先進計算機構成論 11

東京大学大学院 情報理工学系研究科 創造情報学専攻
塩谷 亮太
shioya@ci.i.u-tokyo.ac.jp

質問や感想

- コメントなしだと評価が下がるということを知らずに，出席送信する際にコメントなしで出席送信していることが多々ありました．．．次から気をつけます
  - ちょっとその辺説明があいまいだったので，ほぼ差は付けないと思う

質問や感想

- 本日，ハードウェア例外の話がありましたが，meltdown attackでは，この例外処理の仕組みを悪用しており，例外発生をさせてそのアドレス先に処理を飛ばせてから，例外ハンドラが実行されている間にout of orderで目的のコードを実行させてキャッシュに残す，という理解であっていますか？

質問や感想

- 古い組込みCPUだと割り込みはあっても例外はなかったように思うのですが、例外は比較的最近のものでしょうか?

質問や感想

- メモリアクセスの例外を避けるために配列のインデックスがサイズを超えないようにプログラムを書きますが、実際にはOSが何KBかのブロックごとにメモリを渡すので、そのブロック境界までは安全にアクセスできるらしいですね。

- その性質を使って、メモリアクセスがちょっとはみ出すのを許してSIMD命令を行うと高速になるというのを聞いたことがあります。

質問や感想

- LSQの順序違反のあたりがよくわからなかったです。

質問や感想

- 投機的なロード・ストアの発行のように, 現在のCPUで実際に使われているようなものが教科書に書かれていないのはなぜなのでしょうか. また, そういった最新の技術に関して勉強するには論文を追うのがやはり一番よいのでしょうか.

質問や感想

- 発行キューやRe-order buffer、LSQは、レジスタ(またはキャッシュやメモリ)をバッファとして使用しているのでしょうか？

質問や感想

- alpha21264の数字に意味はありますか

質問や感想

- 分岐予測も含めて実行順序をout-of-orderにするために、分岐予測器や発行キュー、ROB、LSQなど様々な付属回路が必要になっている気がしますが、これらの大量の付属回路を含めた上でもやはりout-of-orderに実行したほうが圧倒的にメリットがある、ということでしょうか？

質問や感想

- 周辺回路が増えてきて、どのタイミングでどの回路が動作しているかのイメージが掴みにくくなってきたのですが、初心者でも使える/webで動くようなCPUシミュレータなどあったりしますでしょうか…？

- アーキテクチャシミュレータGem5を使ってみる
  - https://qiita.com/kaityo256/items/00fc50221d86ce3ff2ea
  - https://qiita.com/kaityo256/items/a0b79c7601c2cd10c2eb

質問や感想

- out-of-order 実行でも 最終的に コミットパイプで in-orderでやる，ロードストア実行の時のプログラムの正しさを保証するなどいろんな工夫が必要であることが分かりました．こういう追加的な手続き，動作などで，in-order実行よりも逆に遅くなるケースなどはありますか？

  - ◇ 予測ミス時からの状態の巻き戻しに時間がかかることがある
  - ◇ 予測ミスが極めて多いと逆転することがある

質問や感想

- 投機的方法のデメリットは？

質問や感想

- LSQの仕組みは大雑把にわかったのですが、どのようにして追い出しを行うのかが分かりませんでした。

- また聞き逃したかもしれないですが、エントリの並び替えはどうやって行うのでしょうか(アセンブリ時の命令列順だと例えば前にジャンプしたがフラッシュすることになった時に比較演算で対応できないですし、カウンタを使うにしても有限なので何処かでループするように思いました)

  - ◇ 基本的に動的な実行順で命令が格納されるリングバッファになっている
  - ◇ フェッチしたら tail に挿入，コミット時に head から抜く
  - ◇ 自分から後ろをフラッシュや検索 = 自分から tail ポインタまで･･･ のような制御

質問や感想

- 不可分操作に用いられるアトミック命令はCPU内部でどのようなモジュールで実行、実現されるのでしょうか
- ROB、発行キュー、物理レジスタ数の間には関係(制約？)があると思いますが、どういうふうにこれらのエントリ数は決定しているのでしょうか。

質問や感想

- 就活してて自分の分野と仕事をマッチさせるの難しいなと言う所なのですが、このアーキテクチャの分野だと、メーカーになるのかITになるのかどっちなんでしょう。開発ではある気がするのですが

質問や感想

- 本筋とは無関係なのですが、違反が合った時に以降のプロセスについて「全部殺す」という表現に疾走感のようなものを感じました（プロセスを殺すというのは普段から使う表現ですが、全部とつくとその勢いをはっきり感じられるという意味で）
- IT用語には物騒なものが多くあるという話題をどこかで見かけたのを思い出しました。

質問や感想

- I am a second-year master student, who is facing graduation this August. Under such circumstances, will the deadline for the final report be different for me?
  - ◇ Basically the same, but please submit it faster if possible.

今回の内容

1. GPU のアーキテクチャの基本
    1. 基本的な構造
    2. バックエッジの対処
2. アクセラレータは何故速いのか（概略）

GPU：Graphics Processing Unit

- GPU
  - ◇ もともとはグラフィックを高速に処理するためにあった
- GP-GPU（General Purpose computing on GPU）
  - ◇ グラフィック以外に，汎用の計算に GPU を使用する使い方
  - ◇ 大量の単純な処理の繰り返しが得意
    - □ 行列積 → 機械学習，科学技術計算
    - □ 仮想通貨のマイニング
- この講義では GP-GPU での使用を前提として説明
  - ◇ 以降で「GPU」と言った場合，GP-GPU での使用を仮定

GPU が対象とする処理

- 大量の単純な処理の繰り返し
  - ◇ 並列処理できる部分が自明
  - ◇ マルチスレッド化や SIMD 命令化（後述）が簡単にできる

並列処理できる部分が自明なループのマルチスレッド化の例

- ループで書いた行列積のコード

```
for (k = 0; k < 1024; k++) // 最外周ループ
  for (j = 0; j < 1024; j++)
    for (i = 0; i < 1024; i++)
      a[j][i] += b[j][k] * c[k][i];
```

- 最外周ループ部分を複数のスレッドとしてマルチスレッド化

```
// func が 1024 個起動されて並列に実行される
// 0-1023 がスレッド ID として渡される
func(thread_id) {
  k = thread_id;
  for (j = 0; j < 1024; j++) // ここは同じ
    for (i = 0; i < 1024; i++)
      a[j][i] += b[j][k] * c[k][i];
}
```

マルチスレッド化による並列実行

- マルチスレッド化による並列実行による高速化
  - 先ほどの例では 1024 スレッドが同時実行される
- マルチコア CPU でも同じなのでは？
  - マルチコア CPU = 複数の CPU を束ねたもの
  - 複数の CPU でそれぞれでスレッドを並列に実行できる

マルチコアとの違い：回路量あたりの性能の向上

- GPU は同じ処理をするスレッドを多数走らせることに特化
  - ◇ 行列積の例では，全スレッドが「同じ処理」をしている
  - ◇ 正確には，演算は同じでデータが異なる
- みんな同じ処理をしているので，それを利用して回路を簡略化
  - ◇ みんな同じ処理をしている = 制御回路を共有できる
- 簡略化した分，大量に載せることで性能を上げる

代表的な GPU のアーキテクチャ

- GPU は同じ処理をするスレッドを多数走らせることに特化
  - ◇ これを効率的に実現するのが SIMD と呼ばれる方式
  - ◇ SIMD は GPU のハードウェア方式やプログラミング・モデルの根幹に関わる
- 大きく分けて 2 つの方式
  1. NVIDIA： Single Instruction Multiple Thread (SIMT) 方式
  2. AMD/Intel：Single Instruction stream Multiple Data (SIMD) 方式
- SIMD 方式から順に説明
  - ◇ SIMT 方式は SIMD を発展させた概念であるため

GPU の基本的な構造

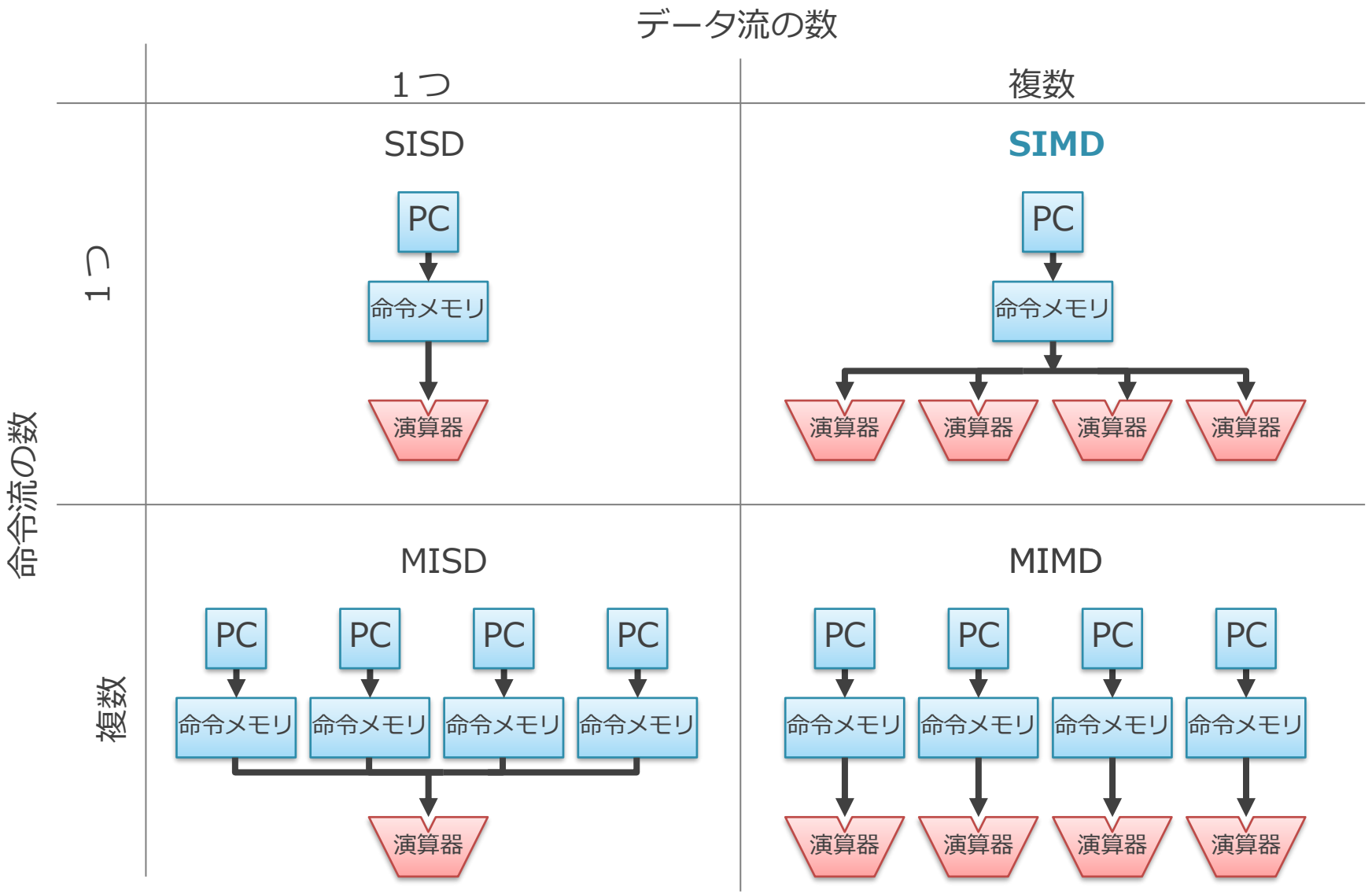
1. フリンの分類と SIMD
2. GPU における回路量当たりの性能向上
3. SIMD プロセッサのプログラミング・モデル

