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』

後半：GPUプログラミング（実習）

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/

二日目の内容：GPUプログラミング（実習）

- Hello world
  - 環境設定
  - CUDAの基本的な文法
    - CUDA C++とCUDA Fortran等があるが，今回はCUDA C++ (C++に対するCUDA拡張)を扱う
- 高速アルゴリズムの設計
  - 基本的な考え方
  - 例：リダクション
- 高速化関連トピックス

本日使用するGPU

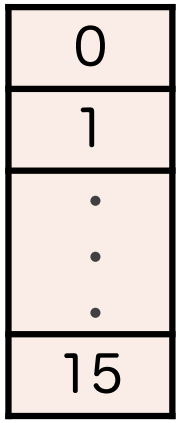
- NVIDIA Tesla P100
- Pascalアーキテクチャ
- Whitepaper
  https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
- 主要な仕様はdeviceQuery（前半）の結果に書かれている
  - $ less deviceQuery.nqs.o(リクエスト番号)

Tesla P100の主な仕様

- トータルコア数：3584
  - 56マルチプロセッサ × 64(コア/マルチプロセッサ)
- グローバルメモリサイズ：16GB
- マルチプロセッサあたりの
  共有メモリサイズ：64KB
- マルチプロセッサあたりの
  最大スレッド数:2048（最大ワープ数:64）
- ブロックあたりの最大スレッド数：1024

実習1. 最初のプログラム

- 問題
  - 16要素からなる整数配列のそれぞれの要素にindex番号（0～15）を書き込み表示しなさい



高速アルゴリズムの設計

- カーネル関数（GPU起動関数）を繰り返し呼ぶことで処理を行う
- 各カーネル関数では，以下の2つの階層を考えると良い
  - 全体: スレッド生成と各スレッドへのタスク振り分け
  - Local: ブロック内でのスレッドの処理

全体のアルゴリズム設計

- 気をつけること
  - スレッド数を十分に大きくする
    ⇒ スレッド数を多くしないと
    ハードウェアマルチスレッディングを活かせない
    ⇒ すべてのマルチプロセッサに最大数のスレッドを
    割り当てることを目指す
- オキュパンシ（占有率）
  - 割り当て可能なスレッド数に対する実際の割り当て
    スレッド数の比 ⇒ 100%にすることが望ましい

オキュパンシの増やし方

1. ブロック数を大きくする
   - 1つのマルチプロセッサに複数のブロックが割り当てられる
2. ブロック内のスレッド数を大きくする
   - 仕様上の最大値は1024
   - 32の倍数にすると効率的（∵1ワープ32スレッド）

⇒ どちらでも良いができるだけ2を活用した方が効率的なアルゴリズムを設計しやすい

ブロック

- ブロック内の全スレッドは必ず同一のマルチプロセッサで実行される
  - ブロック内のスレッドは共有メモリを用いてデータのやり取りをすることができる
- 複数のブロックが同時に一つのマルチプロセッサに割り当てられることもある
  - オキュパンシ計算時にはこのことも考慮する
  - ブロック間のデータ交換はできない

ワープ

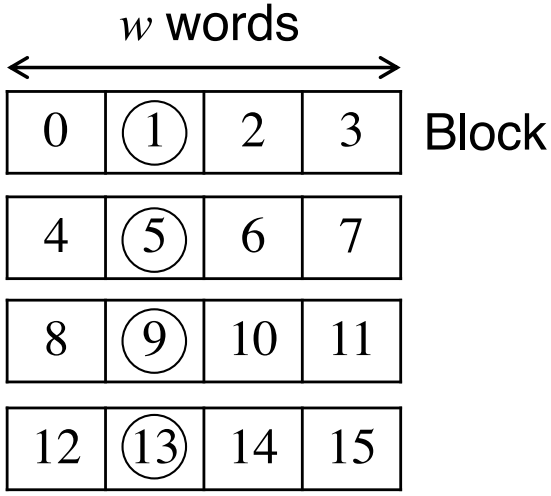
- ブロック内のスレッドは32スレッドごとに1ワープにまとめられる
- ワープ内のスレッドは基本的に，常に同じ命令を実行する（Voltaは別）
- マルチプロセッサのコア数が64の時，1クロックにつき2ワープをコアに割り当てる
  ⇒ 32 × 2 = 64
- ブロック内の各ワープは異なるタイミングで命令を実行する

オキュパンシを増やす際の障壁

- マルチプロセッサ内のスレッドは共有メモリとレジスタを共有する
  ⇒ スレッド数が多いと1スレッドあたりのメモリ割り当て量が小さくなる
  ⇒ メモリ使用量の大きい手法は使えない
- 一般に1スレッドに割り当てられるデータ量が大きいほど，アルゴリズムを設計しやすい（次の例参照）

よくあるトレードオフ（オキュパンシ vs メモリアクセス）

- 32 × 32 行列がグローバルメモリに格納されており，各列にアクセスしたい
  - 32個の各スレッドが1要素にアクセスするとコアレスアクセスではなくなってしまう
  - そこで一旦全要素を共有メモリにコピーすることにする
    ⇒コアレスアクセスが可能になる



4×4の例

よくあるトレードオフ（オキュパンシ vs メモリアクセス）(続き)

- メモリ使用量の解析
  - 各要素が4バイト変数とすると32×32行列のサイズは 32×32×4 = 4096B = 4KB
  - P100の共有メモリ：64KB
  - よって最大でも64/4=16ワープしか配置できない
  - P100の最大ワープ数/SM：64
  - オキュパンシは25%(=16/64)
- 扱う行列サイズを16×16にするとオキュパンシは100%になるが，メモリアクセスは多くなる

