（@Quadro RTX 6000）
  - 1nsあたり672Byte（4Byte変数168個分）

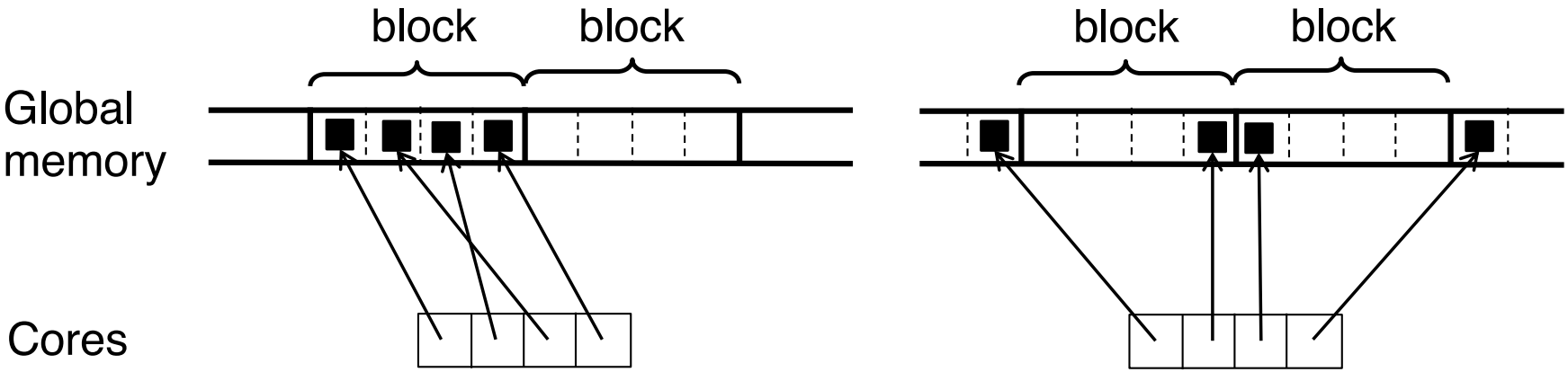
DRAMへの効率的なアクセス

1. キャッシュメモリ
   - 多くのプログラムにおいて，一度アクセスしたアドレスとその周辺は近いうちにアクセスされる傾向がある（時間的局所性，空間的局所性）
   - ある1ワードにアクセスした時，そのワードが含まれるブロック（行の一部）の全データをキャッシュにバースト転送する
   - GPUもキャッシュメモリを持っている

DRAMへの効率的なアクセス

2. コアレスアクセス

- 複数のコアが連続するデータにアクセスする時,
  DRAMのバーストリードにより高速読込できる



(a) コアレスアクセス

高速

(b)

低速

例：SIMDアーキテクチャ

- 代表的な並列アーキテクチャの一つ
  - GPUアーキテクチャはSIMDの進化版と言える
- 1命令で複数のデータを並列処理する
  - 全データに対して同一の演算を行う（Single Instruction Multiple Data）
  - 演算器をデータ数分用意しておく
- 連続データを処理するのでコアレスアクセスが可能
  - 各コアが独立に動作するアーキテクチャより効率的

DRAMへの効率的なアクセス

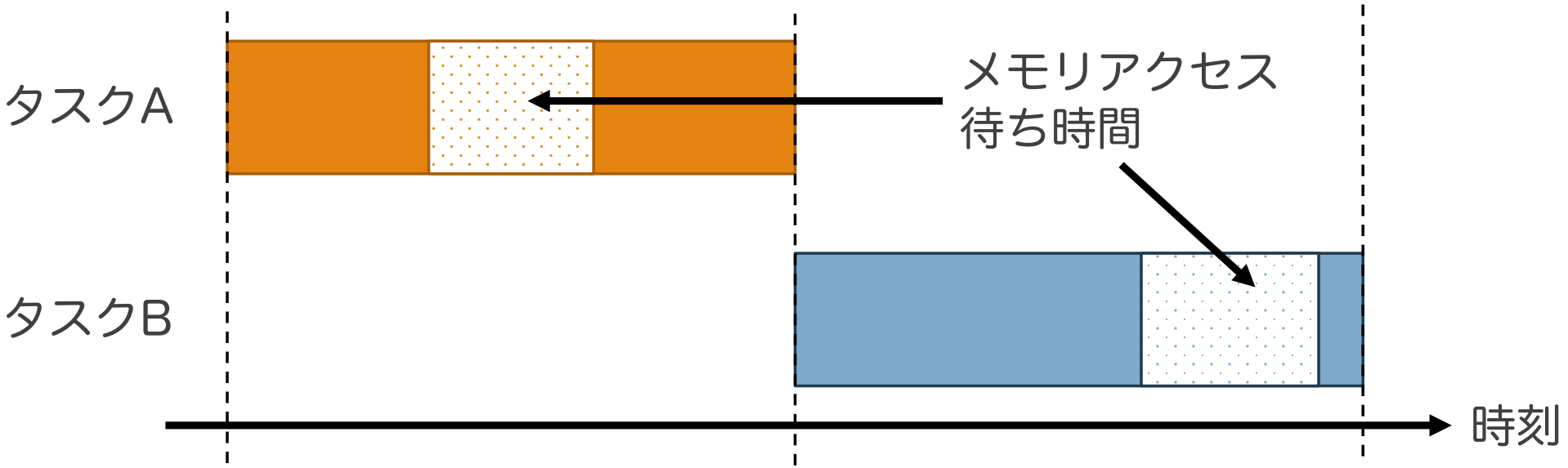
3. ハードウェアマルチスレッディング
   - メモリアクセスによる待ち時間（レイテンシ）を隠蔽できる