フリンの分類と SIMD

- Single Instruction stream/Multiple Data stream (SIMD)
  - ◇ コンピュータ・ハードウェアの構成方法の分類の１つ
  - ◇ あいだの「stream」は省略されることもある
- フリンによる分類に由来
  - ◇ M. J. Flynn, "Very high-speed computing systems," in Proceedings of the IEEE, vol. 54, no. 12, pp. 1901-1909, Dec. 1966
- プログラムを処理する際の
  - ◇ 「命令の流れ」と「データの流れ」に着目
  - ◇ それぞれが同時に１つあるか，複数あるかで分類

フリンの分類

混乱したらこの図を見返そう

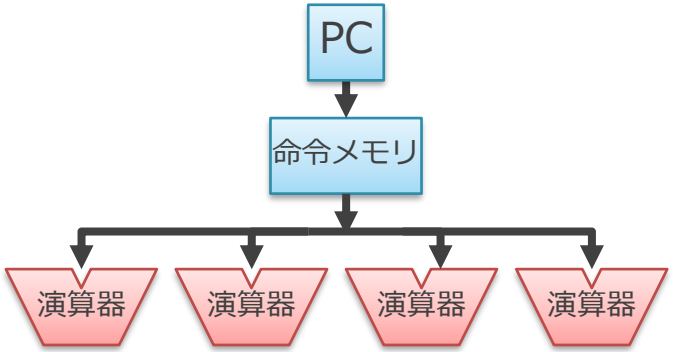


SISD（Single Instruction stream/Single Data stream）

- １つの命令流が１つのデータ流を処理する方式
  - 動作：
    - 同時に１つのプログラム（命令の流れ）を実行
    - それぞれの命令は１つのデータを演算
  - 典型的にはシングルコア CPU がこれに該当

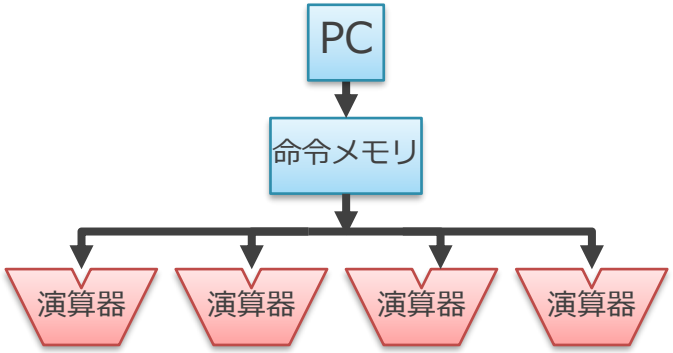
- この分類では，スーパスカラの CPU も SISD の一種であると考える
  - 命令の「流れ」に対するデータの「流れ」は１つのまま
  - 各命令は，１つの演算をデータに対して行っている

SIMD（Single Instruction stream/Multiple Data stream）



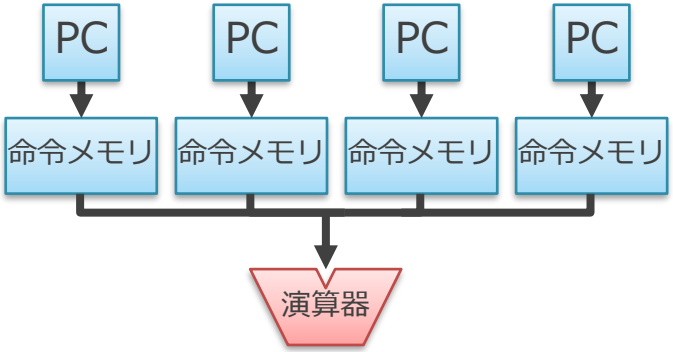
- １つの命令の流れが複数のデータの流れを処理する方式
  - ◇ SIMD は同時に１つのプログラム（命令の流れ）を実行
  - ◇ それぞれの命令は複数のデータに対して同じ演算を行う
- 例：４つの演算を同時に行う SIMD 方式
  - ◇ 解釈された命令は４つの演算器に送信
  - ◇ それぞれの演算器は異なる４つのデータに対して同じ演算を行う
  - ◇ たとえば加算命令であれば，１つの加算命令が４つの加算を行う

SIMD（Single Instruction stream/Multiple Data stream）



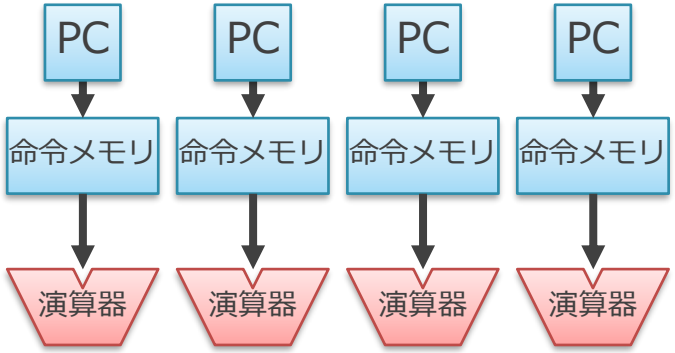
- 現在の主だった GPU はこの SIMD 方式をベースにしている
  - ◇ 一部，CPU にもこの考えを取り入れた命令がある
- GPU は同じ処理を多数走らせることに特化
  - ◇ PC や命令メモリは共有して，各演算器に配ればよい

MISD（Multiple Instruction stream/Single Data stream）



- 複数の命令の流れが単一のデータの流れを処理する方式
  - ◇ 事実上分類の都合上存在していると言ってもよい
  - ◇ この方式の実用的なコンピュータは現在一般的ではない

MIMD（Multiple Instruction stream/Multiple Data stream）



- 複数の命令の流れが複数のデータの流れを処理する方式
  - ◇ 典型的にはマルチコア化された CPU がこれに該当

フリンの分類と実際

- 現代のコンピュータをこのフリンの分類にそのまま厳密に当てはめようとするとやや無理がある
  - 通常は複数の要素を同時に併せ持つことが多い
- たとえば･･･
  - GPU は SIMD 方式のコアを複数備えていることが一般的
    - SIMD と MIMD の要素を併せ持つ
  - CPU では一部の命令でのみ複数のデータを同時に処理する
    - SISD と SIMD の要素を併せ持つ
    - マルチコア化されると MIMD の要素もある
- フリンの分類は，あくまでコンピュータをある特定の軸から見た際の分類であると考える

余談：SIMD の発音

- 業界では「シムディー/sim-dee」と発音する
  - ◇ シムド と読んじゃだめ

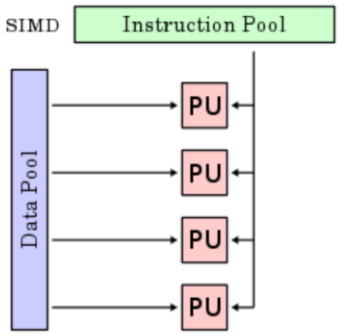
mdn web docs _ References Guides MDN Plus

MDN Web Docs Glossary: Definitions of Web-related terms › SIMD

SIMD

SIMD (pronounced "sim-dee") is short for **Single Instruction/Multiple Data** which is one classification of computer architectures. SIMD allows one same operation to be performed on multiple data points resulting in data level parallelism and thus performance gains — for example, for 3D graphics and video processing, physics simulations or cryptography, and other domains.