全体のアルゴリズム設計まとめ

- オキュパンシ100%を目指す
  - 作成するスレッド数（=ブロック数×ブロックあたりのスレッド数）をデバイスで同時割り当て可能なスレッド数以上にする
    - 例：P100はSM数：56, SMあたりの最大ワープ数：64
      ⇒ 作成ワープ数を3584 (=56*64)以上にする
  - ただし，オキュパンシはローカルで使用するアルゴリズムの共有メモリ使用量（とレジスタ使用量）に依存する

Localのアルゴリズム設計

- ブロック内のスレッドの処理の設計法を述べる
- 32スレッドごとに1ワープになっていることに注意する
- ブロック内のワープは同一の命令列を実行するがタイミングはバラバラである
- 全ワープを同期する命令も用意されている
  - __syncthreads()
  - 遅いワープが追いつくのを待つ

Localのアルゴリズム設計

- ワープ内のスレッドはSIMD的に動作することに注意する（Voltaは別）
  - グローバルメモリへのコアレスアクセス
  - 共有メモリへのバンクコンフリクトを避ける
  - 条件文では，非該当のコアは待ち状態になる
- オキュパンシを大きくするために
  共有メモリ使用量を極力小さくする

アルゴリズム設計の具体例：リダクション

- リダクションとは？
  - 総和の一般化
- $\sum_{i=0}^{3} a_i = a_0 + a_1 + a_2 + a_3$
- $a_i \uparrow a_j \stackrel{\text{def}}{=} \max(a_i, a_j)$と定義すると
  $\max(a_0, a_1, a_2, a_3) = a_0 \uparrow a_1 \uparrow a_2 \uparrow a_3$
- ここで，$a_0 \oplus a_1 \oplus a_2 \oplus a_3$を考えると，
  $\oplus$を+にすれば総和，↑にすればmaxになる

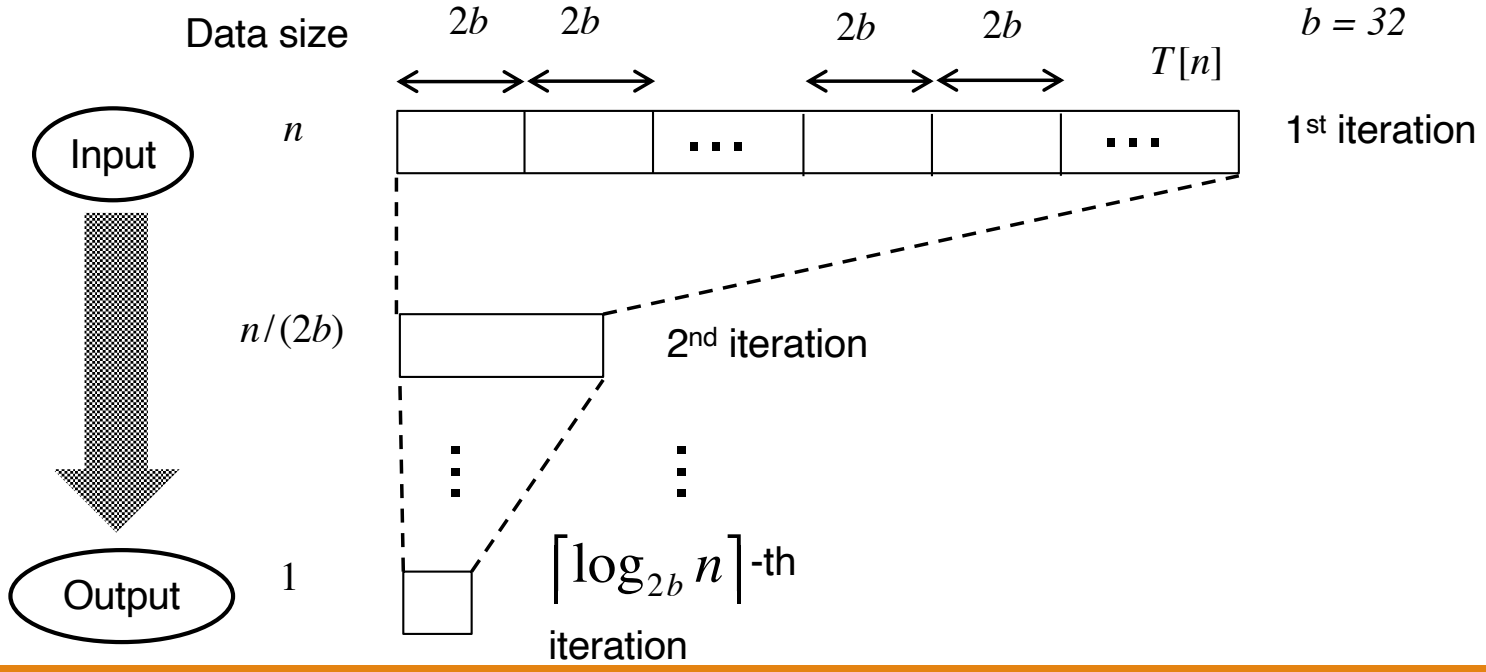
アルゴリズム設計の具体例：総和計算

- 問題
  - n要素からなる配列の総和を求めよ
  - これから2つのやり方を紹介する
    - Tree-basedアルゴリズム
    - Cascadingアルゴリズム
  - 今回は配列の総和を考えるが，
    max, minなども同じ方法で計算できる