   - 隠蔽？
   - 隠蔽したら速くなるの？

   ⇒ 具体例で説明する

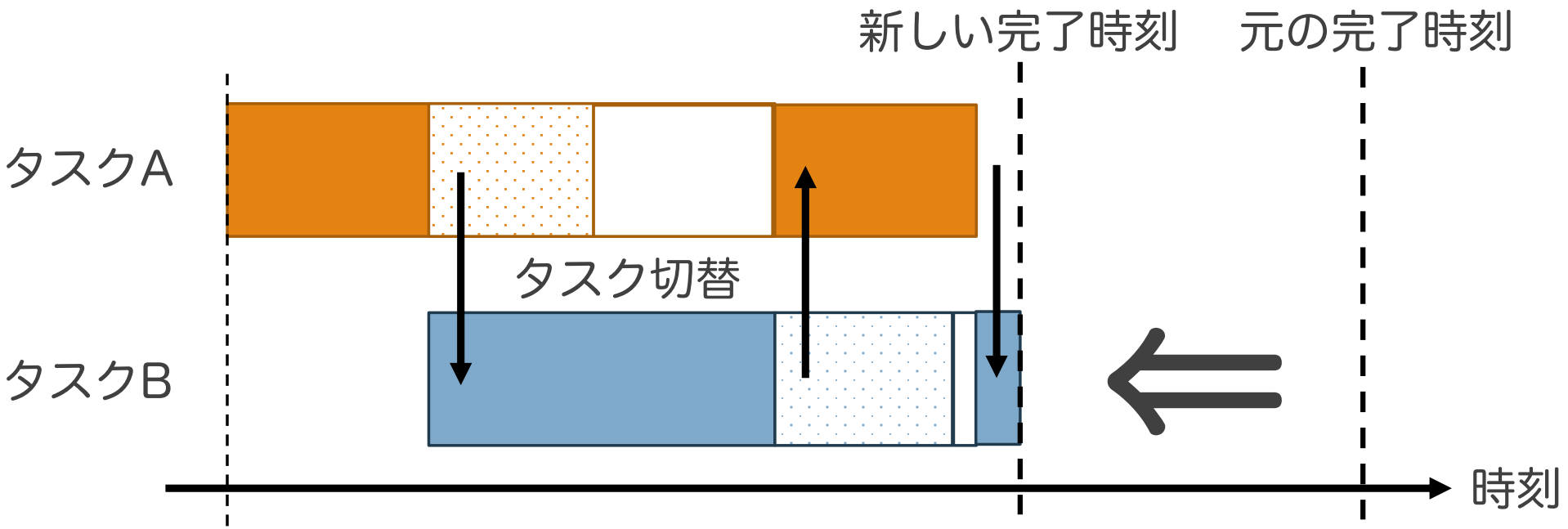
レイテンシ隠蔽による高速化

- 独立した二つのタスクA,Bを行うとする
- 1コアで，タスクAを終えた後にタスクBをやる場合，以下のようになったとする



レイテンシ隠蔽による高速化

- タスクA,Bをいつでも部分的に実行できるとすると・・・



処理完了時刻が速くなった
（レイテンシの影響を受けなくなった）

レイテンシ隠蔽の実装

1. 複数のタスクをマルチスレッド化する
   ⇒ ソフトウェア側で頑張る
2. スレッドを高速に切り替えられるようにする
   ⇒ ハードウェアがクロック単位で切り替える

