

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

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

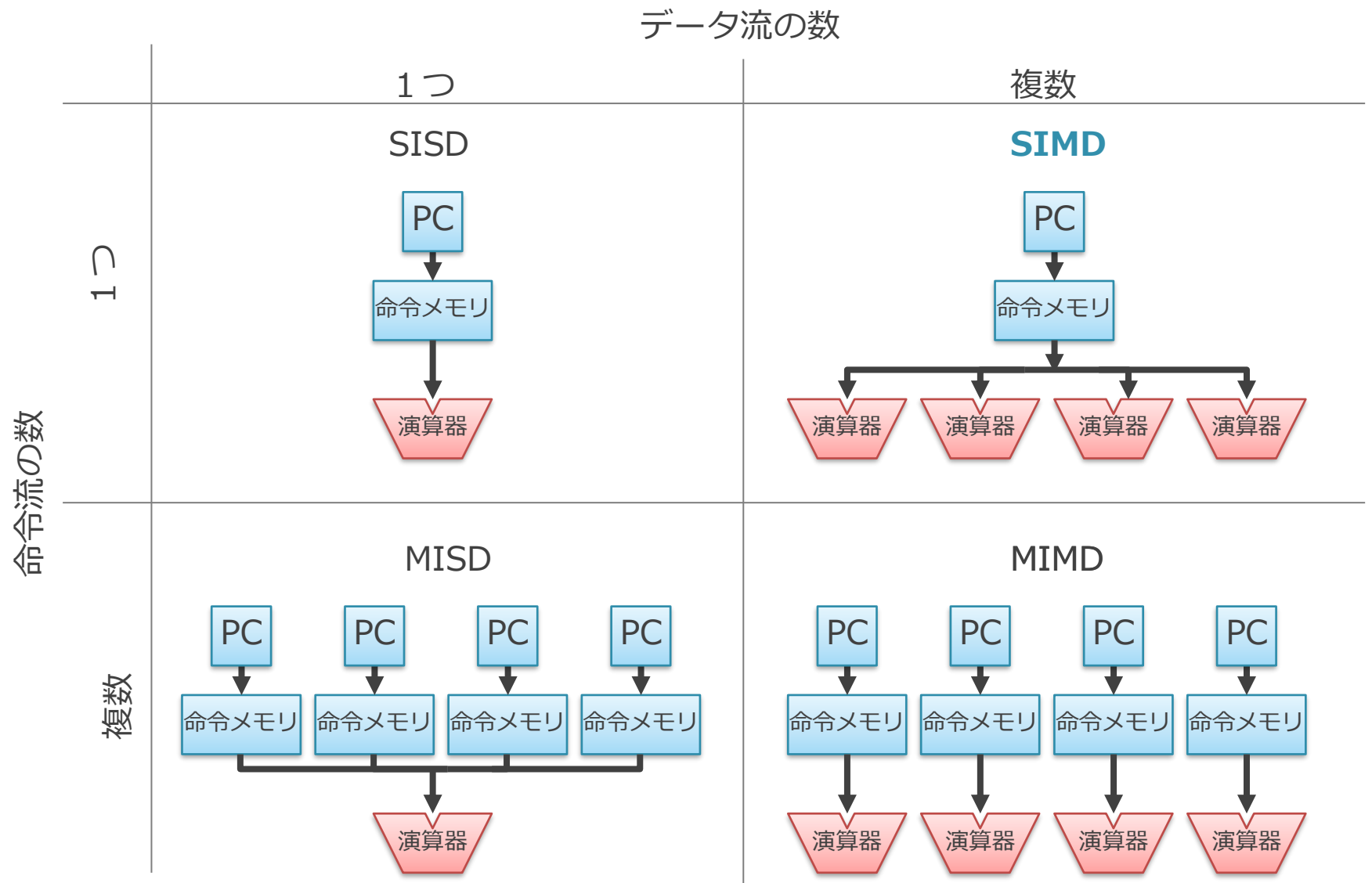
1. フリンの分類と SIMD
2. GPU における回路量当たりの性能向上
3. SIMD プロセッサのプログラミング・モデル

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

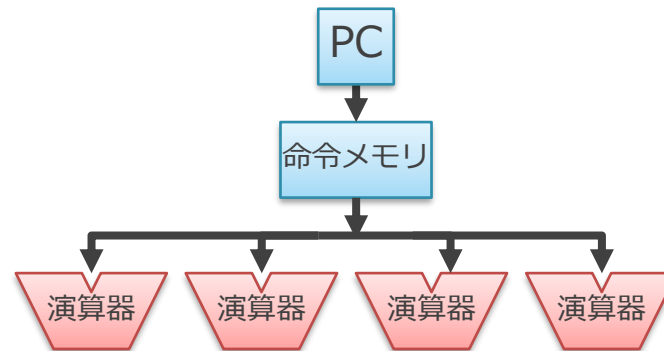


SISD (Single Instruction stream/Single Data stream)



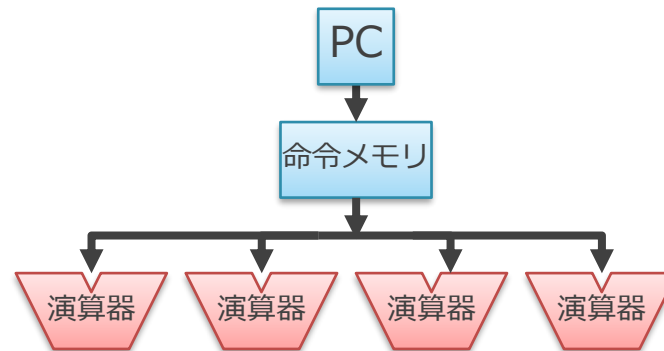
- 1つの命令流が1つのデータ流を処理する方式
 - ◇ 動作：
 - 同時に1つのプログラム（命令の流れ）を実行
 - それぞれの命令は1つのデータを演算
 - ◇ 典型的にはシングルコア CPU がこれに該当
- この分類では、スーパスカラのCPUもSISDの一種であると考えられる
 - ◇ 命令の「流れ」に対するデータの「流れ」は1つのまま
 - ◇ 各命令は、1つの演算をデータに対して行っている

SIMD (Single Instruction stream/Multiple Data stream)



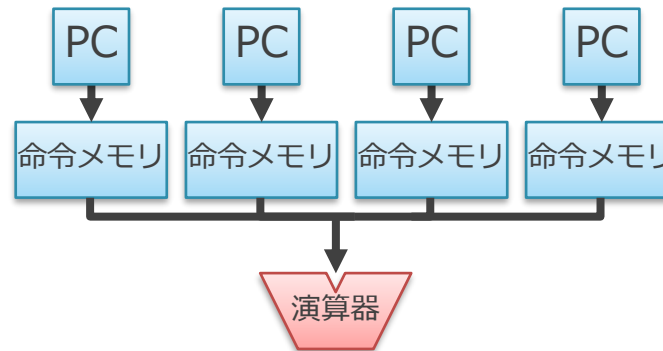
- 1つの命令の流れが複数のデータの流を処理する方式
 - ◇ SIMD は同時に1つのプログラム（命令の流れ）を実行
 - ◇ それぞれの命令は複数のデータに対して同じ演算を行う
- 例：4つの演算を同時に行う SIMD 方式
 - ◇ 解釈された命令は4つの演算器に送信
 - ◇ それぞれの演算器は異なる4つのデータに対して同じ演算を行う
 - ◇ たとえば加算命令であれば、1つの加算命令が4つの加算を行う

SIMD (Single Instruction stream/Multiple Data stream)



