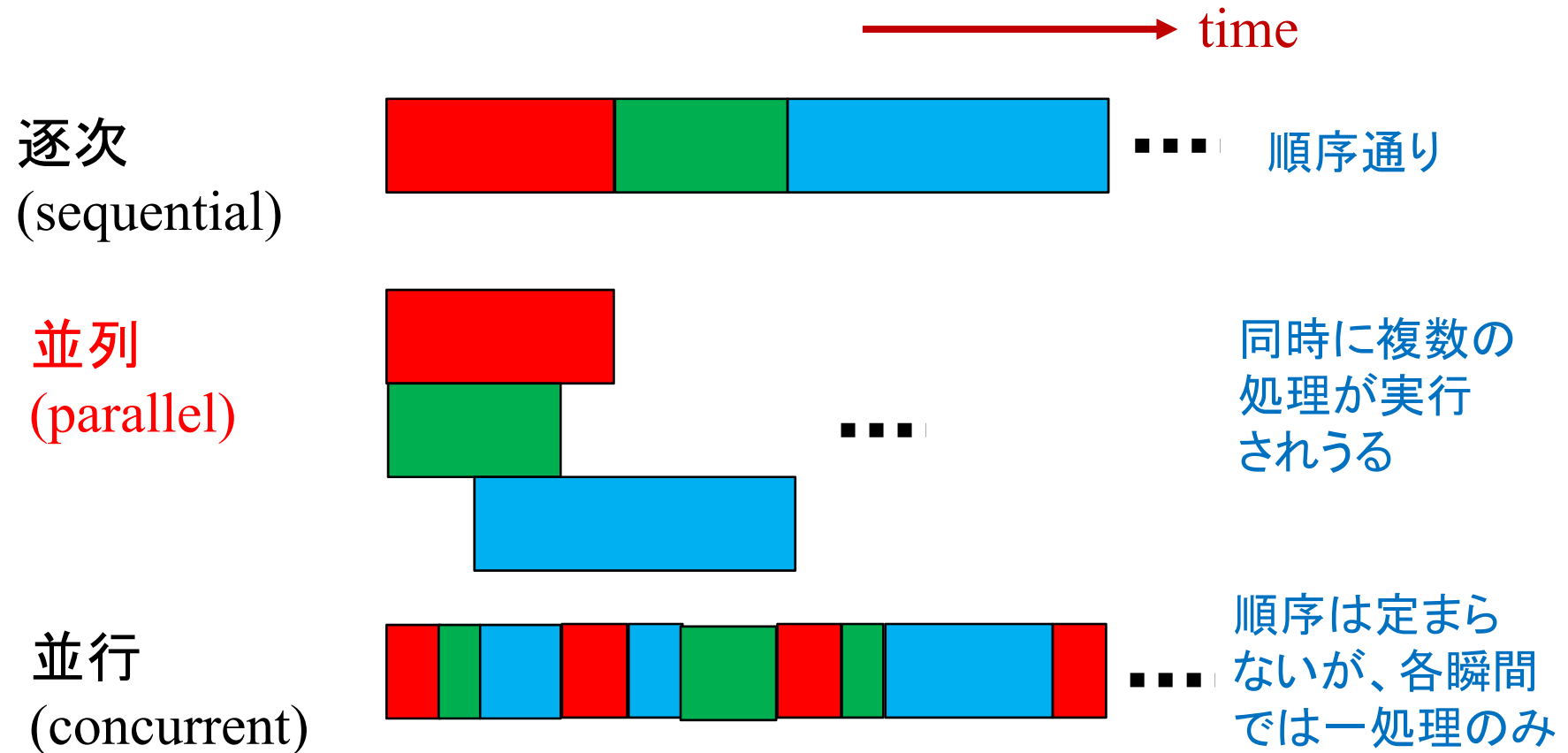

第10回
2019/01/23
様々な並列性

逐次・並列・並行

- ある複数の処理(機械語命令 or 関数 or プログラム)があるときに、それらがどのようなタイミングで実行されるか？

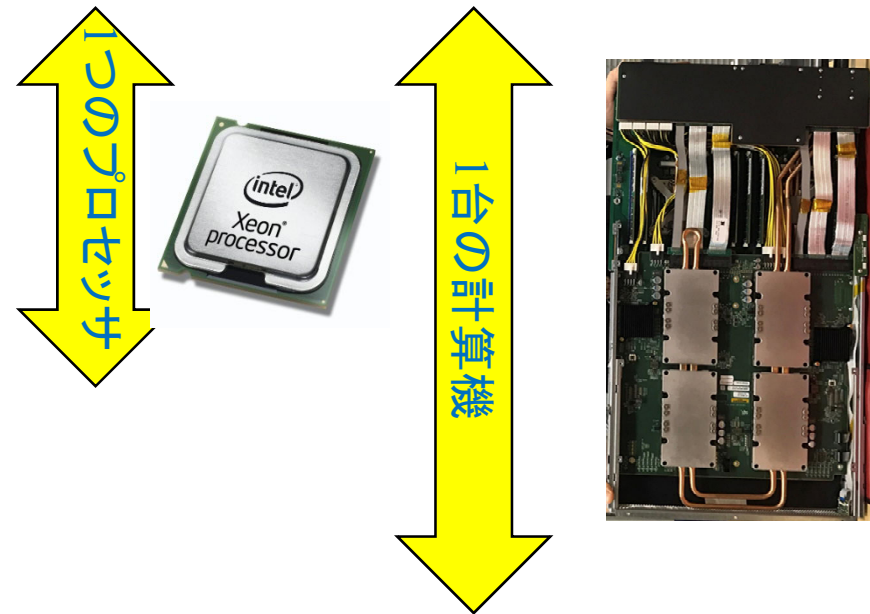


全体の実行時間を短縮させるためには、並列実行が必要

さまざまな並列性

現代のプロセッサ・計算機システムには、さまざまな並列性がある

1. 機能ユニット間の並列性
 - パイプライン処理により活用