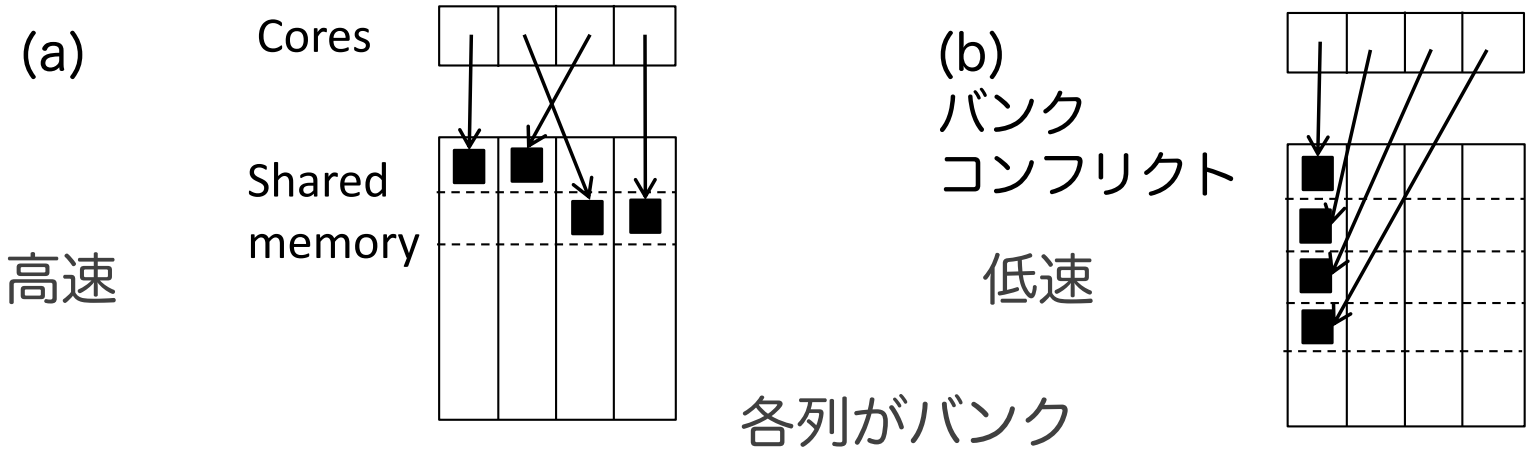
- **ハードウェアマルチスレッディング**
  - ハードウェア管理によるマルチスレッディング
  - スレッドの高速な切り替えを可能にする

SRAM

- 半導体（フリップフロップ回路）により情報を記憶する
- 高速アクセスが可能（1～数クロックで1ワード(通常64bit)を取得可能）
- 通常、複数のバンクで構成される
- マルチプロセッサの共有メモリがSRAM

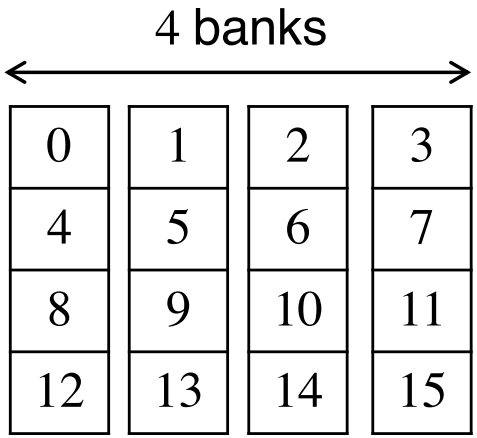
SRAM

- 各バンクに同時にアクセスできる
  - 複数のコアが別々のバンクにアクセスする時,
    同時に処理が可能
  - 逆に複数のコアが同一バンクにアクセスすると、
    待ちが発生する→バンクコンフリクト



SRAM上のアドレス配置

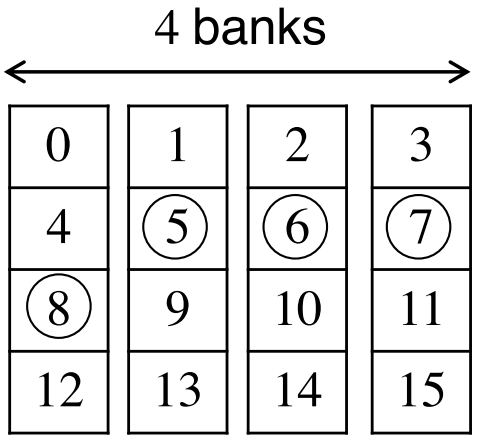
- 連続するアドレスが異なるバンクになるように配置される
  ⇒ 各コアが連続するワードにアクセスすることでバンクコンフリクトを回避できる



4バンクSRAMでのアドレス配置

例：SIMDとバンクコンフリクト

- 処理するデータを連続データにすることで
  バンクコンフリクトを回避できる



4並列SIMDでのバンクコンフリクト回避例
（◯がアクセスするワード）

メモリアクセスまとめ

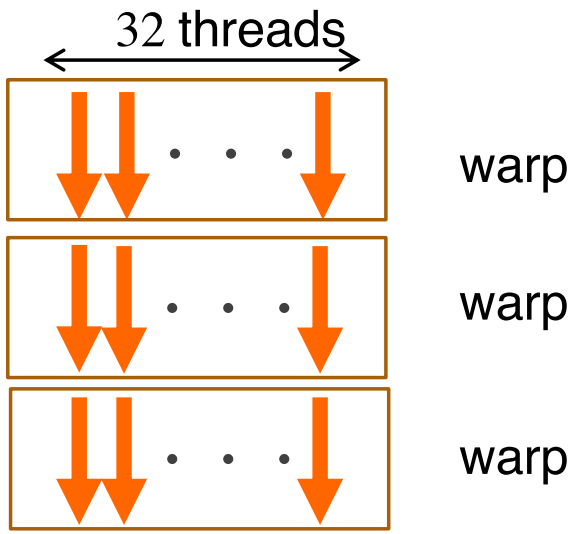
- DRAMもSRAMも連続データに同時アクセスすることで効率の良いデータ転送が行える
  - SIMDアーキテクチャによりこれを実現できる
- 効率的なDRAMへのアクセスの別のアプローチとして，ハードウェアマルチスレッディングがある
  - レイテンシ（待ち時間）を隠蔽できる