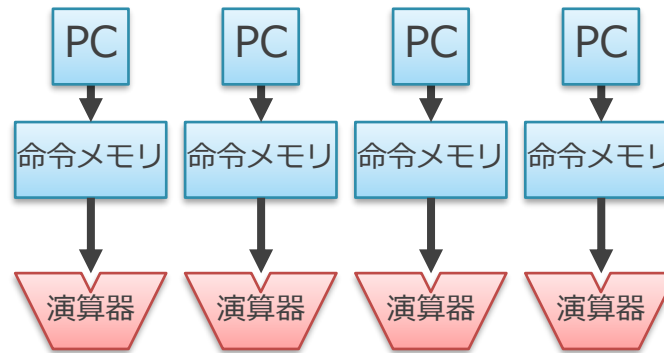
- 現在の主だった GPU はこの SIMD 方式をベースにしている
 - ◇ 一部, CPU にもこの考えを取り入れた命令がある
- GPU は同じ処理を多数走らせることに特化
 - ◇ PC や命令メモリは共有して, 各演算器に配ればよい

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

余談：SIMD の発音

■ 業界では「シムディー/sim-dee」と発音する

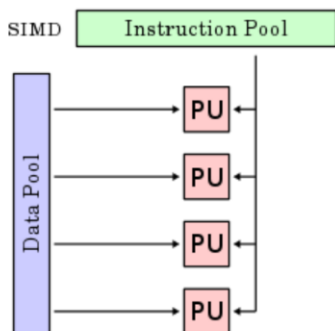
◇ シムド と読んじゃだめ

[mdn web docs](#) [References](#) [Guides](#) [MDN Plus](#)

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

SIMD

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)

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.

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

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
 1. 演算器と制御部
 2. SIMD による制御部の共有
 3. 制御部自体の小型化
3. SIMD プロセッサのプログラミング・モデル

回路量当たりの性能の背景：演算器と制御部

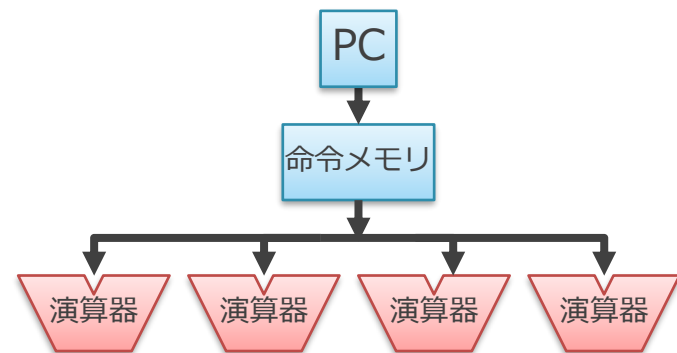
- CPU や GPU などを構成する回路を以下に分けて考える

- ◇ 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

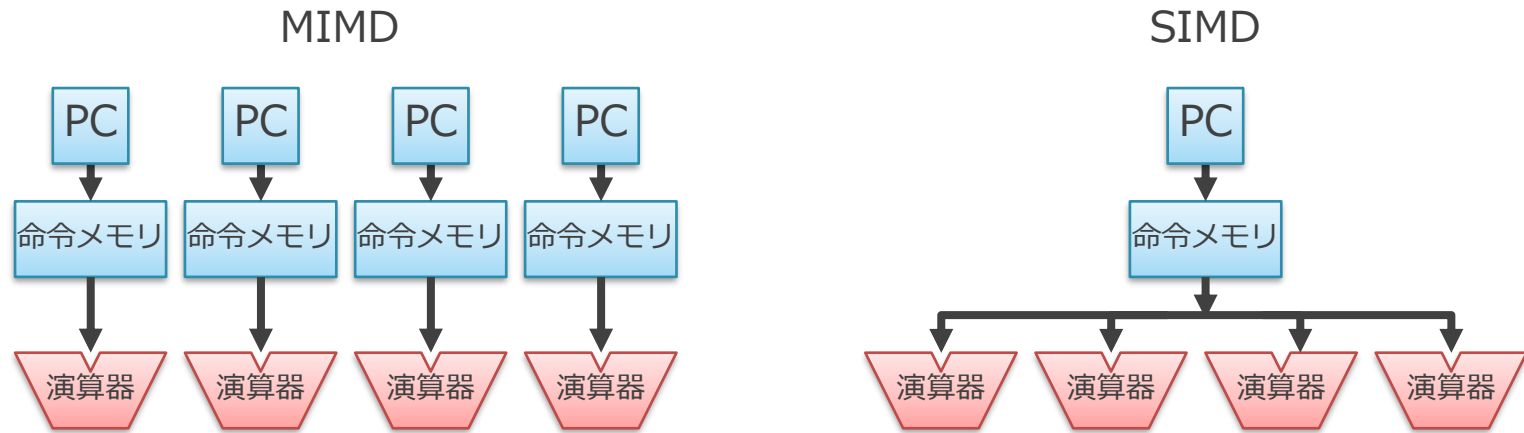


- これに基づいて回路量あたりの性能を議論

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

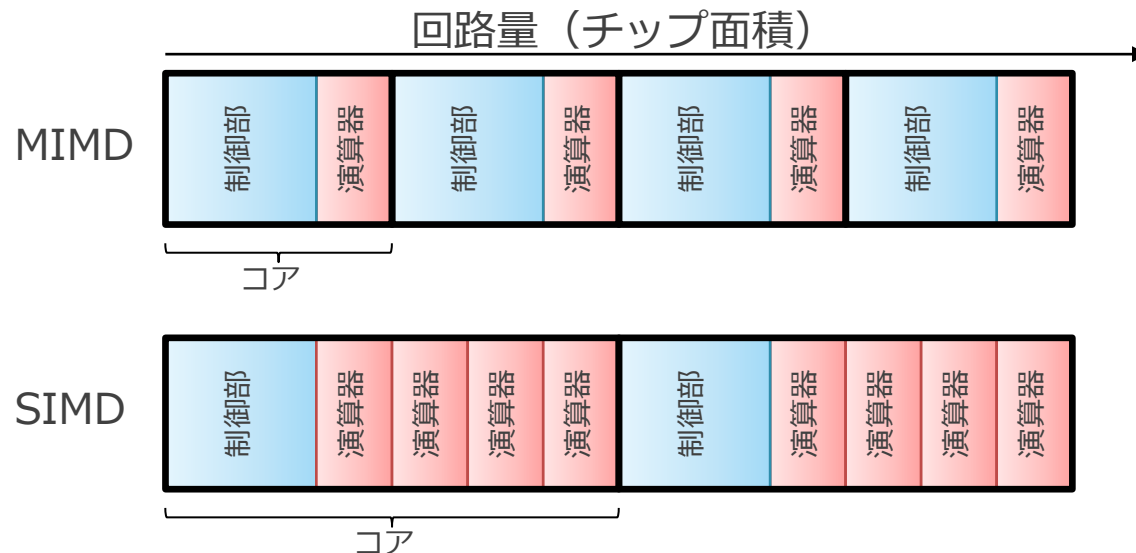
SIMD では複数の演算器間で制御部を共有できる



- 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 **MIMD 4 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