Single-Instruction/Multiple-Data Stream
(SIMD or “sim-dee”)



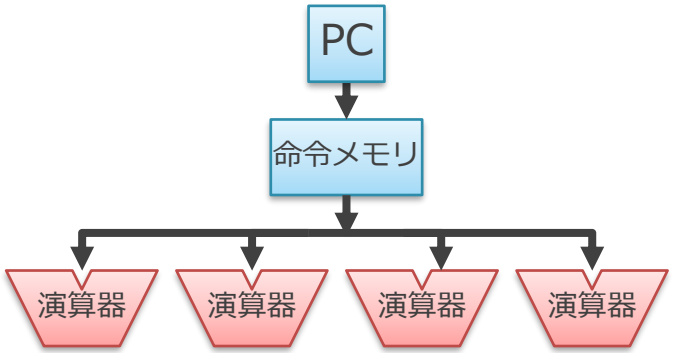
- SIMD computer exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

それぞれ以下より
Mozilla の開発者ページ
https://developer.mozilla.org/en-US/docs/Glossary/SIMD
UCSB の講義資料
https://sites.cs.ucsb.edu/~tyang/class/240a17/slides/SIMD.pdf
イラスト
村上太一@TIER IV

もくじ

回路量当たりの性能の背景：演算器と制御部

- CPU や GPU などを構成する回路を以下に分けて考える
  - 制御部（青）
    - 命令を解釈しそれに基づいて演算器を制御する回路
    - 右の図だと PC や命令メモリ
  - 演算器（赤）
    - 加算や乗算などの演算そのものを行うための回路

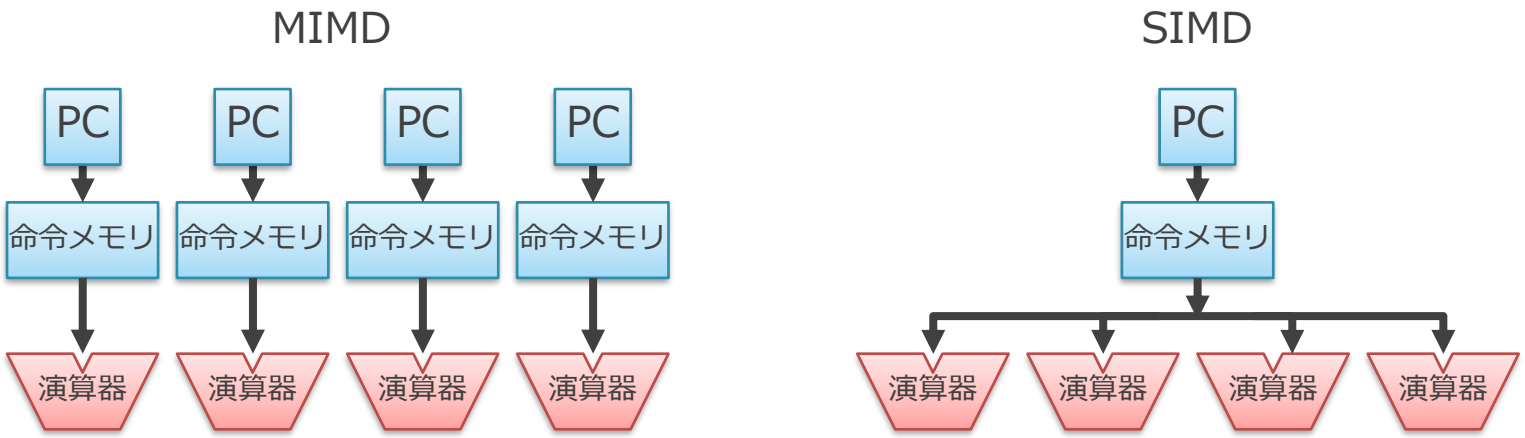


- これに基づいて回路量あたりの性能を議論

GPU における回路量あたりの性能向上

1. 制御部の共有
2. 制御部の小型化

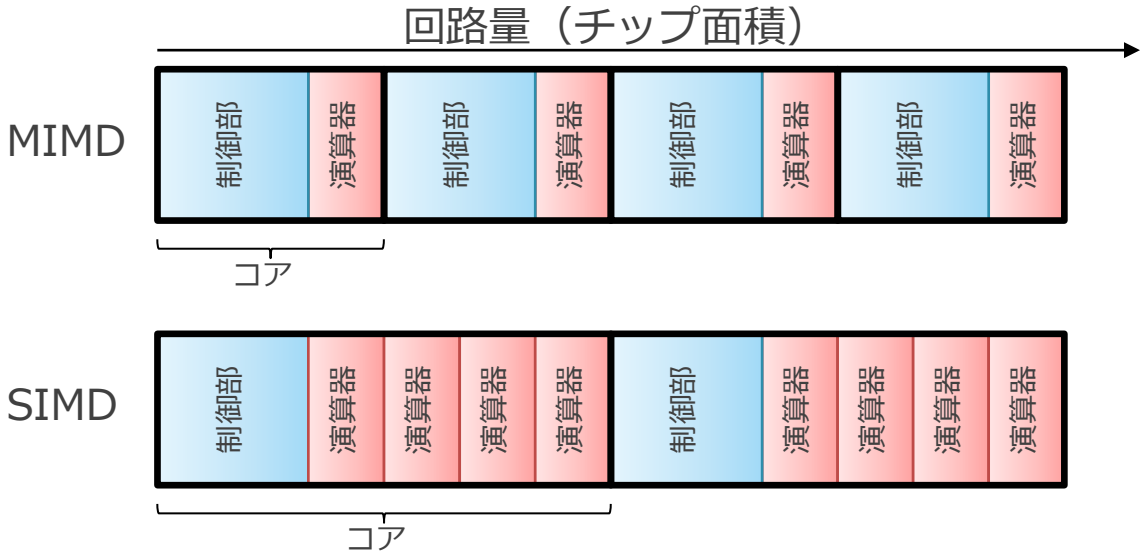
SIMD では複数の演算器間で制御部を共有できる



- たとえば，上の図の MIMD と SIMD の場合
  - 4つの演算器を持ち最大で同時に4並列で演算ができる
    - = 同じ最大性能を持つ
  - SIMD では 1 つの制御部が演算器間で共有されている
    - 実際の GPU では 16 個程度の演算器が共有されていることが多い

SIMD は同じ回路量で高い性能を実現できる

- SIMD は MIMD と比べて全体をより小さく作ることができる
  - ◇ 同じ総面積であれば，SIMD の方がより多くの演算器を搭載できる
    - □ 下の図だと演算器は MIMD４個 vs. SIMD ８個
  - ◇ つまり，同じ回路資源量あたりで，高い性能を出せる
- これが，どの GPU も基本的には SIMD を採用している理由

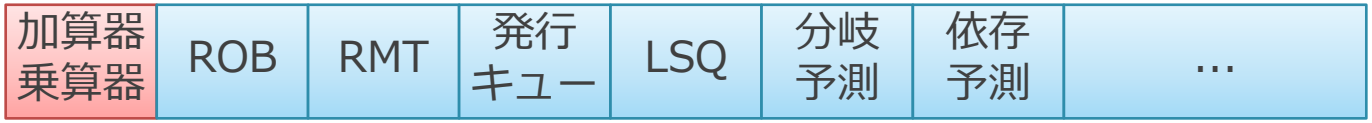


GPU における回路量あたりの性能向上

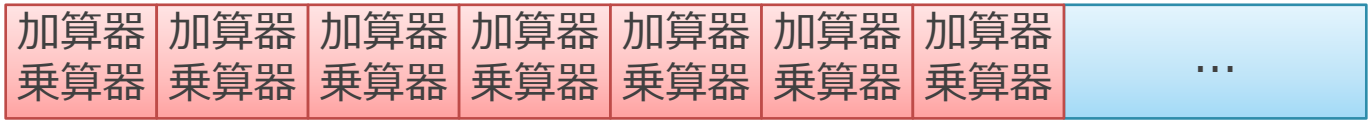
1. 制御部の共有
2. 制御部自体の小型化
   - ◇ （この話は SIMD の話とは直交しているが，関わるのでここで

GPU では制御部自体を簡素化

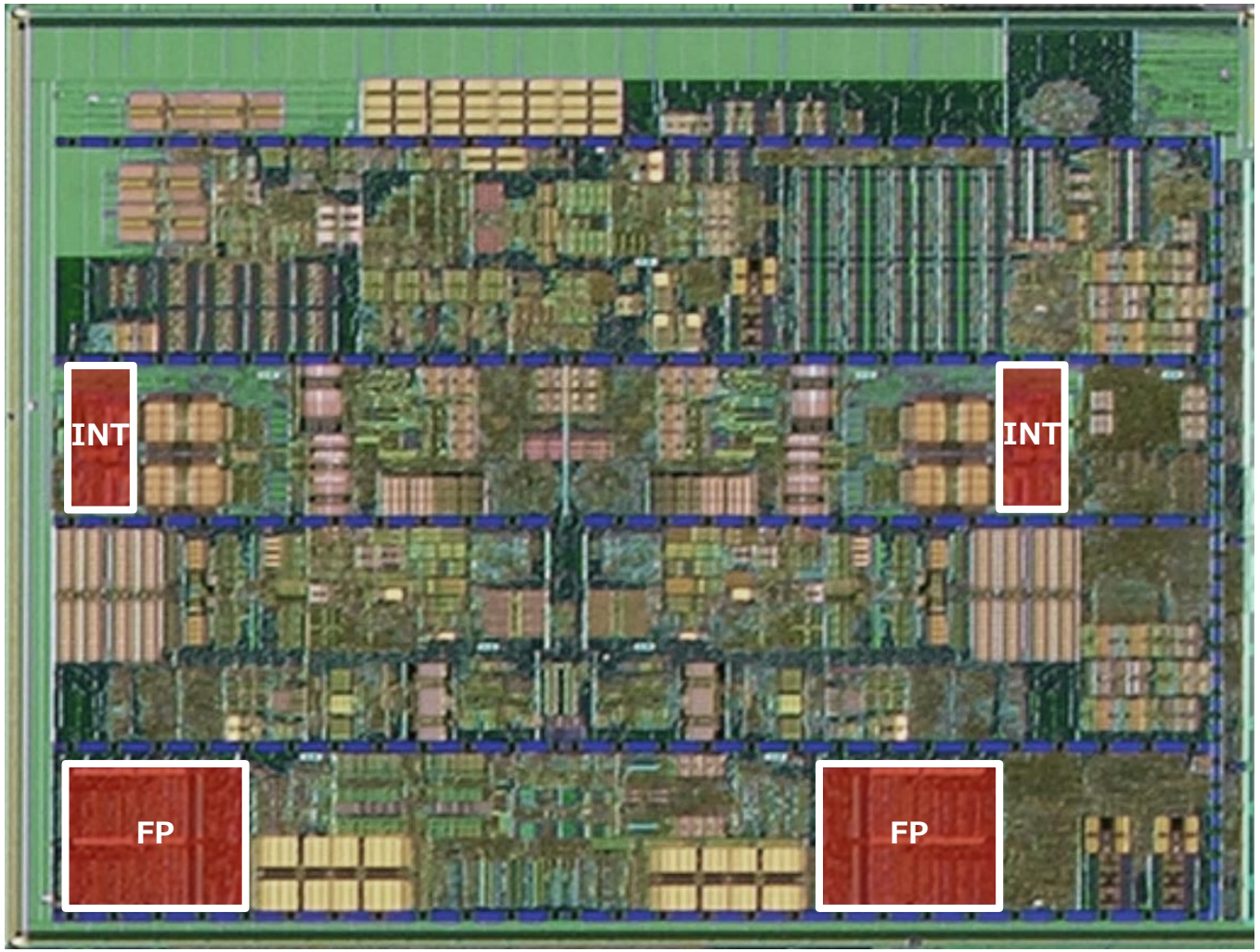
- CPU
  - ◇ 単一のスレッドの実行時間を短くすることに特化
  - ◇ 各種投機実行や，動的命令スケジューリングに回路資源を投入