Optimizing Parallel Reduction in CUDA, Mark Harris
https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Tree-basedアルゴリズム

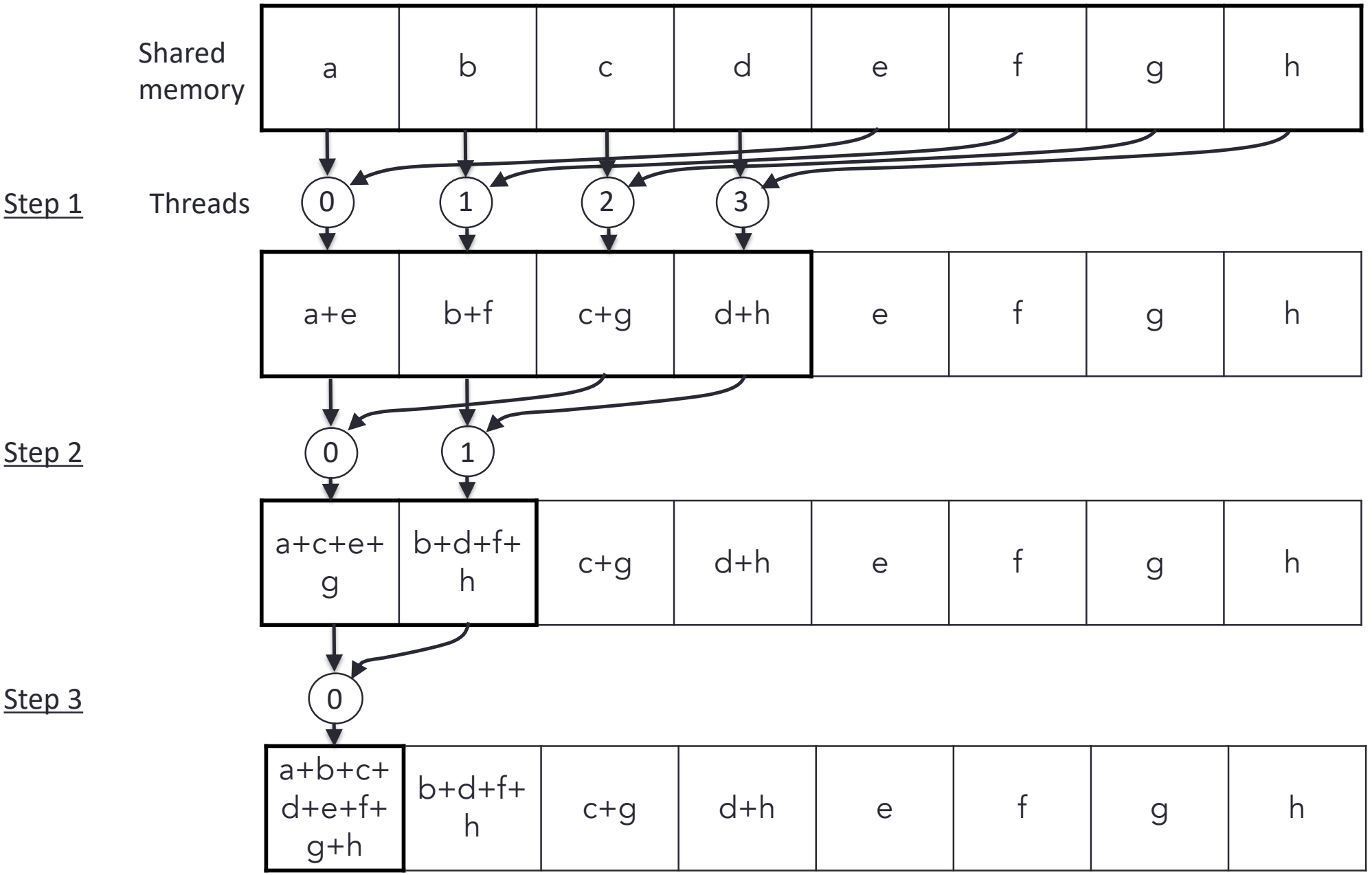
- 全体のアルゴリズム設計
  - 各ワープに64要素を割り当て1要素にリダクション
  - 要素数が1になるまで繰り返す



Tree-basedアルゴリズム

- ローカルのアルゴリズム設計
  - 1ステップごとに要素を半分にする
  - 要素数が1になるまで繰り返す

8要素の例


Shared memory
a
b
c
d
e
f
g
h
Step 1
Threads
0
1
2
3
a+e
b+f
c+g
d+h
e
f
g
h
Step 2
0
1
a+c+e+g
b+d+f+h
c+g
d+h
e
f
g
h
Step 3
0
a+b+c+d+e+f+g+h
b+d+f+h
c+g
d+h
e
f
g
h

Tree-basedアルゴリズム：ローカルのアルゴリズム

- 共有メモリ使用量の解析
  - 1ブロックあたり64変数分の共有メモリ使用.
    1変数4バイトとすると256Byte使用
    ⇒ 32ブロック（64ワープ）でも8KByte
    ⇒ オキュパンシに影響を与えない
- 計算量
  - スレッド数をbとするとO(log b)
  - 待ち状態のコア（スレッド）が多いので非効率