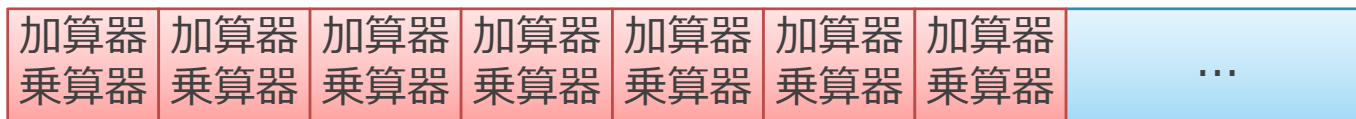
■ CPU

- ◇ 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



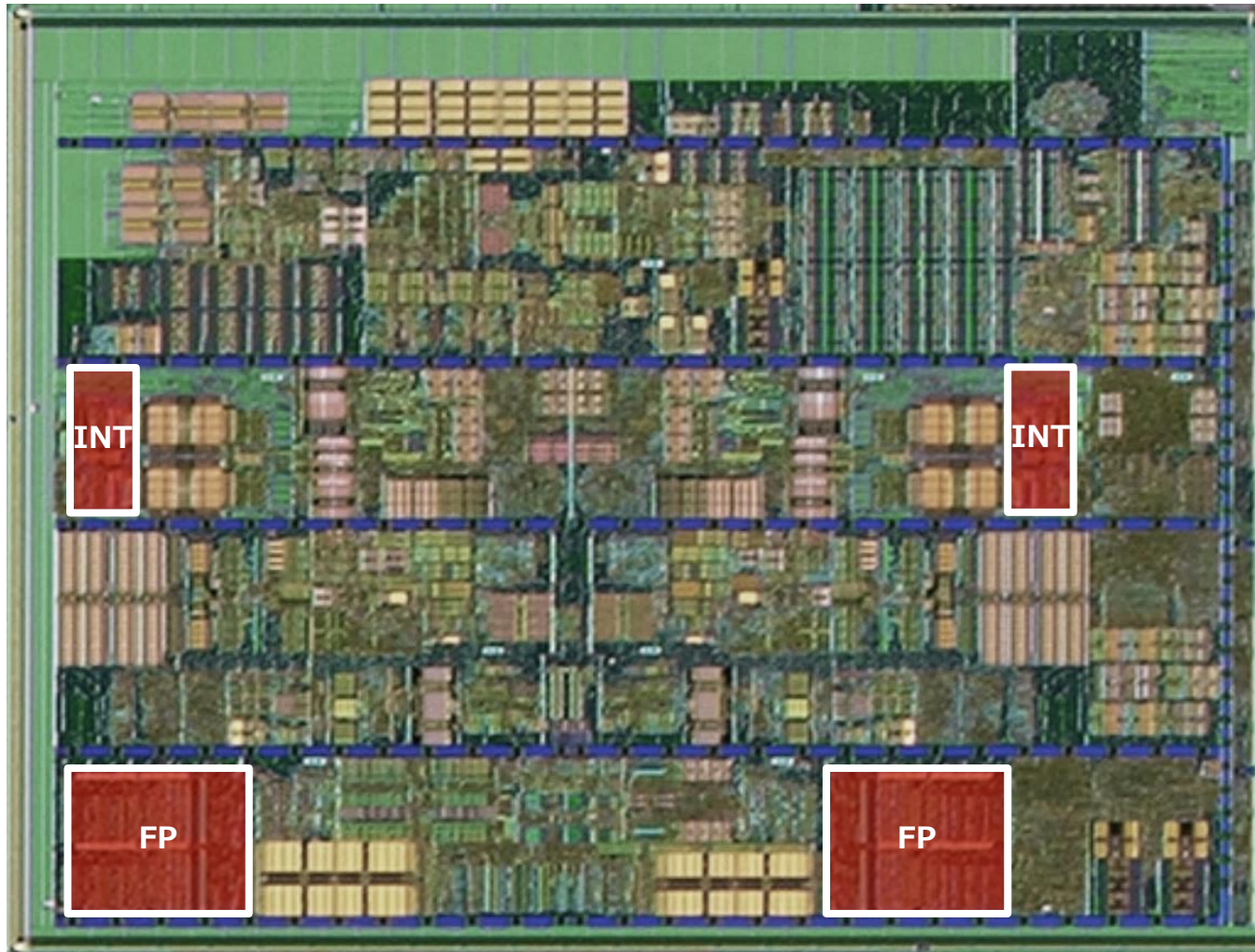
■ GPU

- ◇ 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



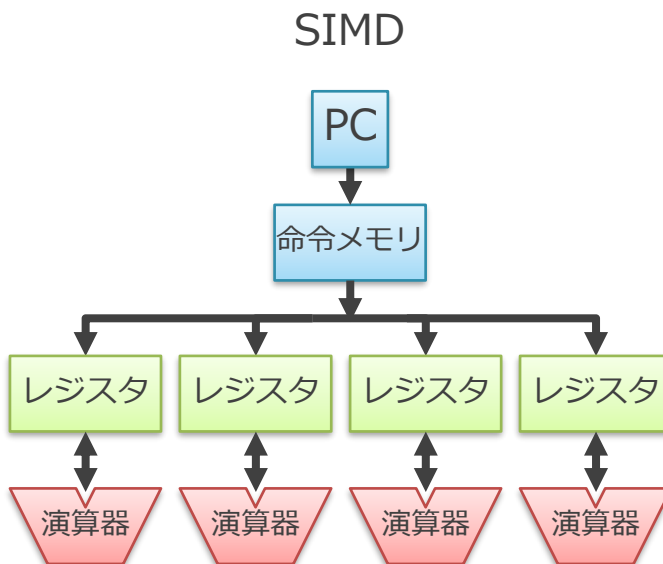
- 演算器（INT/FP）が CPU コア全体に占める割合はわずか

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

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

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	SPMD/SIMD 命令
Intel (GEN)	SIMD	SPMD/SIMD 命令

もくじ

1. SIMD 命令
2. SPMD と SIMT

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

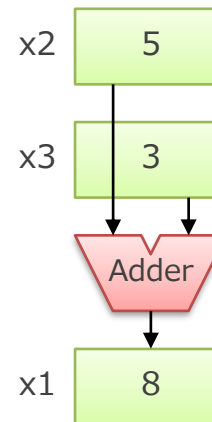
■ まぎらわしい :

- ◇ 「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（スカラー命令）

◇ `add x1 ← x2 + x3`

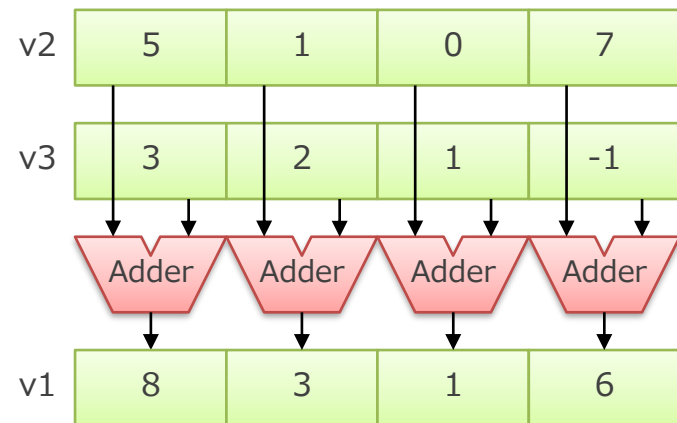


■ SIMD 命令

◇ `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. 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（つらい）

(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
// **コンパイル時にこの関数と等価な結果が得られる SIMD 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