- GPU
  - ◇ 大量のスレッドの実行スループットを上げることに特化
  - ◇ 演算器以外の部分をそぎ落とし，なるべく大量の演算器を積む



AMD Bulldozer のコア部分に占める演算器部分
チップ写真は https://en.wikichip.org/wiki/File:Bulldozer_Die_Shot.jpg より



- 演算器（INT/FP）が CPU コア全体に占める割合はわずか

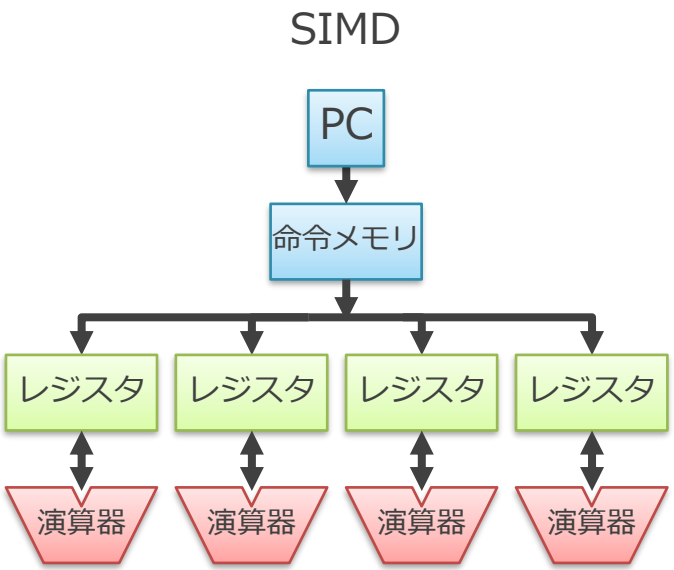
GPU における回路量当たりの性能向上

- GPU における回路量あたりの性能向上
  1. SIMD による制御部の共有
  2. 制御部自体の小型化

- たとえば先ほどのチップ写真の場合・・・
  - ◇ 赤色の部分のみを N 倍に増やしても，全体は大きくは増えない
  - ◇ 非赤色の部分を減らすと，かなりの数の赤色が積める

補足

- ここまでは簡単のためデータの供給についての議論を無視している
  - ◇ メモリやレジスタ・ファイルは演算器間で共有できない
- ここをさらになんとかするのが，行列積に特化したハードウェア
  - □ TPU や Tensor Core
  - □ これらについては後述



もくじ

プログラミング・モデル

- ここまでは SIMD のハードウェアについて説明
  - ◇ ここからはその上でどのようなプログラムを走らせるのかを説明
- プログラミング・モデル
  - ◇ どのような命令によりプログラムを記述するかを規定
  - ◇ これまでに説明したハードウェアの構成方式とはある程度独立した概念

ハードウェア方式とプログラミング・モデル

- 異なるハードウェア方式とプログラミング・モデルの組み合わせ
  - NVIDIA の GPU
    - ハードウェア：　　SIMD（を拡張した SIMT）
    - プログラミング・モデル：Single Program Multiple Data (SPMD)
  - AMD や Intel の GPU
    - ハードウェア：　　SIMD
    - プログラミング・モデル：「SIMD命令」と呼ぶ形式の命令
    - （AMD/Intel でも SPMD モデルにより書かれたプログラムをコンパイラによって SIMD 命令に変換して実行することが行われる

	ハードウェアの方式	プログラミング・モデル
NVIDIA (Volta)	SIMT	SPMD
AMD (GCN,RDNA)	SIMD	SIMD 命令
Intel (GEN)	SIMD	SIMD 命令

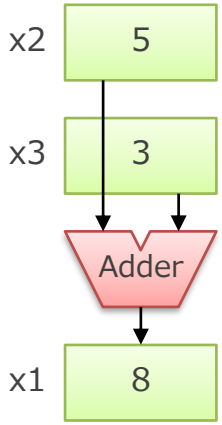
もくじ

用語の定義

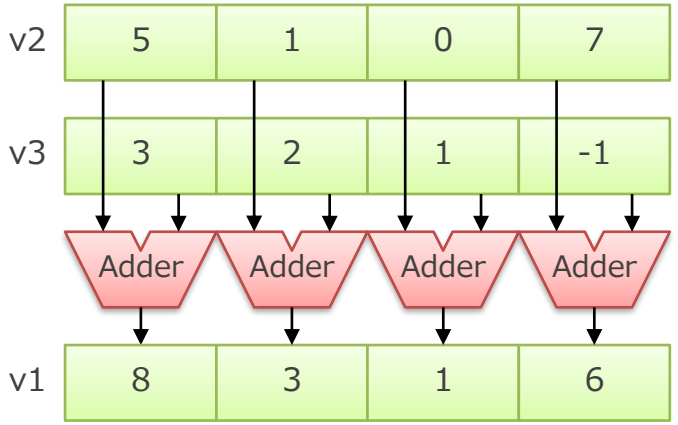
- SIMD プロセッサ：
  - ◇ １つの命令を解釈し，複数のデータに対して同じ演算を同時に行うハードウェア
- SIMD 命令：
  - ◇ SIMD プロセッサ上で複数のデータを対象に同時に演算を行う命令
- まぎらわしい：
  - ◇ 「SIMD（プロセッサ）」はハードウェアの構成方式だが，
  - ◇ 「SIMD 命令」は命令の表現方式を意味することに注意

SIMD 命令：複数のデータを対象に同時に演算を行う

- 通常の命令（スカラ命令）
  - add x1←x2+x3



- SIMD 命令
  - add_x4 v1←v2+v3
  - v1~v3 は１つのレジスタに４つの値が入ったレジスタ
  - SIMD レジスタやベクトルレジスタと呼ばれる



SIMD 命令を使う方法

- SIMD 命令を使う方法
  1. コンパイラに頑張ってもらう
     - 複数同時に計算しても大丈夫なところを自動で検出する
     - 後述の SPMD で書かれたプログラムならある程度できる
  2. （余談）組み込み関数を使って人間が直接書く
     - 自動でうまくいかない場合は人間が頑張る

（余談）組み込み関数（intrinsic）

```
// 連続した加算を４つ行う SIMD 命令に相当する関数
// この関数と等価な結果が得られる命令に変換される
void add_x4(int* dst, int* src1, int* src2) {
    dst[0] = src1[0] + src2[0];
    dst[1] = src1[1] + src2[1];
    dst[2] = src1[2] + src2[2];
    dst[3] = src1[3] + src2[3];
}

// 連続した乗算を４つ行う SIMD 命令に相当する関数
void mul_x4(int* dst, int* src1, int* src2) {
    dst[0] = src1[0] * src2[0];
    dst[1] = src1[1] * src2[1];
    dst[2] = src1[2] * src2[2];
    dst[3] = src1[3] * src2[3];
}
```

（余談）組み込み関数の使用例

```
// src1 と src2 から n個 のベクトルの
// 内積を行い，返り値として返す関数
int dot_product(int* s1, int* s2, int n) {
    int r = 0;
    for (int i = 0; i < n; i++) {
        r += s1[i] * s2[i];
    }
    return r;
}
```

```
// ベクトルの内積を行う関数の SIMD 命令版
// データの個数は４の倍数であるとする
int dot_product_x4(int* s1, int* s2, int n) {

    // 4並列で内積の処理を行う
    int r_x4[4] = {0, 0, 0, 0};
    for (int i = 0; i < n; i+=4) {
        int tmp_x4[4];
        mul_x4(tmp_x4, s1 + i, s2 + i);
        add_x4(r_x4, r_x4, tmp_x4);
    }

    // 4つの中間結果をまとめる
    int r = 0;
    for (int i = 0; i < 4; i++) {
        r += r_x4[i];
    }
    return r;
}
```

- mul_x4 や add_x4 の部分が SIMD 命令に置き換わる形でコンパイルされる
  - ◇ これを人手でやるのは結構しんどい

もくじ

SIMD 命令と SPMD

- SIMD 命令
  - SIMD のハードウェアを直接命令に見せている
- Single Program Multiple Data (SPMD) ：
  - マルチスレッド方式によってプログラムを記述
    - Single Program = 同じことをするスレッド
    - 本日冒頭の行列積の例そのもの
  - SIMD のハードウェアはプログラマからは直接見えないよう隠蔽される

SPMD による並列化の例

- ループで書いた行列積のコード

```
for (k = 0; k < 1024; k++) // 最外周ループ
  for (j = 0; j < 1024; j++)
    for (i = 0; i < 1024; i++)
      a[j][i] += b[j][k] * c[k][i];
```

- 最外周ループ部分を複数のスレッドとしてマルチスレッド化

```
// func が 1024 個起動されて並列に実行される
// 0-1023 がスレッド ID として渡される
func(thread_id) {
  k = thread_id;
  for (j = 0; j < 1024; j++) // ここは同じ
    for (i = 0; i < 1024; i++)
      a[j][i] += b[j][k] * c[k][i];
}
```