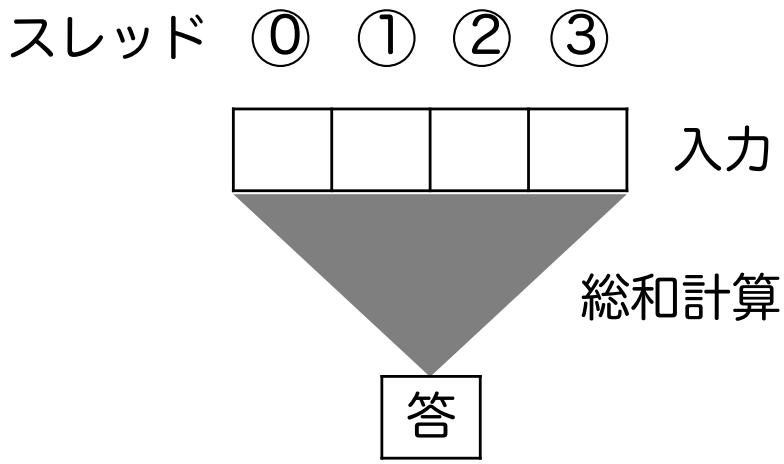
Cascadingアルゴリズム

- Tree-basedアルゴリズムでは,
  ローカルリダクションが非効率だった
  ⇒ 待ち状態のコアが多くもったいない
- 各ワープへの割り当てデータ量を増やすことで効率的にすることができる
  ⇒ Cascadingアルゴリズム