1. SIMD 命令
2. SPMD

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++)  
    for (j = 0; j < 1024; j++)  
        for (i = 0; i < 1024; i++)  
            a[j][i] += b[j][k] * c[k][i];
```

- j の部分のループ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) {  
    j = thread_id; // k だと同じ場所を書いてしまう  
    for (k = 0; k < 1024; k++)  
        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

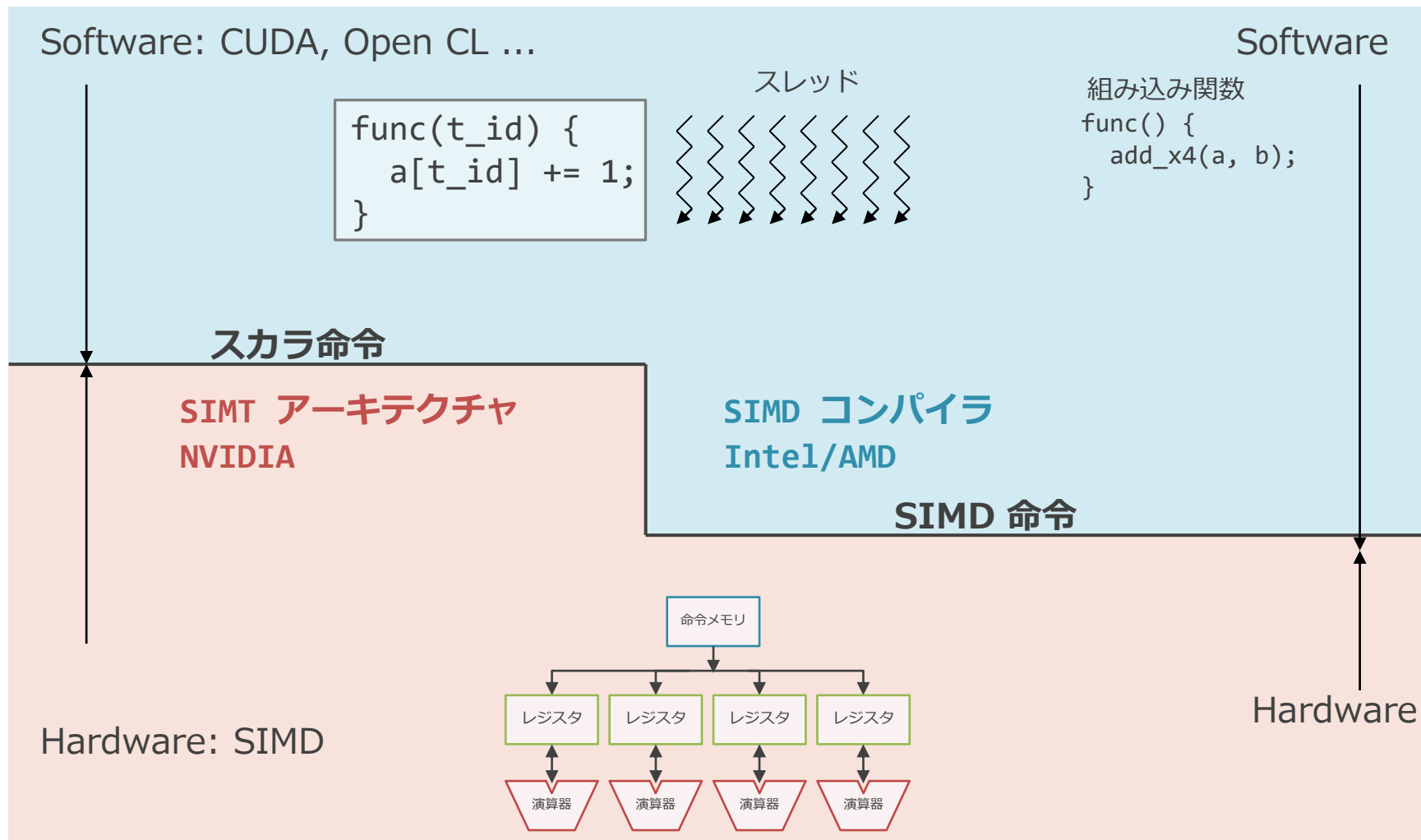
- ◇ コンパイラにより自動で変換
- ◇ 変換された SIMD 命令を実行
- ◇ AMD/Intel の方式

2. Single Instruction Multiple Thread (SIMT) ハードウェアによる実行

- ◇ 命令列自体はスカラー命令で記述
- ◇ 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

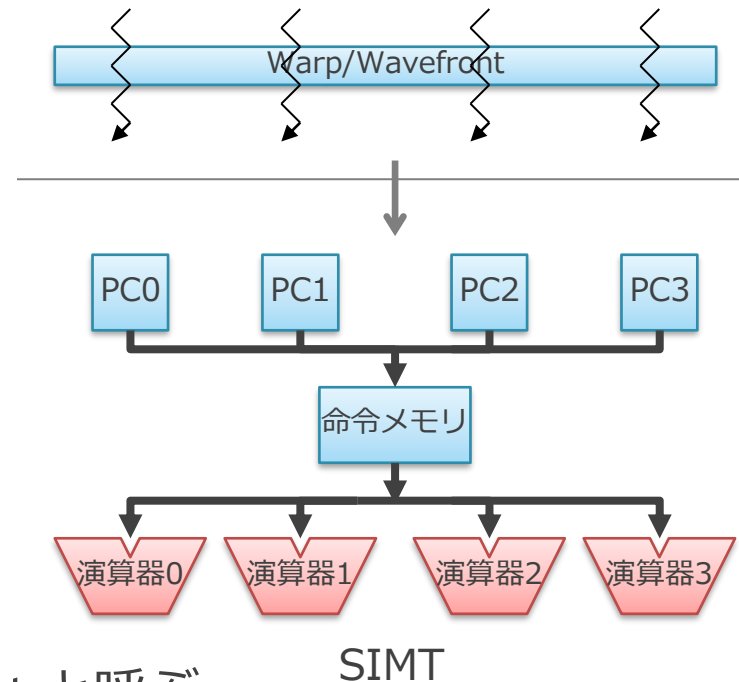
図は Caroline Collange “GPU architecture part 2: SIMT control flow management” より



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 or 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 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
- RISC-V ベースの SIMT プロセッサも提案されている

■ 問題点：

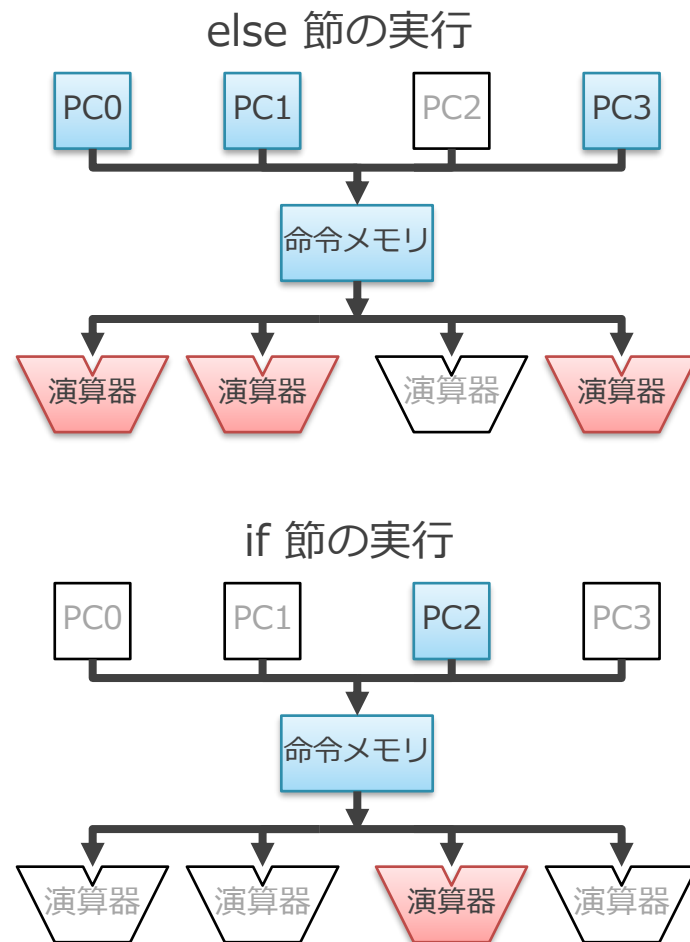
- ◇ スレッド間で違う方向に分岐すると実行速度が落ちる
 - branch divergence と呼ぶ
- ◇ 水平方向の演算ができない