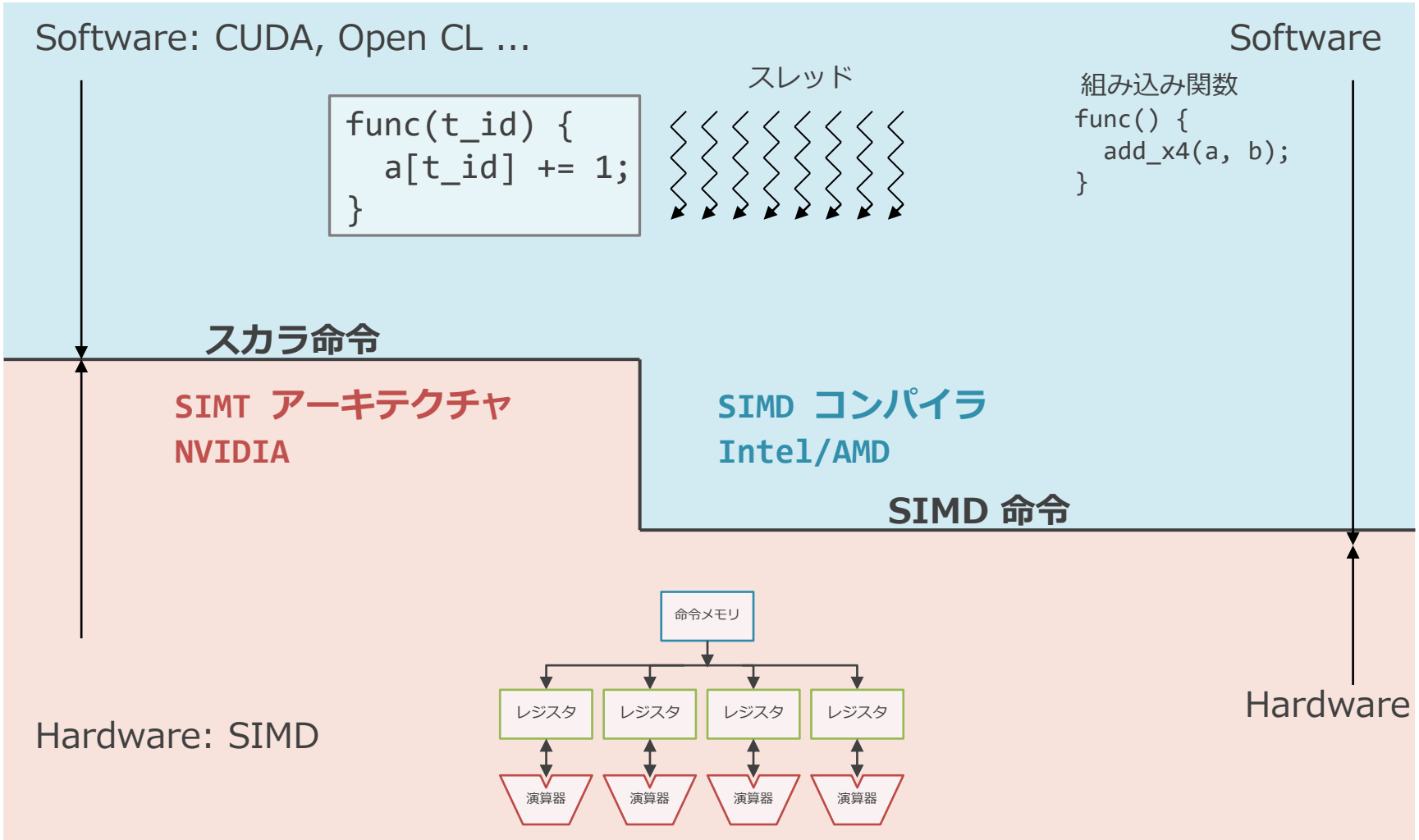
SPMD で書かれたプログラムの実行

1. SIMD 命令への変換
   - ◇ コンパイラにより自動で変換
   - ◇ 変換された SIMD 命令を実行
   - ◇ AMD/Intel の方式

2. Single Thread Multiple Thread (SIMT) ハードウェアによる実行
   - ◇ 命令列自体はスカラ命令で記述
   - ◇ マルチスレッド実行をハードウェアで SIMD にマップする
   - ◇ NVIDIA の方式

GPU におけるソフトとハードの関係

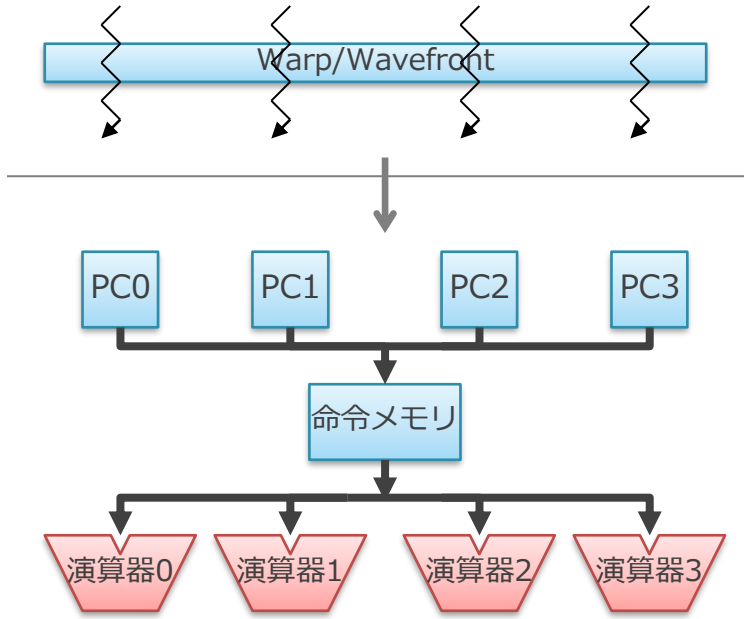
図は Caroline Collange “GPU architecture part 2: SIMT control flow management” より



SIMT の構造

- PC を演算器の分だけ持つ
  - ◇ 命令メモリは１つのまま
- SPMD の各スレッドを PC に割り当てる
  - ◇ 基本的にはスレッドを時分割で
    それぞれ実行
- ポイント：
  - ◇ 全スレッドの PC が同じアドレスを
    指している場合は SIMD プロセッサ
    として振る舞う
    - □ このまとまりを Warp/Wavefront と呼ぶ
  - ◇ 分岐しなければこの状態が保たれる

```
// スレッド内は逐次処理で記述
void add(*a, *b, *c){
    c[t_id] = a[t_id] + b[t_id]
}
```



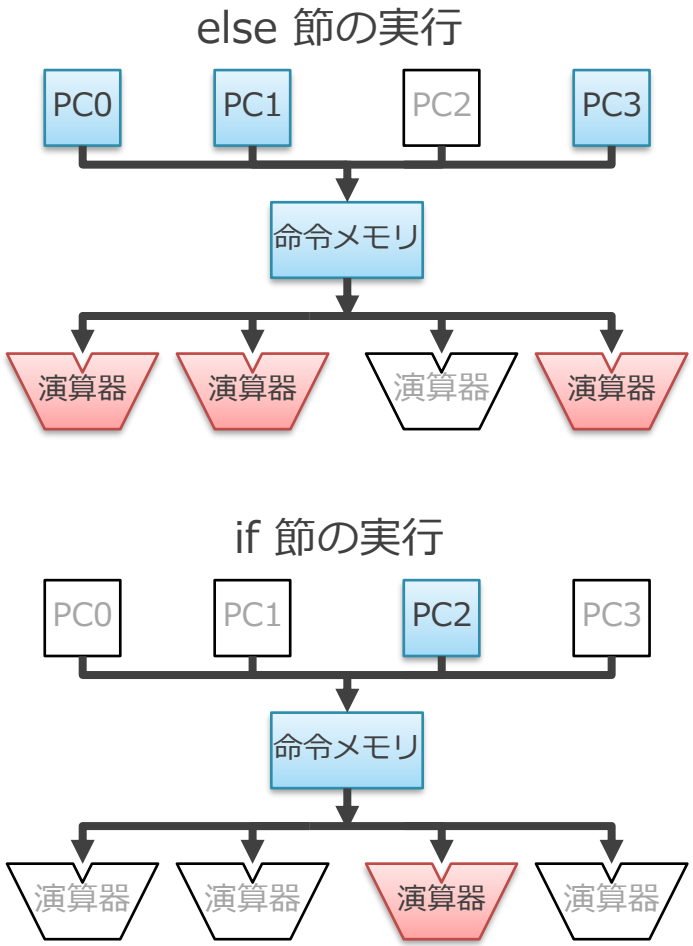
SIMT

SIMT の利点と問題点

- 利点：
  - マルチスレッドで書かれたプログラムが SIMD プロセッサでそのまま実行できる
    = 難しいコンパイラがいらない
- 問題点：
  - スレッド間で違う方向に分岐すると実行速度が落ちる
    - branch divergence と呼ぶ
  - 水平方向の演算ができない

問題１：スレッド間で違う方向に分岐すると実行速度が落ちる（Branch divergence）

```
int branch_func(int t_id, int i) {
    if (t_id == 2)
       i++;
    else
       i--;
    return i;
}
```

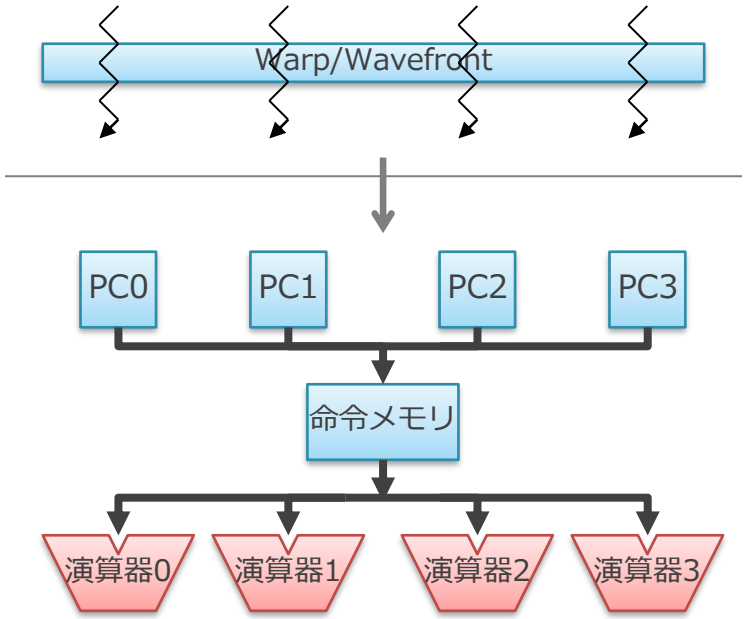


- else と if の ２回に分けて実行される
  - ◇ この例では実行速度が ½ に

問題２ 水平方向の演算ができない

- 各演算器は独立したスレッドに割り当てられる
- 横にある演算器間では直接データのやり取りができない
  - ◇ 横で何が行われているかは保証できないため，そのようなプログラムが記述できない
  - ◇ SIMD 命令であれば，そのようなプログラムが書ける

```
// スレッド内は逐次処理で記述
void add(*a, *b, *c){
    c[t_id] = a[t_id] + b[t_id]
}
```