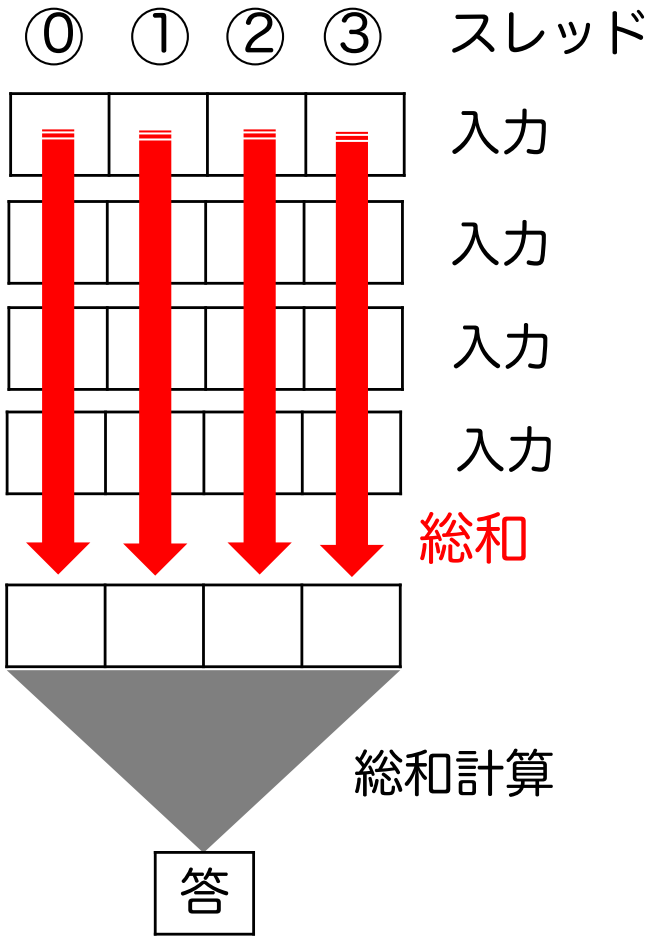
Cascadingアルゴリズム：ローカルのアルゴリズム

1ワープ4スレッドとした場合の例

Tree-based

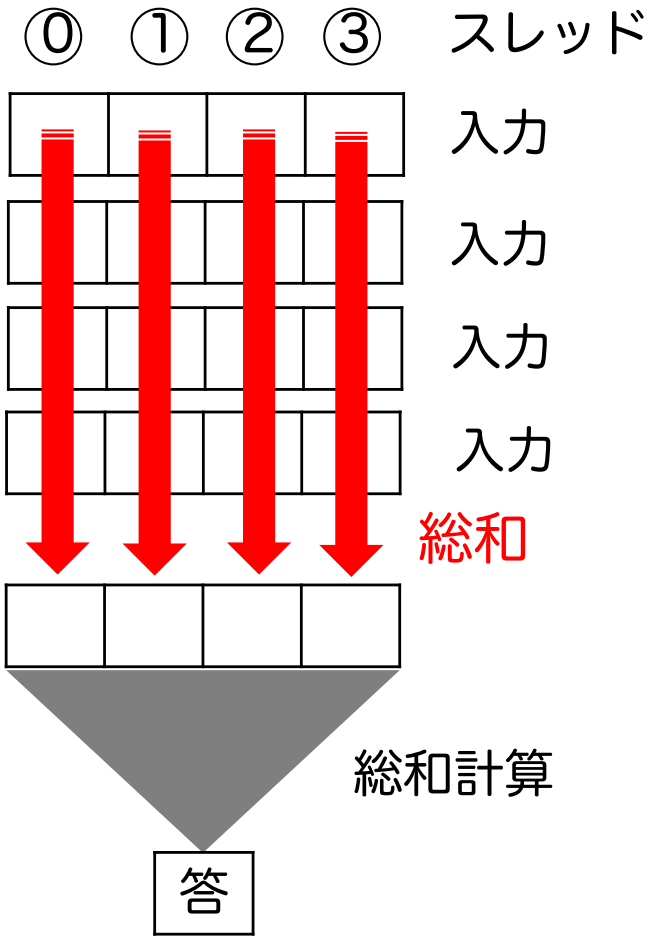


Cascading



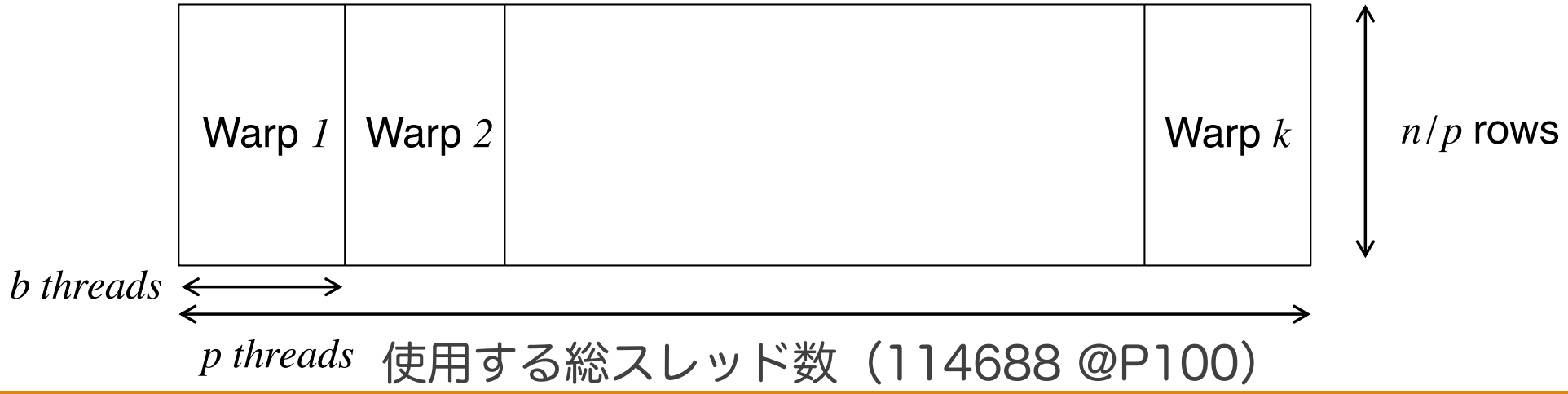
Cascadingアルゴリズム：ローカルのアルゴリズム

- 共有メモリ使用量の解析
  - 共有メモリは各列の結果の格納のみに使用
    ⇒ オキュパンシに影響を与えない
- 計算量
  - 読み込む入力ブロック数が多ければ（log b 以上なら(b:スレッド数)）最後のTree-based部分の計算時間は無視できる
  - コアがほぼフル稼働する



Cascadingアルゴリズム：全体のアルゴリズム

- 先に使用するワープ数kを決める
  - デバイスへの最大割り当て可能数にする(3584@P100)
- 各ワープに均等にデータを割り振る
- 各ワープの処理後，残ったk個をリダクション



アルゴリズム設計まとめ

- アルゴリズム設計の際は以下に注意する
  - オキュパンシ100%を目指す
    （P100では総ワープ数を3584以上，ワープあたりの共有メモリ使用量を1KB以下にする，レジスタの使用量も要注意だが今回は省略）
  - グローバルメモリへのコアレスアクセス
  - 共有メモリのバンクコンフリクト回避（今回省略）
  - 待ち状態のコアを少なくする

アルゴリズム設計まとめ

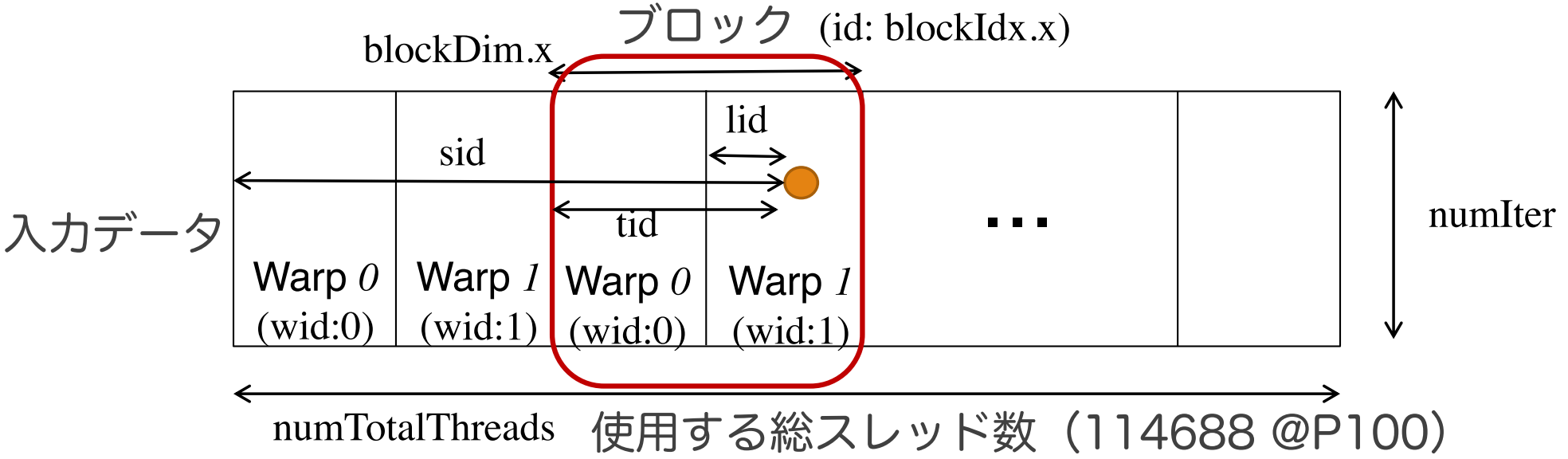
- 計算量の解析
  - KoikeらのGPU計算モデル（AGPUモデル）では，コア数pとマルチプロセッサ内のコア数bを用いて，以下の3つを計算し，アルゴリズムを評価する
    - マルチプロセッサごとのSIMD命令回数
    - グローバルメモリへのアクセス回数
      （コアレスアクセスでない時はその分多くする）
    - 共有メモリ使用量