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

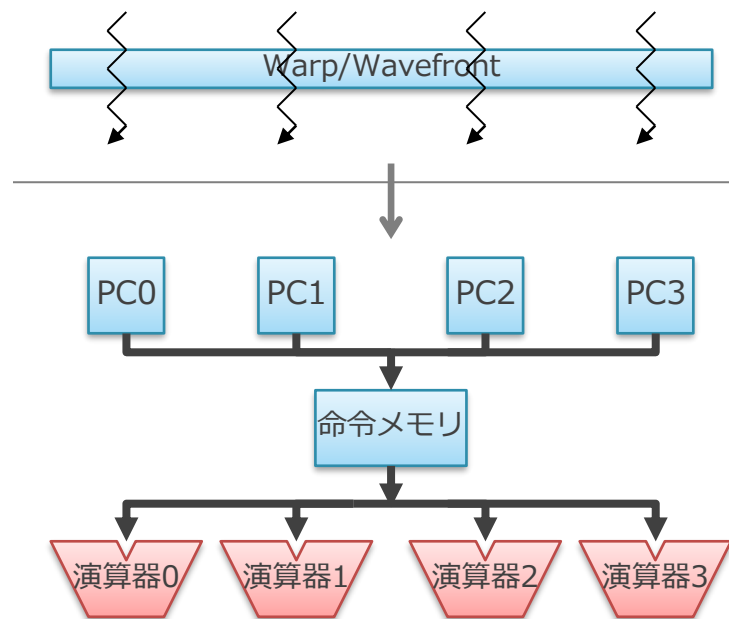
◇ この例では実行速度が $\frac{1}{2}$ に



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...

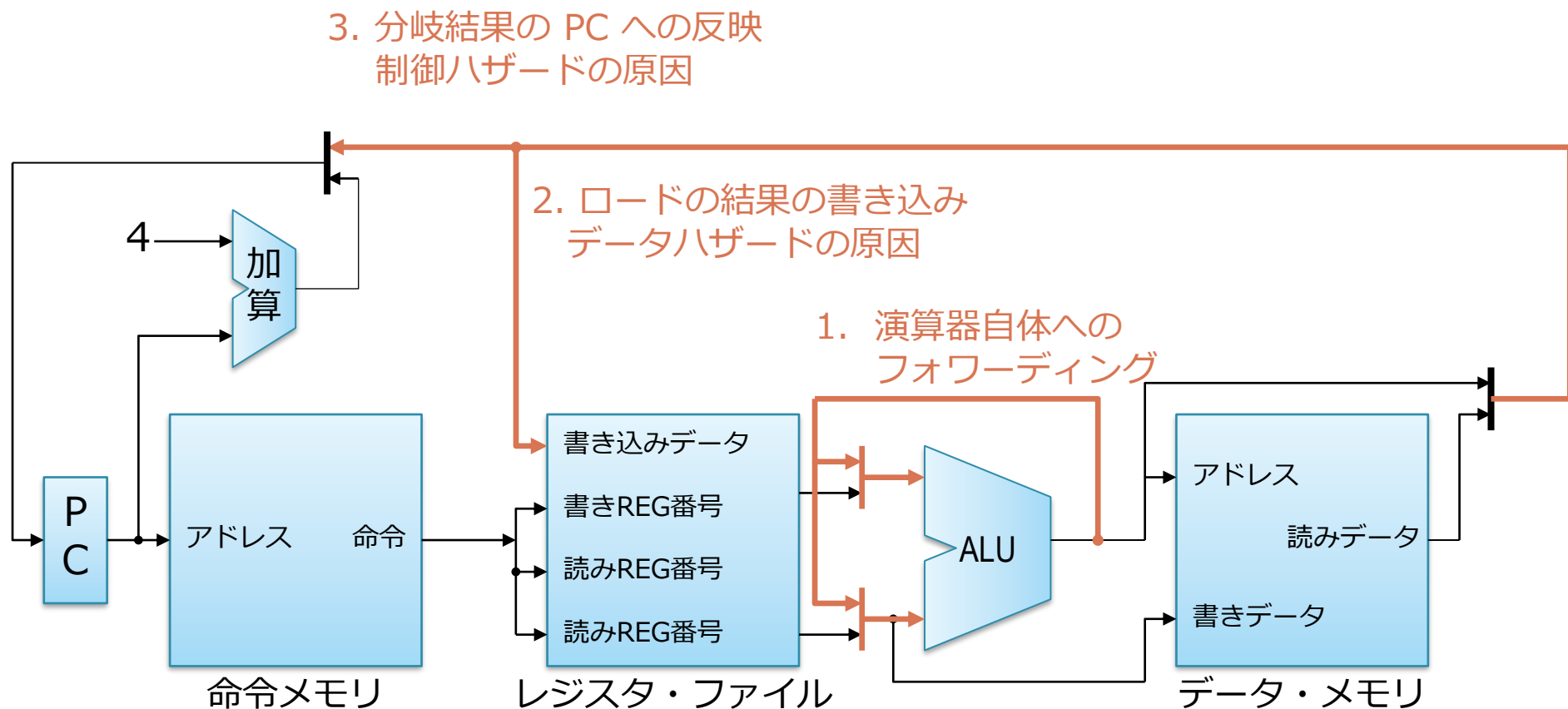
- GPU の場合

- ◇ 上記を一切行わない

- ◇ 全部マルチスレッディングで対処

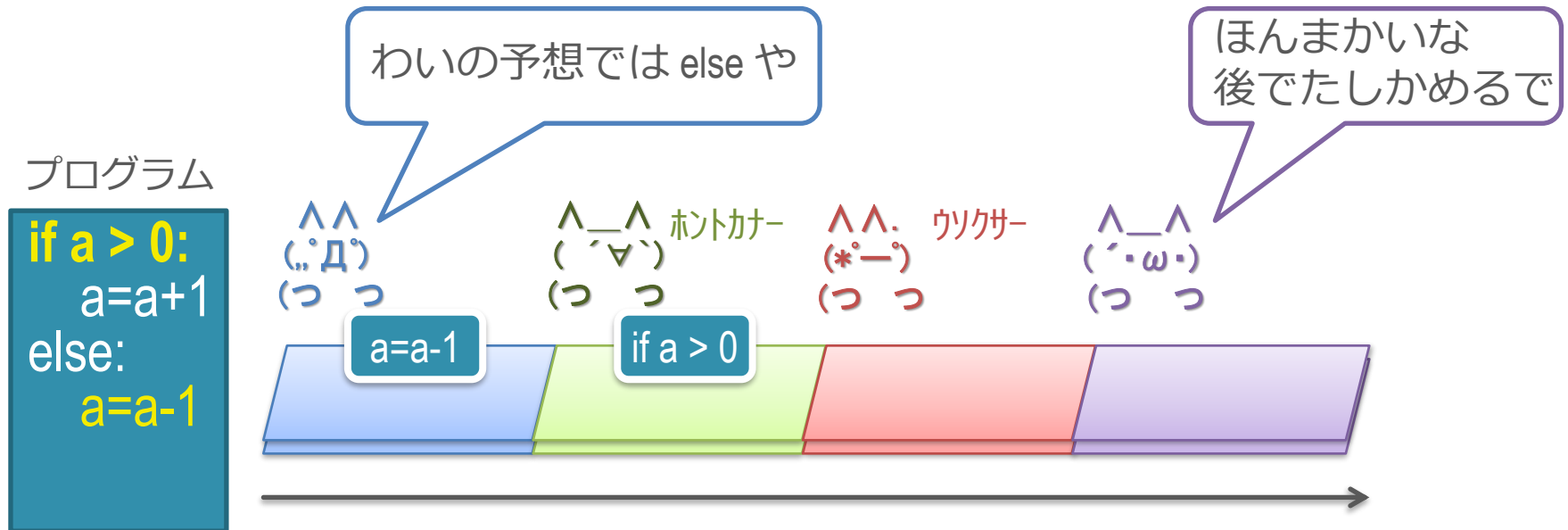
- 動かすべきスレッドは大量にある

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

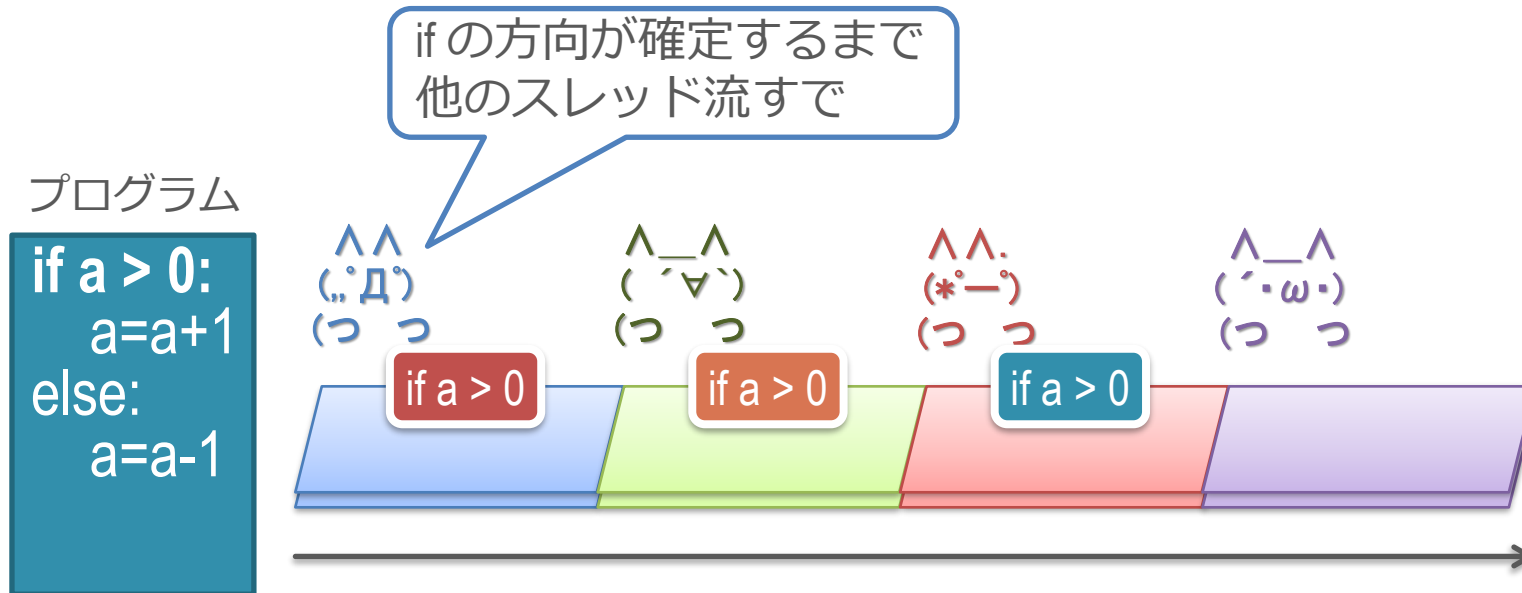
分岐予測



■ 動作

- ◇ 「if a > 0」の結果を予測して、命令を取り込む
 - 前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

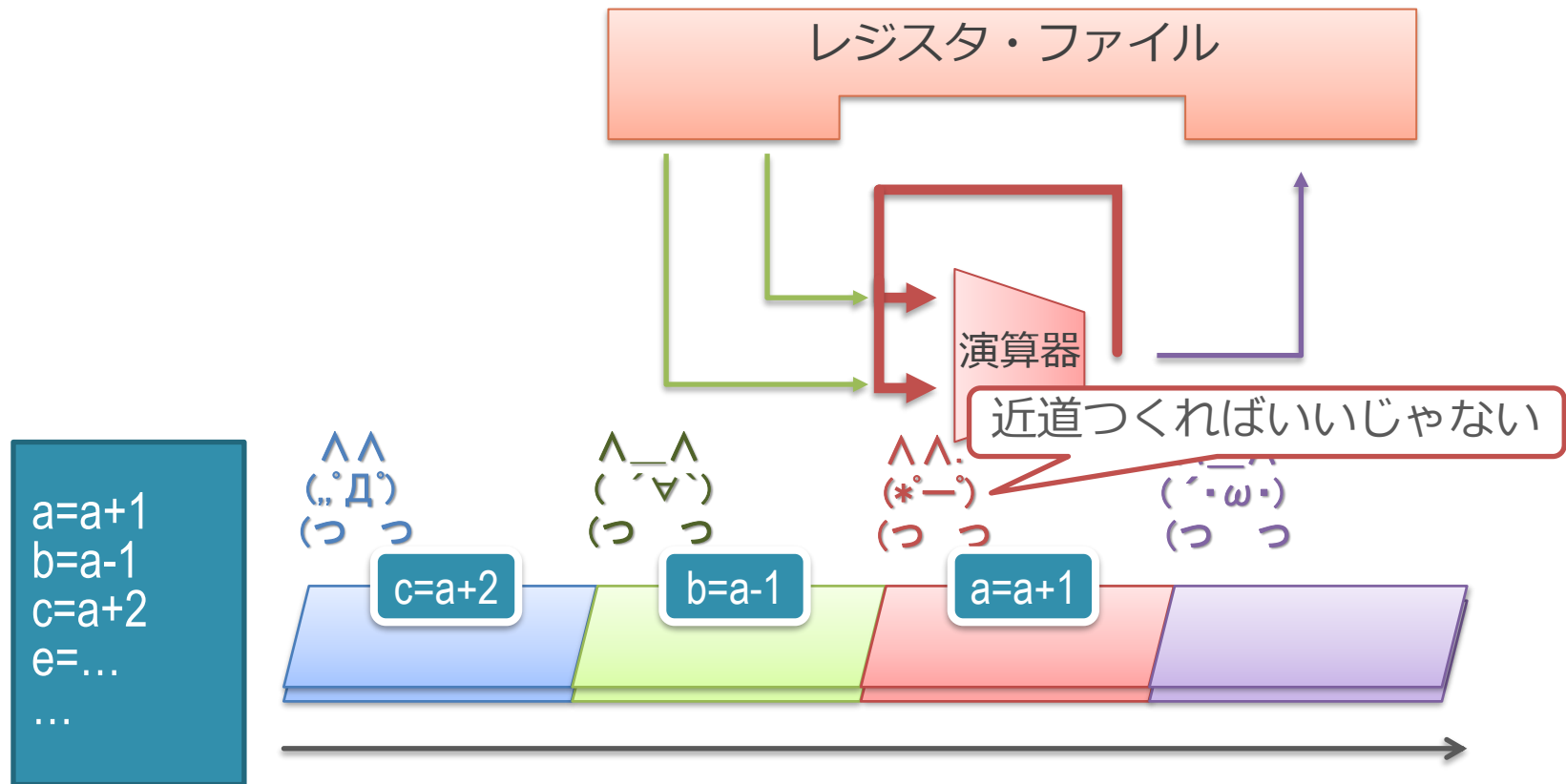
マルチスレッドによる解決



■ 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

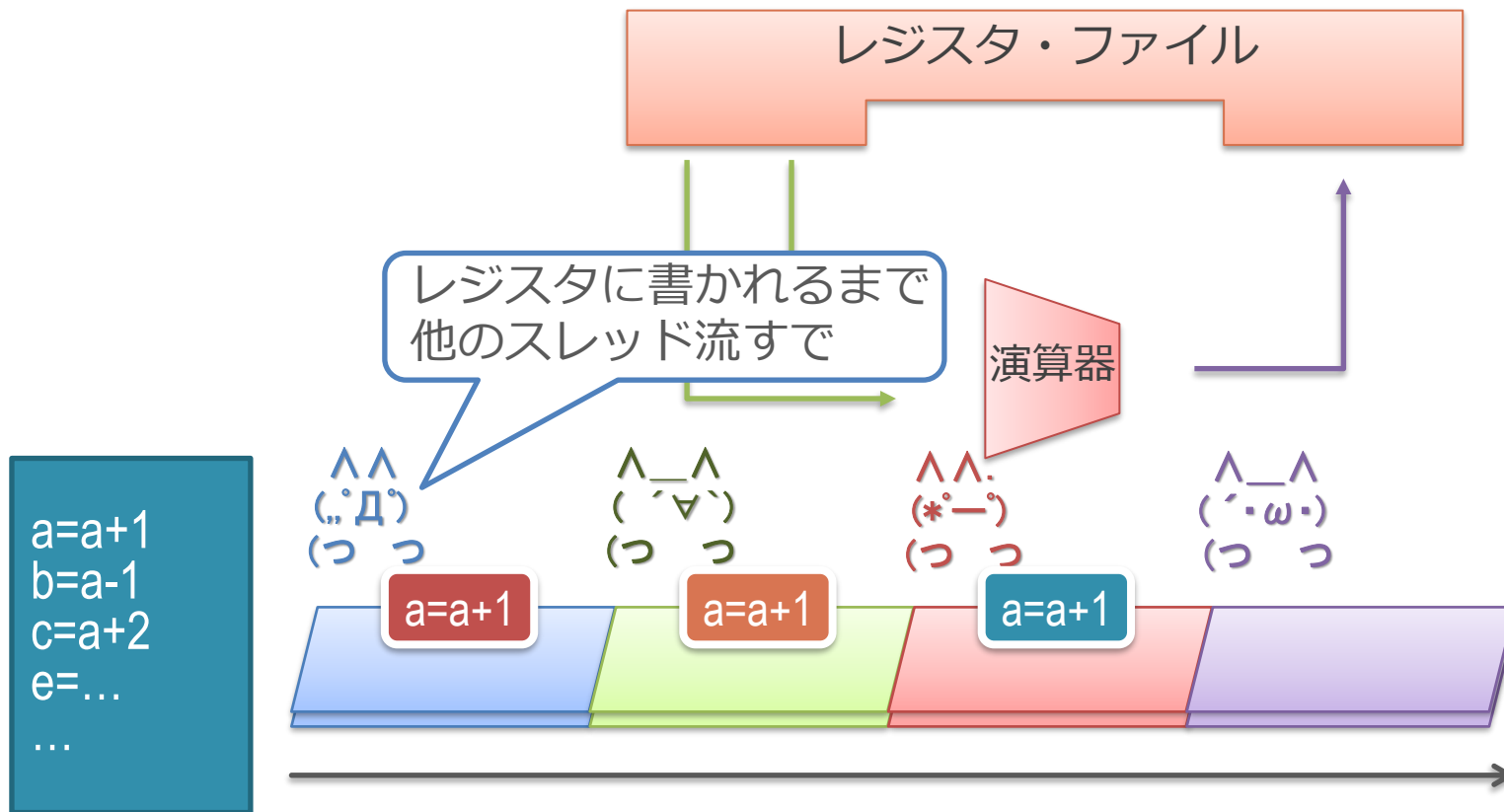
フォワーディング



■ フォワーディング (バイパスとも呼ぶ)

- ◇ $(\wedge \wedge)$ 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

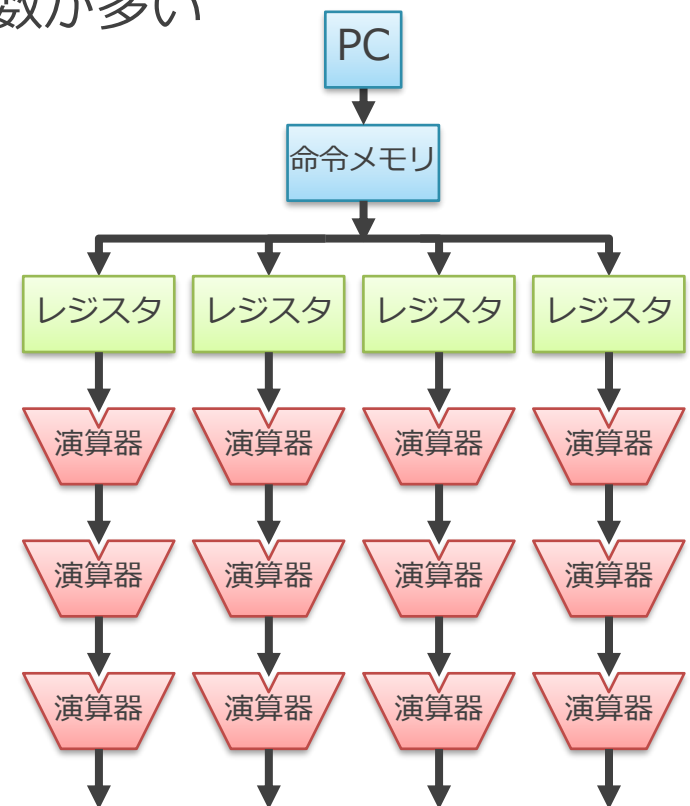
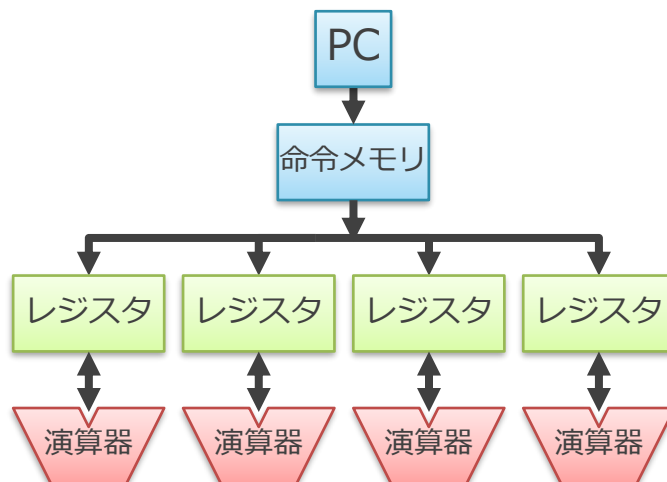
◇ 演算1回ごとに命令を読んだり, レジスタ・ファイルにアクセス

■ 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-Datcenter Performance Analysis of a Tensor Processing Unit, ISCA 2017 より

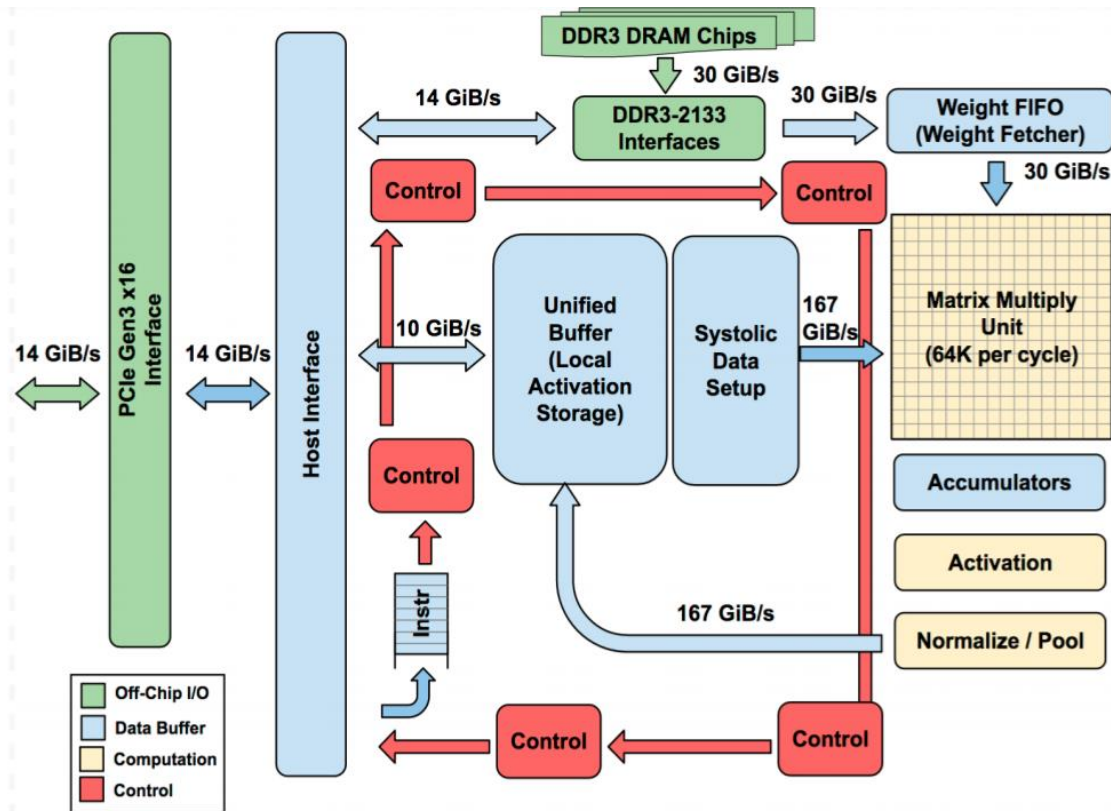