SIMD 命令と SIMT のまとめ

- SIMD プロセッサをどのように使うか
  - ◇ SIMD 命令 vs. SIMT
    - □ （モデルの名前が適切ではないと思う
  - ◇ 異なる層で，複数演算器の同時処理の対応を行っている
    - □ SIMD 命令： コンパイラ or 人手
    - □ SIMT： ハードウェア

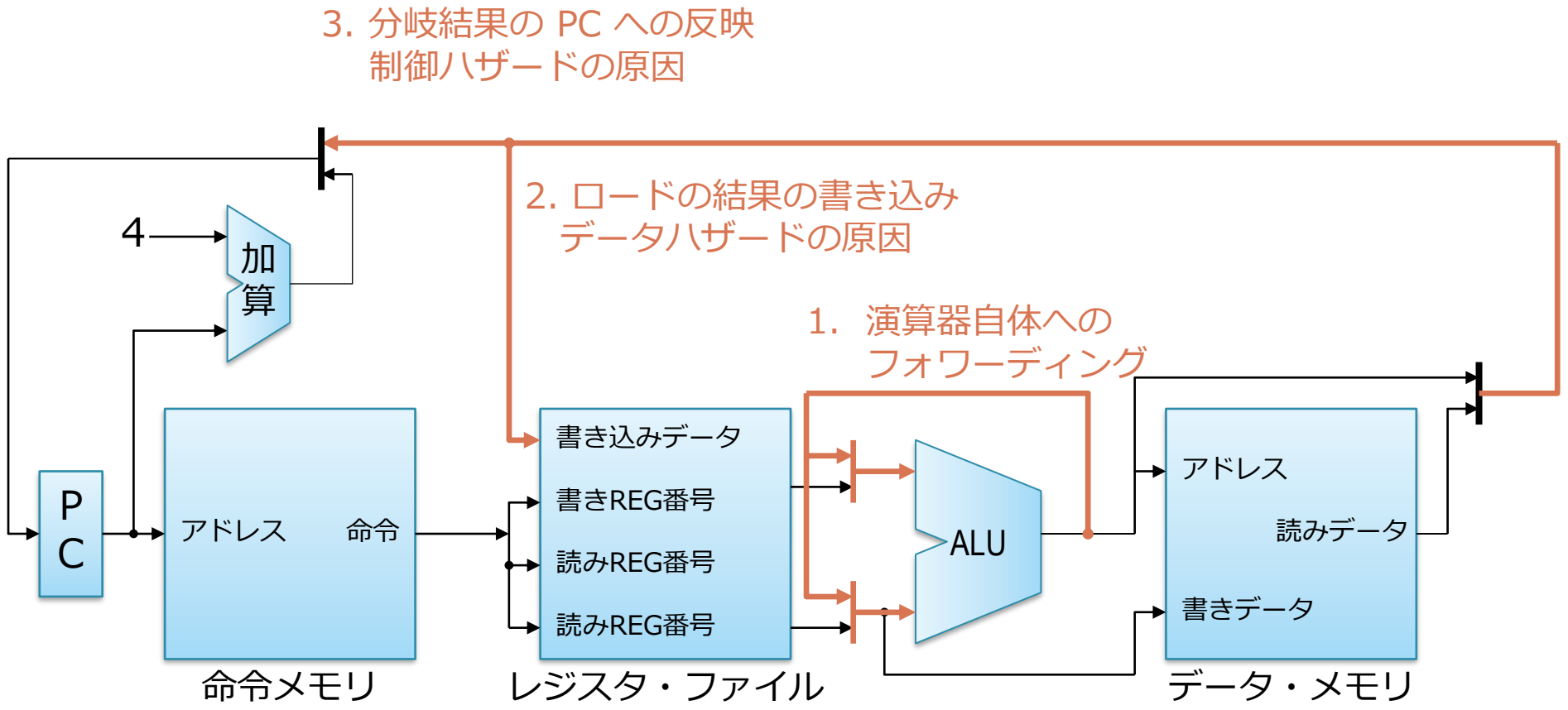
今回の内容

1. GPU のアーキテクチャの基本
    1. 基本的な構造
    2. バックエッジの対処
2. アクセラレータは何故速いのか（概略）

バックエッジに対する対処

- CPU の場合
  - フォワーディング，分岐予測，命令スケジューリング…
- GPU の場合
  - 上記を一切行わない
  - 全部マルチスレッディングで対処
    - 動かすべきスレッドは大量にある

CPU のバックエッジ：逆方向（右から左）

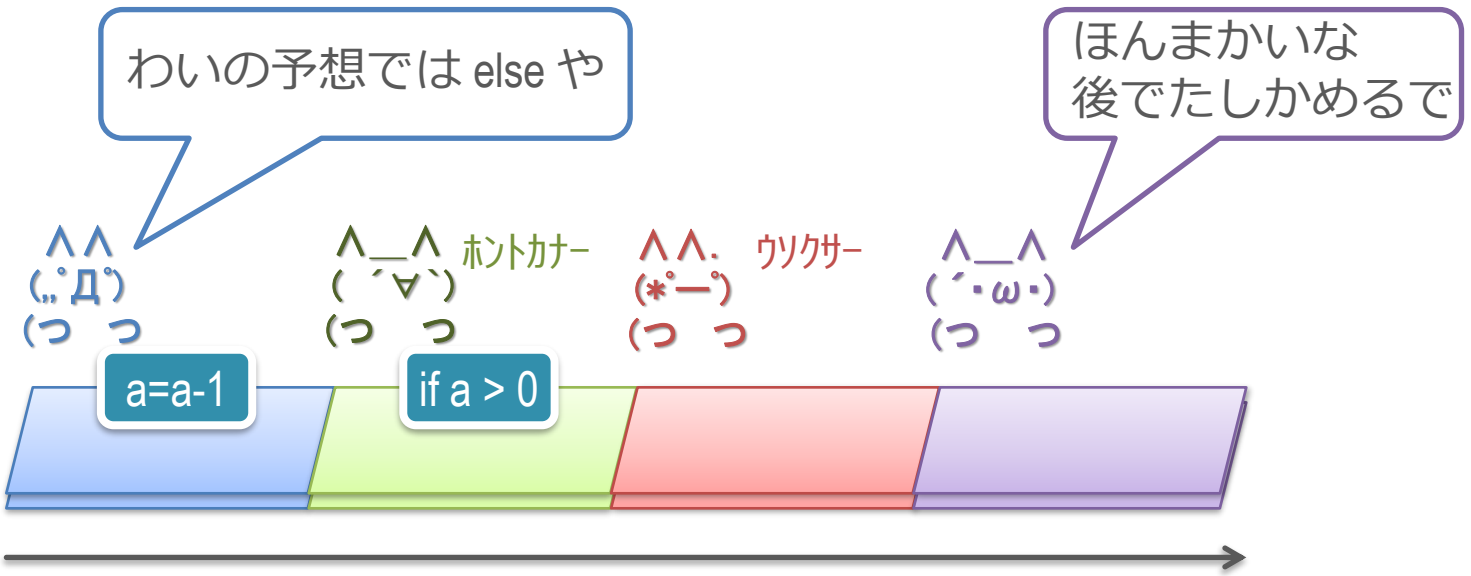


- バックエッジがあるため，命令を単純に流せない場合がある
  - ◇ 工場のラインのように，一方向に流せない

分岐予測

プログラム

```
if a > 0:
    a=a+1
else:
    a=a-1
```

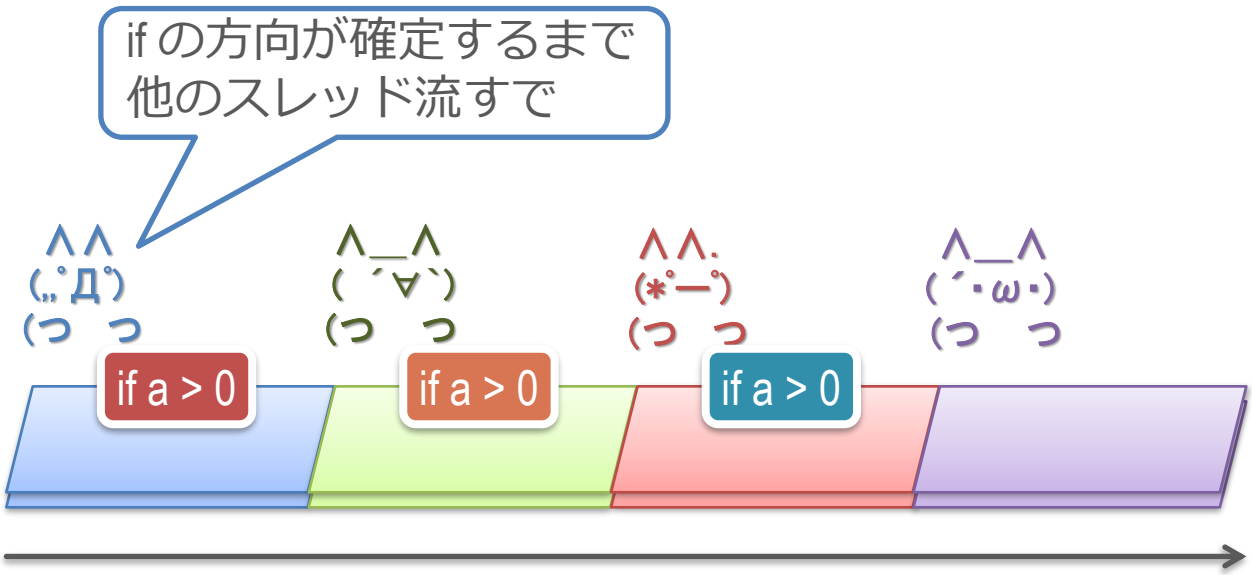


- 動作
  - 「if a > 0」の結果を予測して，命令を取り込む
    - 前回はこっちに行ったので，次もこっちに違いないとかで予測
  - あとから予測が正しいか確認する