実習2. 効率良く総和計算を行うプログラム

- 問題
  - 2^{25}要素からなる配列にはランダムに0または1が書き込まれている．この時，この配列の総和を求めなさい

実習2. 効率良く総和計算を行うプログラム

- Cascadingアルゴリズムの実装

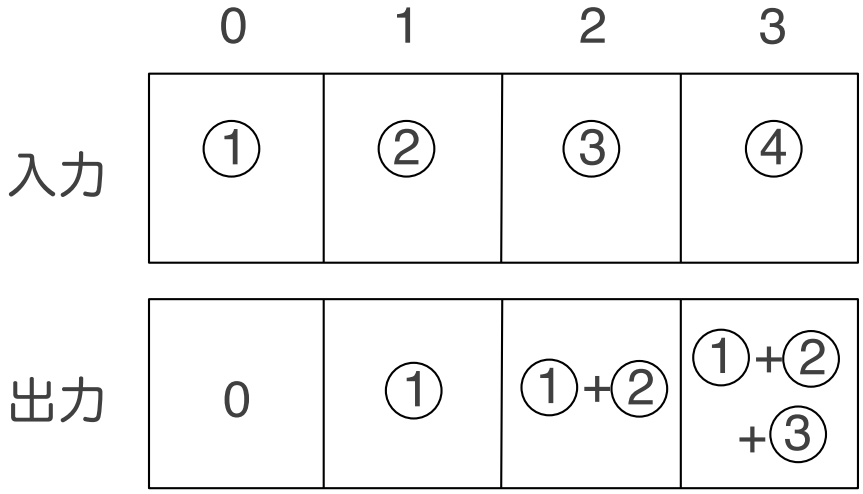


高速化トピックス

- いくつかの観点ごとに高速化手法を列挙する
  - 設計とデータ格納法
    - データに依存性がある場合
    - 構造体の配列とコアレスアクセス
    - 行列データの共有メモリへの配置方法
  - ライブラリの活用
  - 複数GPUの活用
  - GPUハードウェア(CUDA機能)の活用