Figure 1. TPU Block Diagram. The main computation is the yellow Matrix Multiply unit. Its inputs are the blue Weight FIFO and the blue Unified Buffer and its output is the blue Accumulators. The yellow Activation Unit performs the nonlinear functions on the Accumulators, which go to the Unified Buffer.

- ISCA2023/MICRO2023 ないしは、次のページで指定するいくつかの論文のうち1つを読んで、その内容を説明すること

（ISCA2024 はまだ論文が落とせないなので、2023 にしました）

- ◇ <https://dl.acm.org/doi/proceedings/10.1145/3579371>
- ◇ <https://dl.acm.org/doi/proceedings/10.1145/3613424>

- Fetch directed instruction prefetching
 - ◇ 最近の CPU のフロントエンドはこれに近い構造を持っている事が多い.
 - ◇ <https://ieeexplore.ieee.org/abstract/document/809439>
- Best-offset hardware prefetching
 - ◇ Data prefetch championship 2 優勝. ARM のサーバー向け最新 CPU にも実際に入ってるぽい.
 - ◇ <https://ieeexplore.ieee.org/abstract/document/7446087>
- Runahead execution: an alternative to very large instruction windows for out-of-order processors
 - ◇ キャッシュミスしたときに Runahead と呼ぶモードに入りプリフェッチ実行を行う.
 - ◇ <https://ieeexplore.ieee.org/abstract/document/1183532>

- Long term parking (LTP): criticality-aware resource allocation in OOO processors