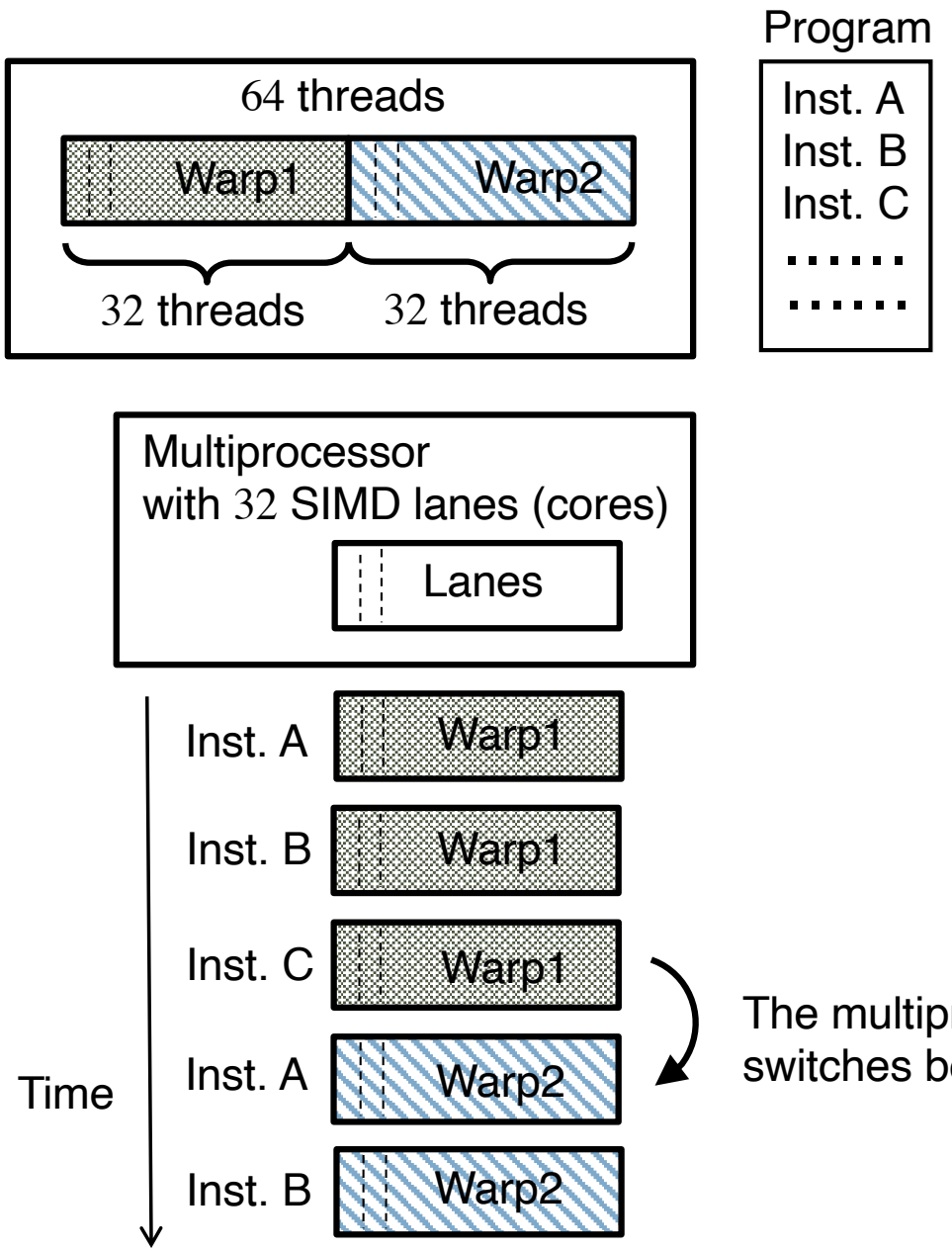
GPUマルチプロセッサの概要

- NVIDIA社はSIMTと呼んでいる
- 特徴
  - SIMDと同様の効率的なメモリアクセス
  - ハードウェアマルチスレッディングによる
    レイテンシの隠蔽
- 「ヘネ＆パタ」ではマルチスレッドSIMDプロセッサと呼ばれている