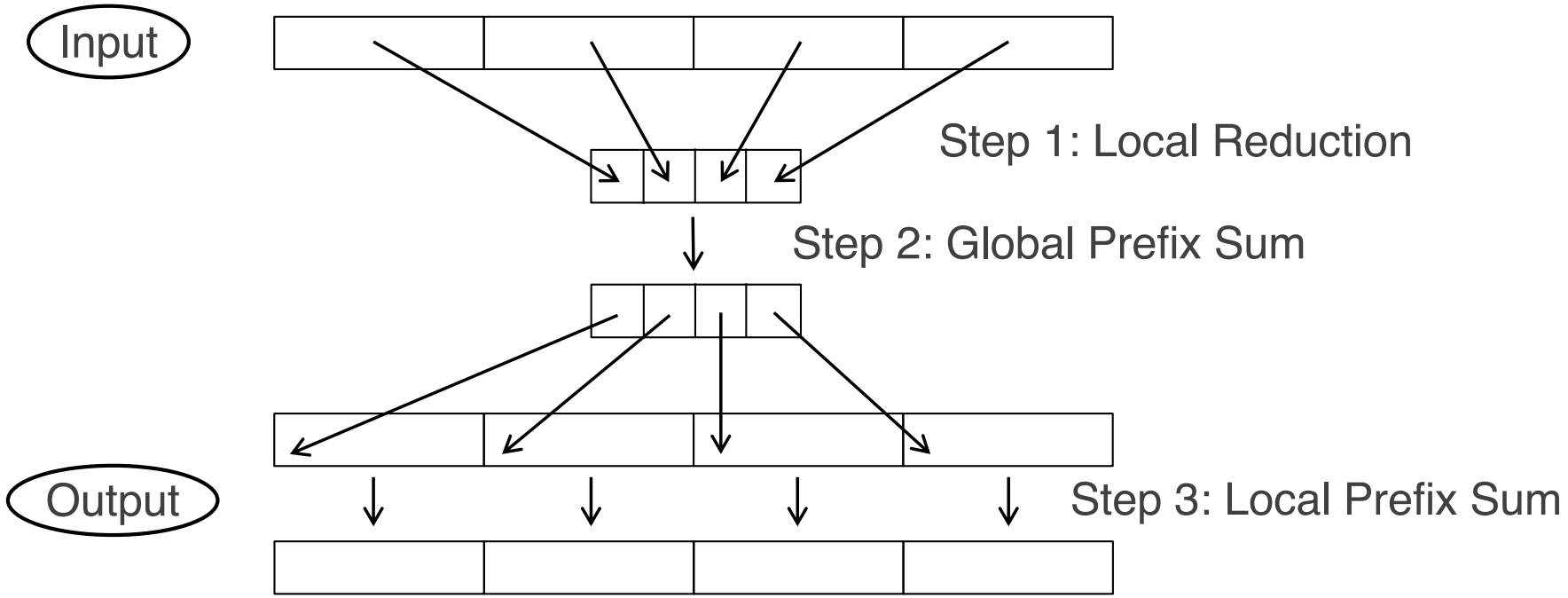
データ依存性がある場合

- データ依存性があると並列化は難しいが,
  単純な依存性であれば解決できることもある
- 例1：Prefix sum



一見前から順に計算するしか
ないようにも見えるが・・・

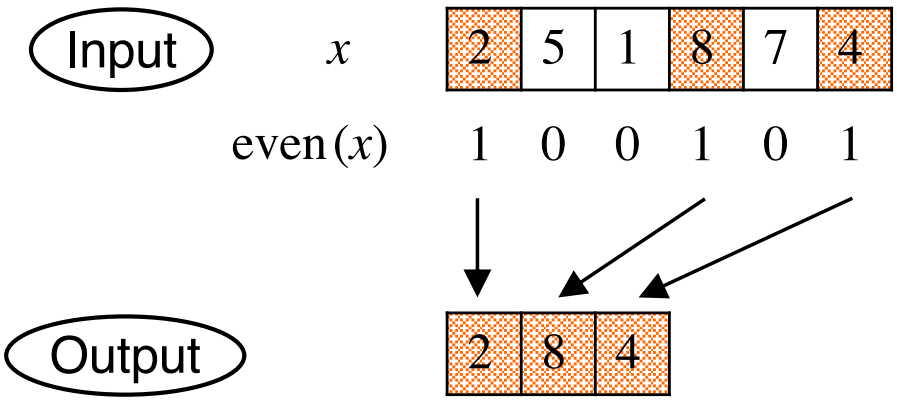
Prefix sumの並列アルゴリズム



先に各ブロックの先頭の値を決めてしまえば
あとは，並列に計算できる

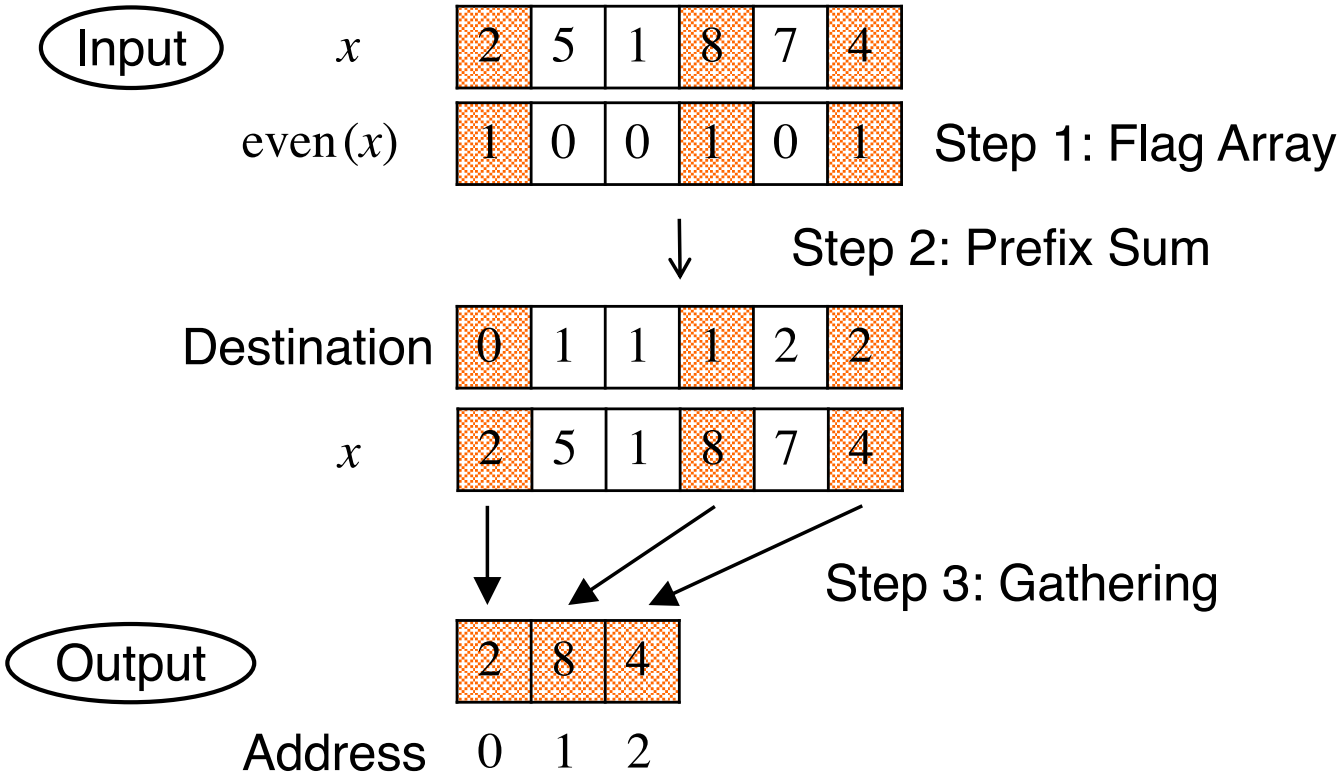
データ依存性がある場合

- 例2: 条件に合うデータをフィルタリング
  - 配列から偶数だけを取り出すとする
    ⇒データの書き込み先が他のデータに依存する



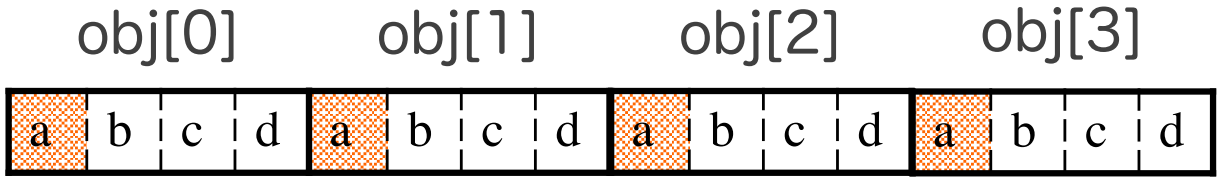
データフィルタリングのアルゴリズム

- 先にデータ書き込み先を求める(Prefix sum)