 - ◇ 割と単純に OoO スーパスカラのスケジューリング能力を上げられる.
 - ◇ <https://dl.acm.org/doi/10.1145/2830772.2830815>
- Constable: Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution (ISCA2024 だが arxiv にあったので)
 - ◇ 毎回固定の値を読むロードを削除して性能を上げる. ISCA 2024 best paper.
 - ◇ <https://arxiv.org/pdf/2406.18786>
- Clockhands: Rename-free Instruction Set Architecture for Out-of-order Processors
 - ◇ リネームの必要がない命令セット. 塩谷も共著. MICRO 2023 best paper nominee.
 - ◇ <https://dl.acm.org/doi/10.1145/3613424.3614272>
- これらの論文は, 講義の内容と関係が深いとか, あるいは有名な論文の中から割と単純なものを選んでいます.

レポート課題

■ 形式：

- ◇ 分量は、日本語なら3000文字、英語なら1500ワード程度を目安
 - これより極端に少ない場合は不可です
- ◇ 「自分で」描いた図を2枚以上使って説明すること

■ 注意：

- ◇ 元の論文の一部を、ほとんどそのまま転記あるいは翻訳しただけのものを提出しないでください
- ◇ 生成 AI にまとめさせたりしないで、自分で書いてください

■ 締め切り：

- ◇ 今年の秋に卒業予定の人：8/6（火曜）まで
- ◇ そうでない人：8/13（火曜）まで

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

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

- 教科書にあまり書かれていない最新の技術を知るには、このような講義以外ではどうすれば良いでしょうか。
- ◇ 論文を追いかけるしかないのがツライ

- 初回の方の授業でRISC-Vの拡張が増え続けているというようなお話があったかと思うのですが、拡張を入れた時に、OoOなど既存の機構がちゃんと動くかなどはどのように確かめているのでしょうか？