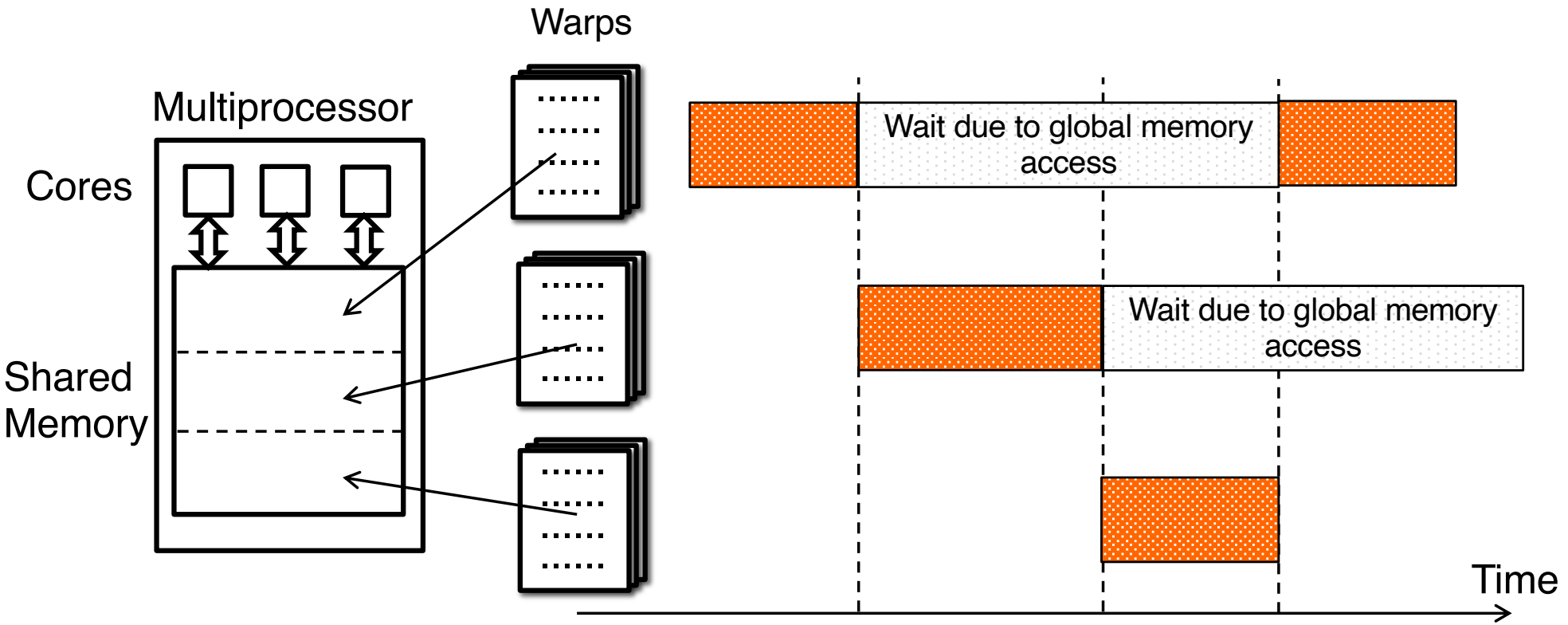
GPUマルチプロセッサの概要

- 各マルチプロセッサに大量のスレッドを割り当てる
- 全スレッドが異なるデータに対して同じ処理を実行する
- スレッドは32スレッドごとに1ワープとしてまとめられる
  - 32コアに1ワープ（32スレッド）を割り当て ⇒ SIMD命令





- 各ワープを交互に実行する必要はない
- 毎クロックで，マルチプロセッサはReadyのワープを選択して実行する
  ⇒ ハードウェア
  マルチスレッディング


Warps
Multiprocessor
Cores
Shared
Memory
Wait due to global memory
access
Wait due to global memory
access
Time