マルチスレッドによる解決

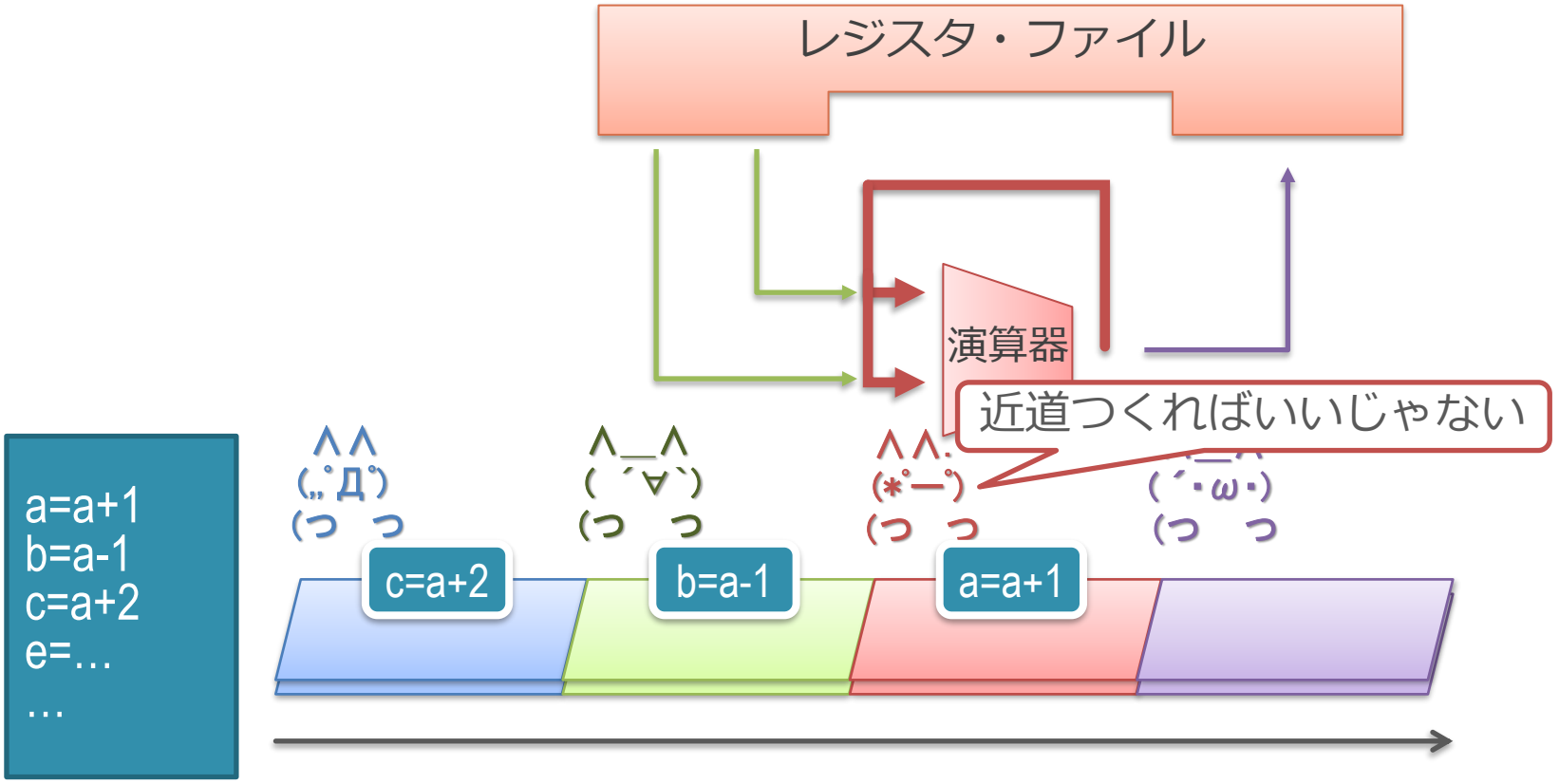
プログラム

```
if a > 0:
    a=a+1
else:
    a=a-1
```



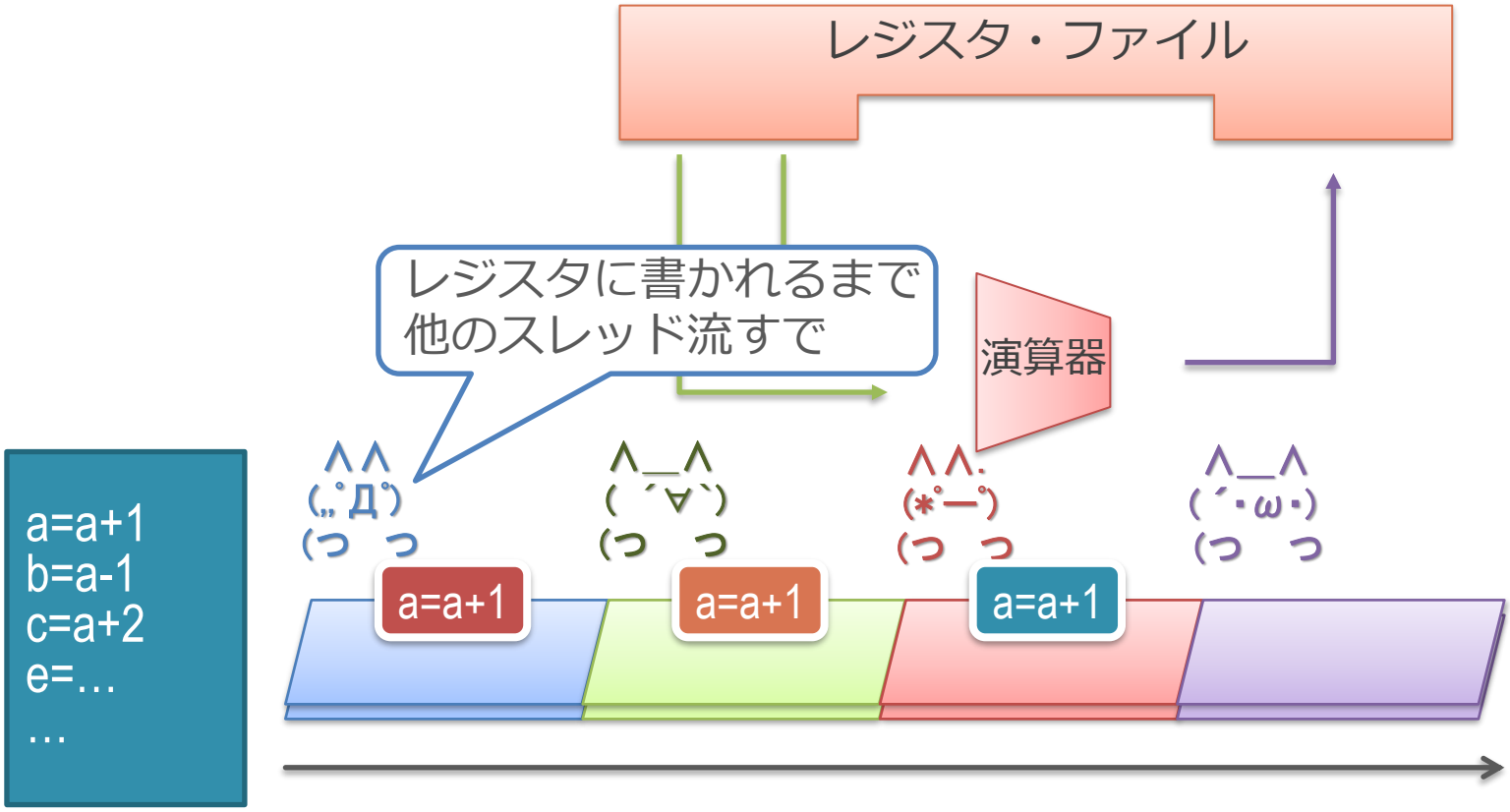
- 動作
  - 「if a > 0」の結果が出るまで他のスレッドの命令をフェッチ
  - フェッチすべき候補のスレッドは大量にある

フォワーディング



- フォワーディング（バイパスとも呼ぶ）
  - (*ﾟ―ﾟ) の人が，次のサイクルに結果を使えるようショートカットを作って手元に結果をおいておく

マルチスレッドによる解決



- レジスタ・ファイルに値が書き込まれるまで，他のスレッドを流す
  - ◇ 複数のスレッドの a が同時に存在するので，レジスタは大きくなる

GPU のアーキテクチャのまとめ

- 下記の性質を利用
  1. 並列に動作する大量のスレッドがある
  2. 各スレッドは基本的に同じことをしている
- 演算器以外の回路を削減
  - ◇ 制御部を共有してハード量当たりの性能を向上
  - ◇ バックエッジは全部マルチスレッドで対処
    - □ 依存関係やキャッシュ・ミスで実行できない場合も，他のスレッドの実行ですます

今回の内容

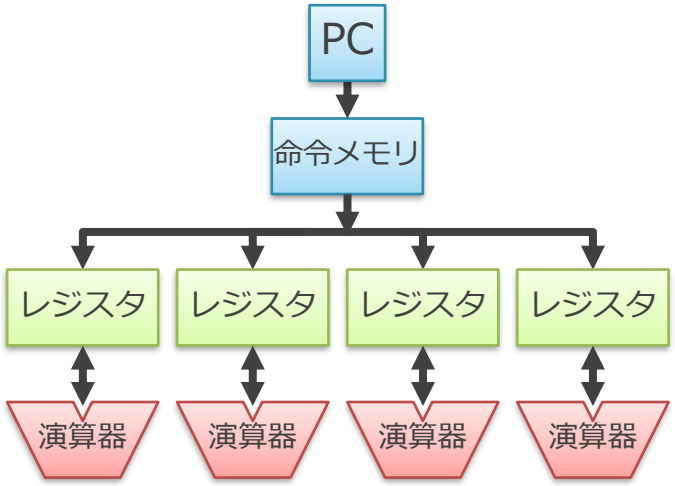
1. GPU のアーキテクチャの基本
    1. 基本的な構造
    2. バックエッジの対処
2. アクセラレータは何故速いのか（概略）