 - 多くの場合、デコーダと演算器にしか影響しない

- フロントエンド・バックエンドという対比は多くの文脈でよく見られて、文脈ごとに全く違うものを指していますよね。別にだからどうということもないんですが.....

質問や感想

- そこで、LSQのストアの値とメモリの内容が常に一致しているでしょうか。キャッシュでいうライトスルー / ライトバックで言うところになるのかということです。
- 非同期的にコミットパイプが働いて一致しているだろうと思います (ライトスルーの動作)。もし前の文が正しいなら、コミットパイプがメモリ書き込みしており完了フラグの検知が遅れることはないでしょうか？
 - ◇ 投機的な書き込みの結果を確定するまでバッファしており、ライトバックともライトスルーともとれます
 - ◇ 完了フラグの検知が済んでからメモリ書き込みを開始します
 - ◇ 他にキャッシュと違うのは、同じアドレスに対する書き込みの複数のバージョンを同時に保持していることです

- Out-of-Order実行で実行順序が正しくなるように保証するアルゴリズムはかなり複雑でそれが正しいものなのかは簡単にはわからないと思うんですが、その正しさを証明するにはどのような方法が使われているのでしょうか？

- 昔に書かれた教科書が多く、物理レジスタ方式やメモリ依存予測など現在主流の手法があまり書かれていないとのことですが、この分野で直近5年～10年の間に書かれた良い教科書はありますか。

- out-of-orderに対応しようとする、特にメモリアクセスにおいてはhardwareでやるが増えてしまうことを実感した。機能を実現するにあたって、どこまでをhardwareでどこまでをsoftwareでやるのかを決めるのはどのように行うのか気になった。

- LSQの解説で検索するというシチュエーションが多く発生していたが、このように検索を多用するアーキテクチャはタイミング制約の観点から支障が出ないのだろうか。優先順位付きで全エントリを検索するという操作はクリティカルパスがとても長そうな感じがした。
- ◇ 連想検索を使わない方法や、検索対象を絞る方法が提案されている

- 質問: ROBがオーバーフローにしないように設計しているかと思っているけど、コミットパイプが非同期でROBを監視しているから、どうやってオーバーフローを防止するのですか？
- ◇ ROB のエン트리使用数のカウンタをフロントエンドで監視して、フルになったらフェッチを止めます