GPUマルチプロセッサまとめ

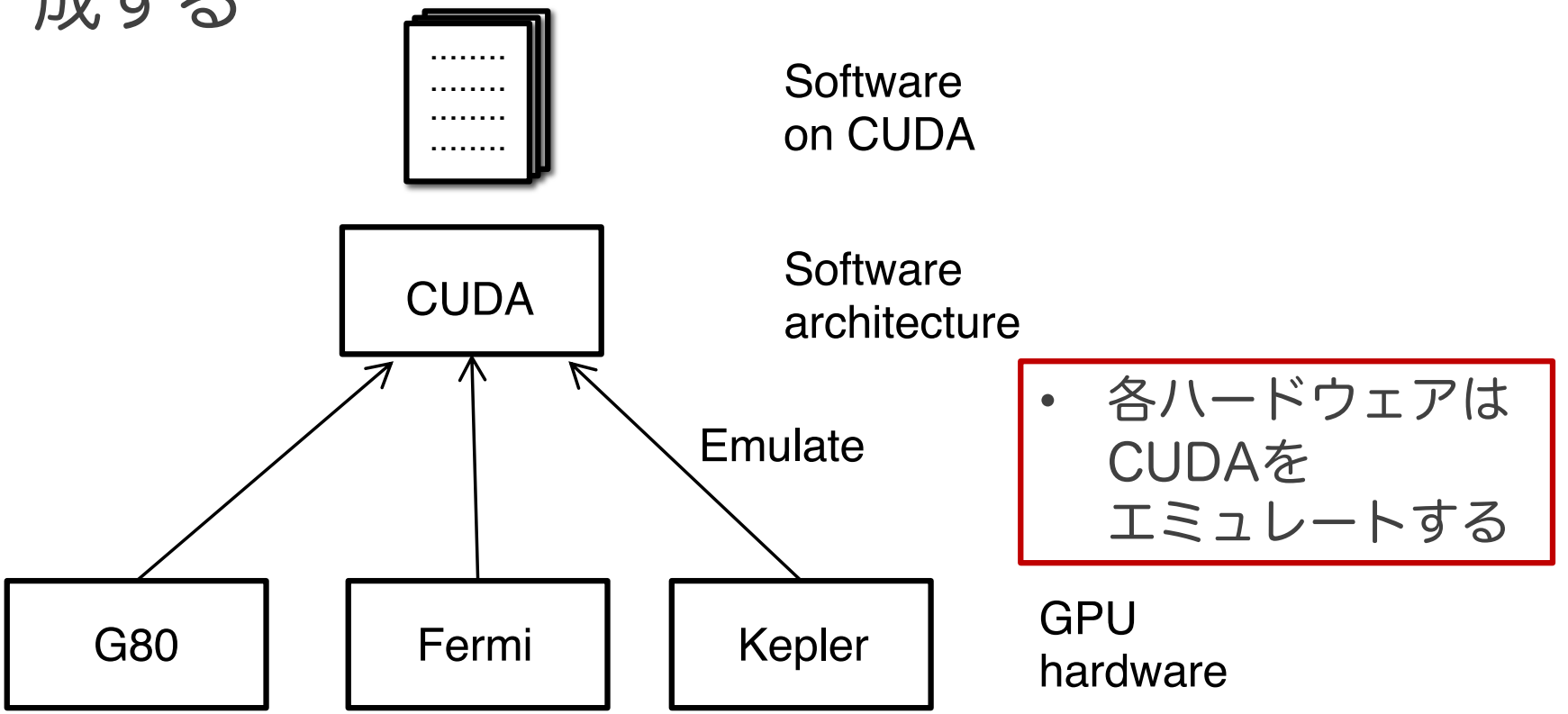
- スレッドをワープ単位で管理することで
  - SIMDメモリアクセス命令が実行できる
  - ハードウェアマルチスレッディングが行える
- プログラム時の注意点
  - コア数よりも多くのスレッドを割り当てないと
    ハードウェアマルチスレッディングが行えない
    - 1SMあたりのスレッド数は仕様上許される最大数にするのが良い
      （最大数に対するスレッド数の割合をオキュパンシと呼ぶ）
  - SIMDメモリアクセス命令においては，基本的に
    連続するデータを指定しないと効率的にならない

GPUプログラミング

- NVIDIA GPUは世代やモデルごとに仕様が大きく変わる
- すべてのGPU上で動作するようなプログラムを書きたい
  ⇒ CUDA (Compute Unified Device Architecture)
  - NVIDIA GPU向けの計算モデル，開発環境