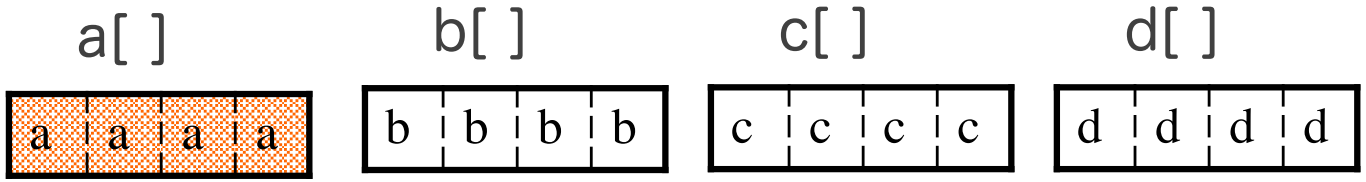
構造体の配列とコアレスアクセス

- 構造体の配列にアクセスしようとするとコアレスアクセスが難しい



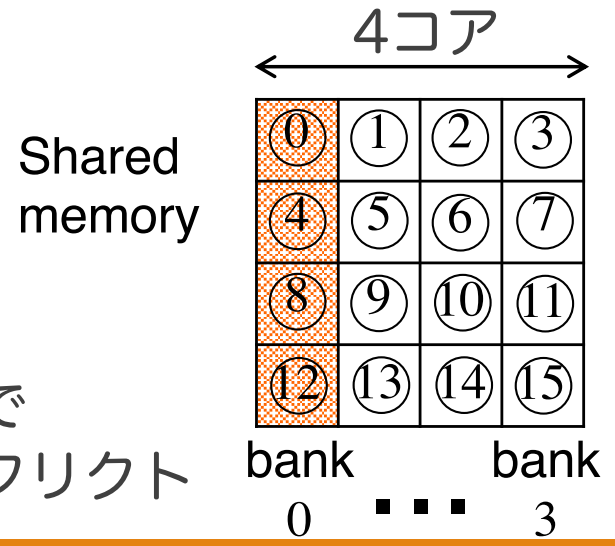
各コアがメンバーaにアクセスするとコアレスアクセスでなくなる

⇒ メンバーごとに配列を作るとコアレスアクセス可能になる

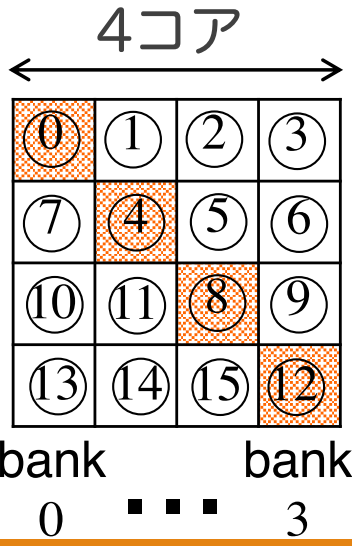


行列の共有メモリへの格納

- 前述の通り，オキュパンシを考慮する
- 格納位置についても注意が必要
  - 行アクセス，列アクセスでバンクコンフリクトが起こらないようにする（(b)のようにする）



(a)
列アクセスで
バンクコンフリクト



(b)
列アクセスでも
バンクコンフリクト
が発生しない

ライブラリの活用

- CUDAでは総和計算ですら効率的なコードを書くのは一苦労
  ⇒ 極力ライブラリに頼るのが良い
- リダクション，ソートなど：Thrust
- FFT：cuFFT
- ライブラリの使い方など：
  https://www.slideshare.net/NVIDIAJapan/1072-cuda

複数GPUの活用

- 1ノードに複数のGPUを設置して高速化
- 以前はGPU間の通信にはCPUを介す必要があり低速だった
- 現在はGPU間が直接通信できるため高速（NVLink）

GPUハードウェア（CUDA機能）の活用

- GPUハードウェア
  - コンスタントメモリとテクスチャメモリ
    - リードオンリメモリを活用してアクセスを高速化する
  - ユニファイドメモリ
    - ホストとデバイスのデータ同期を自動化
    - 必要なデータのみが同期されることで効率化できる
  - テンソルコア
    - 行列計算をハードウェアで行うことで高速化する

GPUハードウェア（CUDA機能）の活用

- CUDA機能
  - マルチストリーム
    - GPUに異なる複数のタスクを行わせたり，GPU処理中にCPUで別の処理を行ったりできる
  - ダイナミックパラレリズム
    - GPUから別のGPUカーネル関数を呼ぶことができる
    - CPUを介さずに動的にタスクを追加できる
  - Warpシャッフル
    - 他のスレッドに割り当てられたレジスタを読むことができる

GPUハードウェア（CUDA機能）の活用

- CUDA機能
  - 独立型スレッドスケジューリング（Volta）
    - 命令をワープ内のスレッドごとに異なるタイミングで実行できる