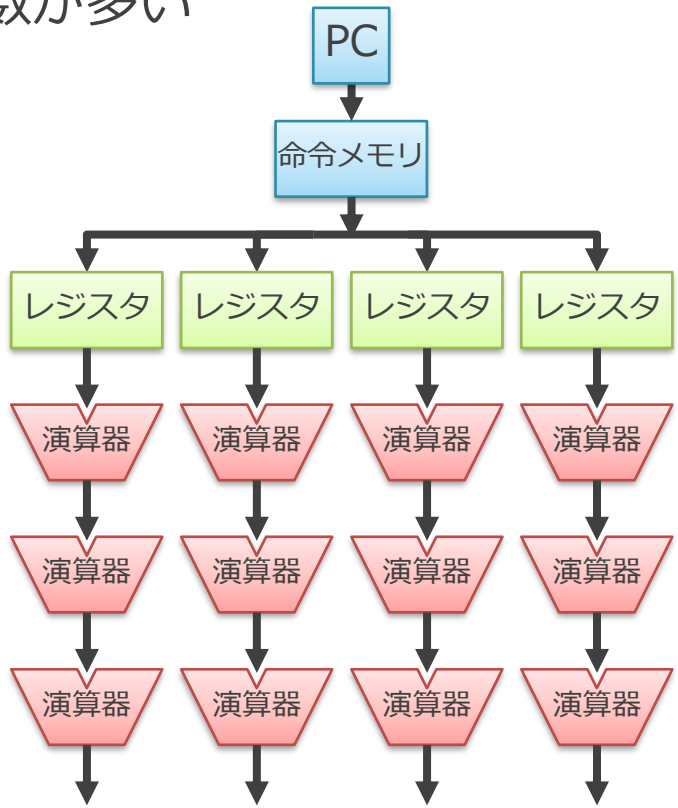
アクセラレータや GPU は何故 CPU より速いのか？

- アクセラレータ
  - ◇ 特定の問題に特化したハードウェア
- 演算器そのものは一緒
  - ◇ 加算器や乗算器などはどのアーキテクチャでも共通
- 制御部やデータのやりとりをいかに減らすかにより決まる
  - ◇ 対象のプログラムの性質に特化することで，機能を削っても性能が落ちないようにする
  - ◇ 減らした分だけよりたくさん演算器がつめる

行列積アクセラレータの場合

- 行列積では GPU はまだ無駄がある
  - ◇ 演算1回ごとに命令を読んだり，レジスタ・ファイルにアクセス
- 演算器を直接繋いでそこにデータを流せば良い
  - ◇ 行列積は入力データ数と比べて演算数が多い
    - □ $O(N^2)$ vs. $O(N^3)$
  - ◇ Google TPU や NVIDIA Tensor Core





Norman P. Jouppi et al., In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017 より

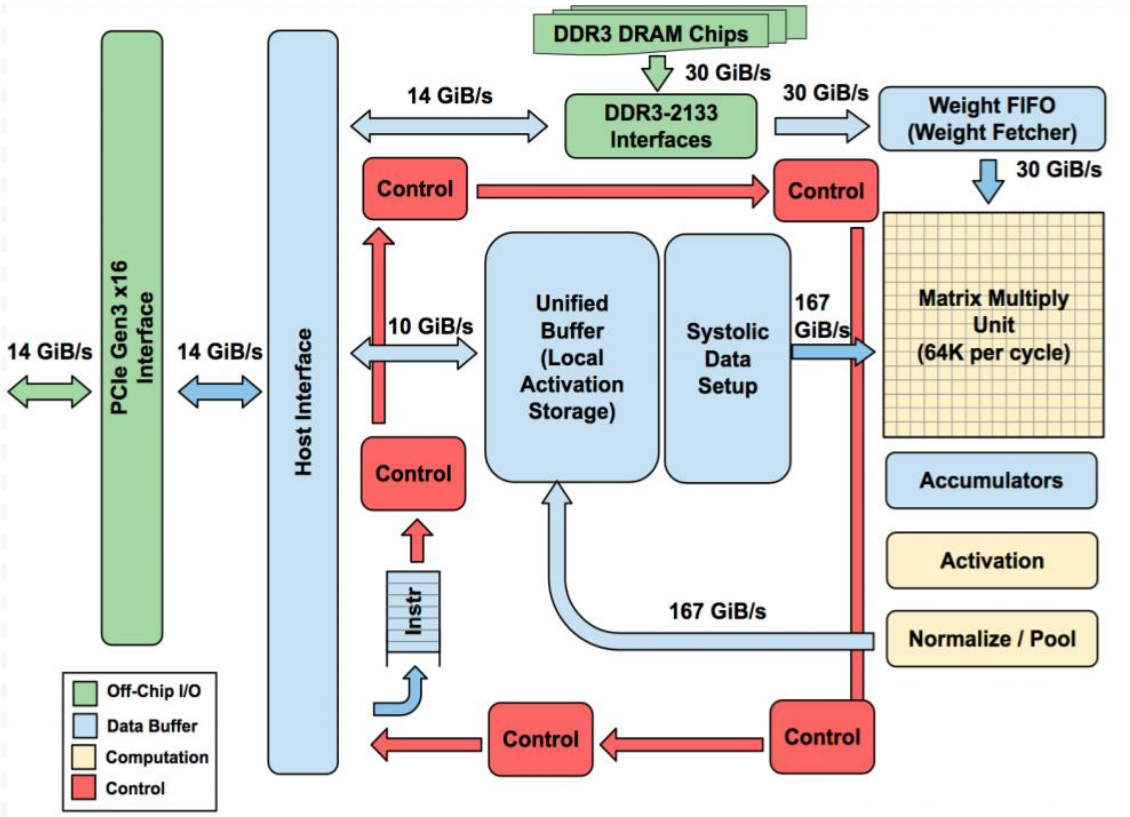
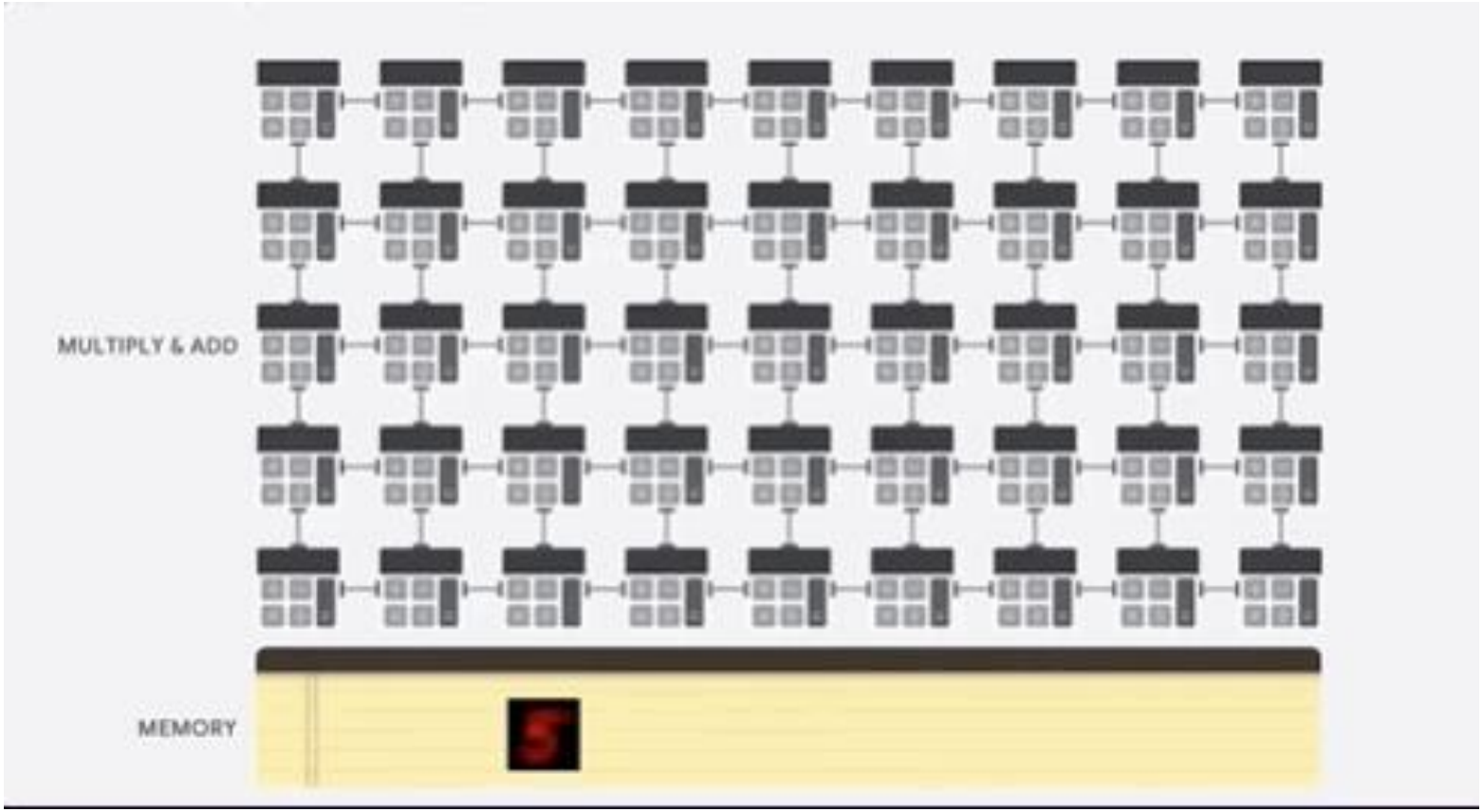


Figure 1. TPU Block Diagram. The main computation is the yellow Matrix Multiply unit. Its inputs are the blue Weight FIFO and the blue Unified Buffer and its output is the blue Accumulators. The yellow Activation Unit performs the nonlinear functions on the Accumulators, which go to the Unified Buffer.

https://cloud.google.com/tpu?hl=ja　より



今回の内容

1. GPU のアーキテクチャの基本
    1. 基本的な構造
    2. バックエッジの対処
2. アクセラレータは何故速いのか

レポート課題

- １０回目講義資料を参照

出欠と感想

- 本日の講義でよくわかったところ，わからなかったところ，質問，感想などを書いてください
  - ◇ LMS の出席を設定するので，そこにお願いします
  - ◇ パスワード:
- 意見や内容へのリクエストもあったら書いてください
- LMS の出席の締め切りは来週の講義開始までに設定してあります
  - ◇ 仕様上「遅刻」表示になりますが，特に減点等しません
  - ◇ 来週の講義開始までは感想や質問などを受け付けます