CUDA

- ユーザはCUDA上で動作するプログラムを作成する

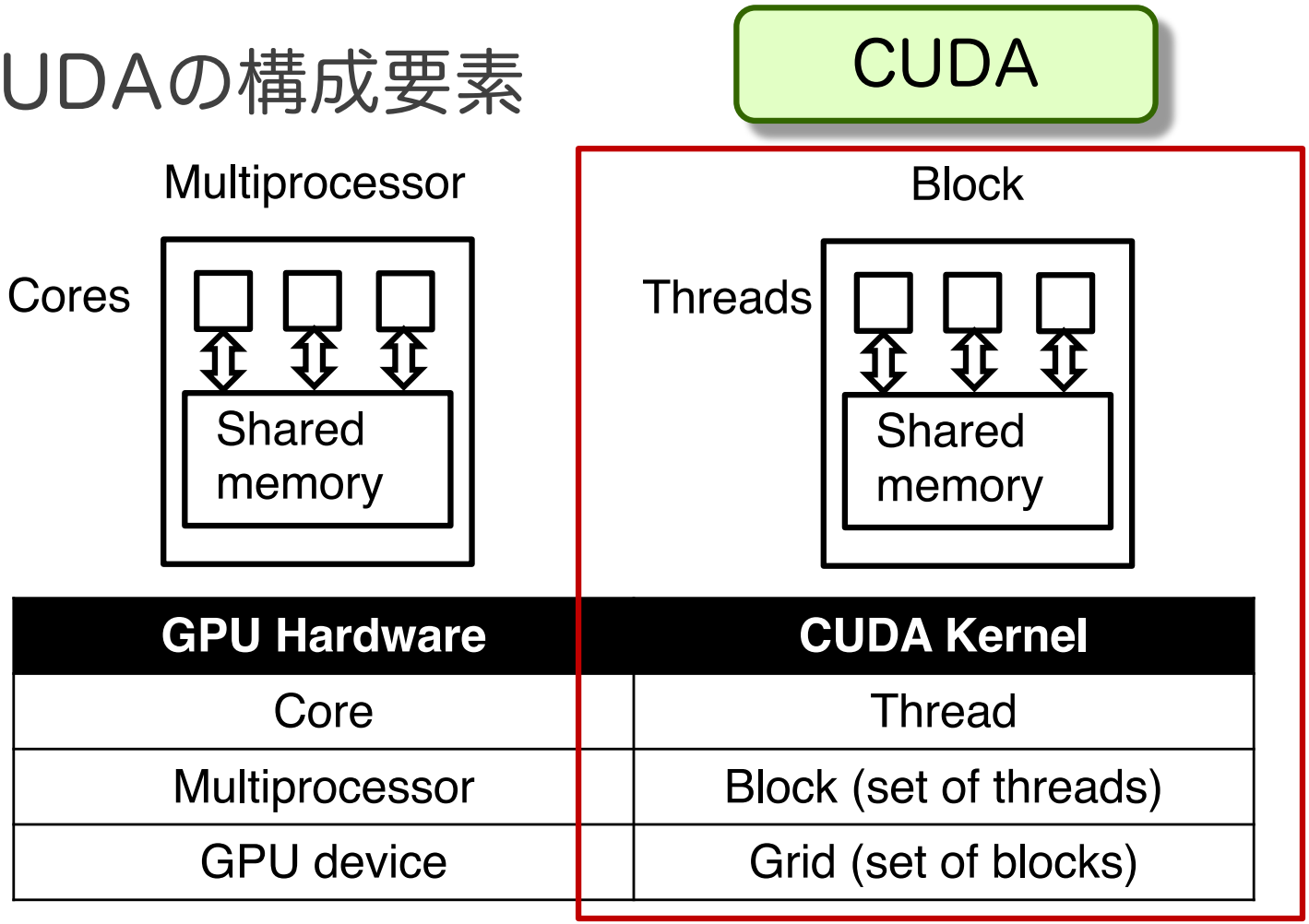


CUDA

- Compute Capability
  - CUDAのバージョン番号であり，この値ごとに対応する仕様（対応可能なGPU）が決まっている
  - Compute Capabilityを指定してコンパイルすることで，最新の機能を使用することができる

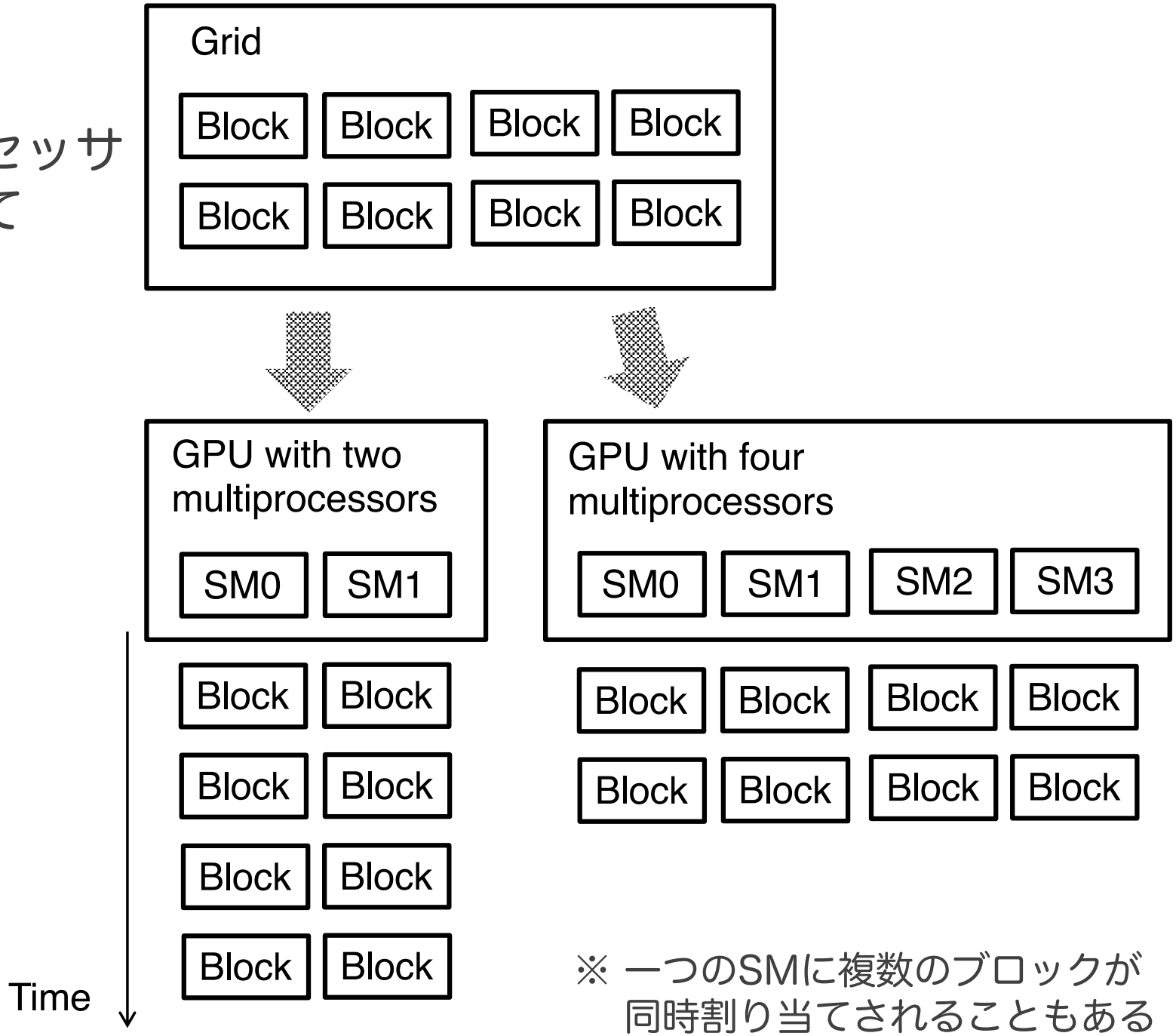
GPUデバイスとCUDA

- CUDAの構成要素



GPU Hardware	CUDA Kernel
Core	Thread
Multiprocessor	Block (set of threads)
GPU device	Grid (set of blocks)

ブロックのマルチプロセッサへの割り当て



※ 一つのSMに複数のブロックが同時割り当てされることもある

実習

- CUDAプログラムを実習環境で実行する
- 今日の目標
  - サンプルプログラムを実行できるようになる
- 二日目の目標
  - 基本的な文法を理解する
  - 高速化のノウハウを学ぶ

実習のための基本事項

- 大規模並列計算システムOCTOPUSを使う
- 詳細は大阪大学サイバーメディアセンターのマニュアルを参照のこと
  http://www.hpc.cmc.osaka-u.ac.jp/system/manual/octopus-use/
- 今回はコマンドベースでコード作成やプログラム実行を行う

実習0：サンプルファイルを実行する

- 本計算機環境ではファイル実行の仕方が通常とは異なる
  - ジョブスケジューラにジョブを投げる
- 本実習ではビルド済みの実行ファイルを実行する
  - deviceQueryと呼ばれるプログラムを実行する
  - CUDAにおいて，最初に実行するサンプルの一つ
  - 使用するGPUの仕様を出力するプログラム

実習0：サンプルファイルを実行する

- 手順１：deviceQueryをユーザ領域にコピーする
  - $ cd ~/
  - $ cp /octfs/apl/CUDA/cuda-9.0/samples/1_Utilities/deviceQuery/deviceQuery ~/
- 手順２：ジョブスケジューラにジョブを登録するためのスクリプトを作る
  - $ emacs deviceQuery.nqs

```
#!/bin/bash
#PBS -q LECTURE
#PBS –y 302
#PBS -l elapstim_req=00:00:10,cpunum_job=1,gpunum_job=1
cd $PBS_O_WORKDIR
./deviceQuery
```

↑ 上記を入力したらC-x C-sで保存，C-x C-cで終了

実習0：サンプルファイルを実行する

```
#!/bin/bash                ← bashを使う（シバン）
#PBS -q LECTURE            ← 講習会用の待ち行列に並ぶ
#PBS –y 302                ← 今日だけ設定する（普段は使用しない）
#PBS -l elapstim_req=00:00:10,cpunum_job=1,
gpunum_job=1               ← 10秒間CPU1ノードGPU1台を使用する
cd $PBS_O_WORKDIR
./deviceQuery              ↑ ジョブ投入時のディレクトリに移動する
```

↑ 実行ファイルdeviceQueryを実行する

実習0：サンプルファイルを実行する

- 手順３：スクリプトを実行しジョブを投入する
  - $ qsub deviceQuery.nqs
- 手順４：ジョブが終わるまで待つ
  （コマンドは後述）
- 実行結果を確認する
  - $ less deviceQuery.nqs.o(リクエスト番号) ⇒標準出力
  - $ less deviceQuery.nqs.e(リクエスト番号) ⇒標準エラー出力

⇒ 標準出力にGPU情報が記載されていれば成功

実習0：サンプルファイルを実行する

- 参考：ジョブリクエスト関連のコマンド
  - $ qstat　← リクエストの状態確認
  - $ sstat　← 実行中ジョブの状態確認
  - $ qdel 123456.oct　← リクエストの削除（適宜数字変更）
  - $ sstatall | less　← 投入されているすべてのリクエストの状態
- 詳細は大阪大学サイバーメディアセンターのマニュアルを参照のこと
  http://www.hpc.cmc.osaka-u.ac.jp/system/manual/octopus-use/scheduler/
- OCTOPUS利用状況（要ログイン）
  https://portal.hpc.cmc.osaka-u.ac.jp/secure/oct-smap/oct-smap.html