 - 隣接した命令間の実行は重なる
2. SIMD並列性
3. ハードウェアスレッド並列性
4. マルチコア並列性
5. マルチプロセッサ並列性
6. 異種マルチプロセッサ並列性
 - CPUとGPUなど
7. マルチノード並列性



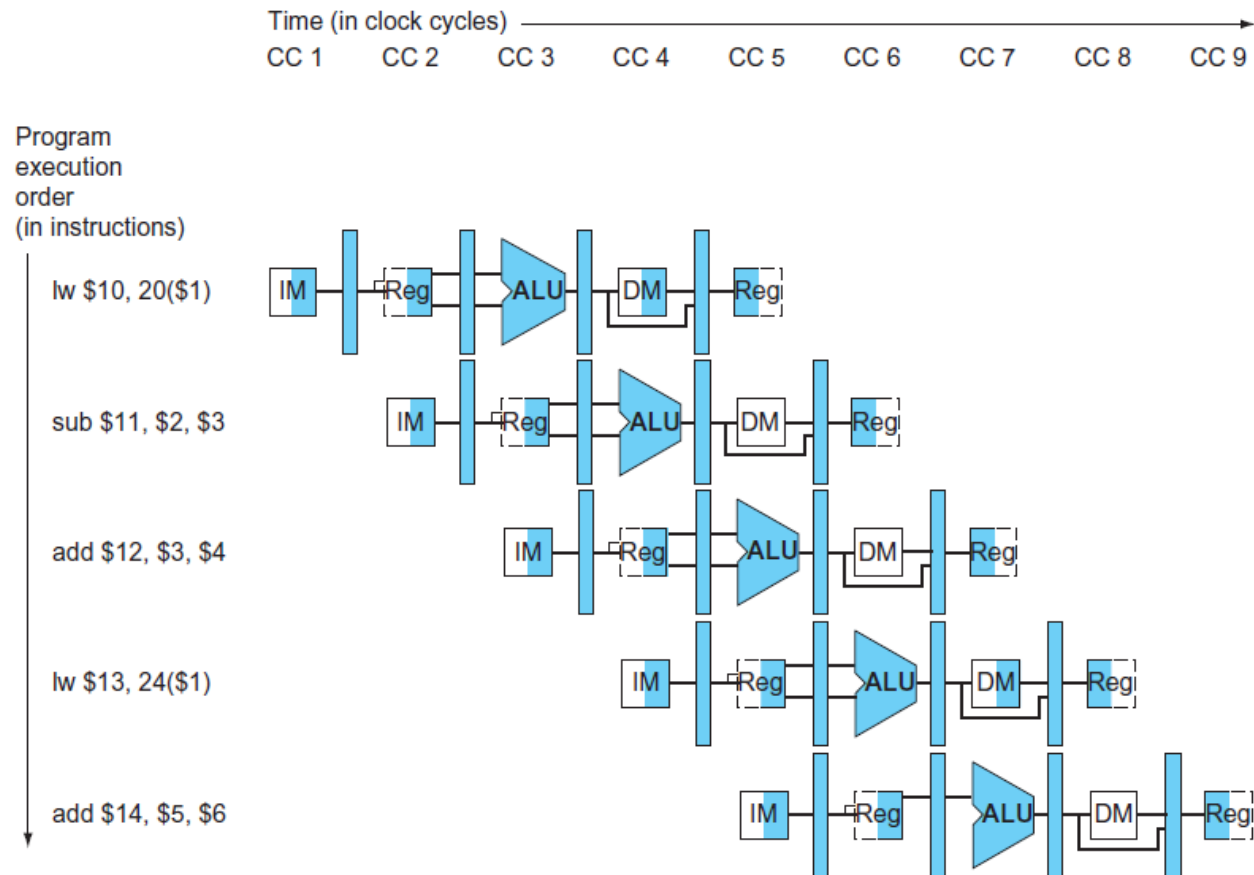
ソフトウェアへの影響もそれぞれ異なる

TSUBAMEスパコンなどでは、これらすべてを活用

…なぜこんなに複雑？ → 一つの技術だけでは限界があるから

例: パイプラインの段数を無限に増やしても、ハザードが多くなり
期待通りの性能向上が得られない

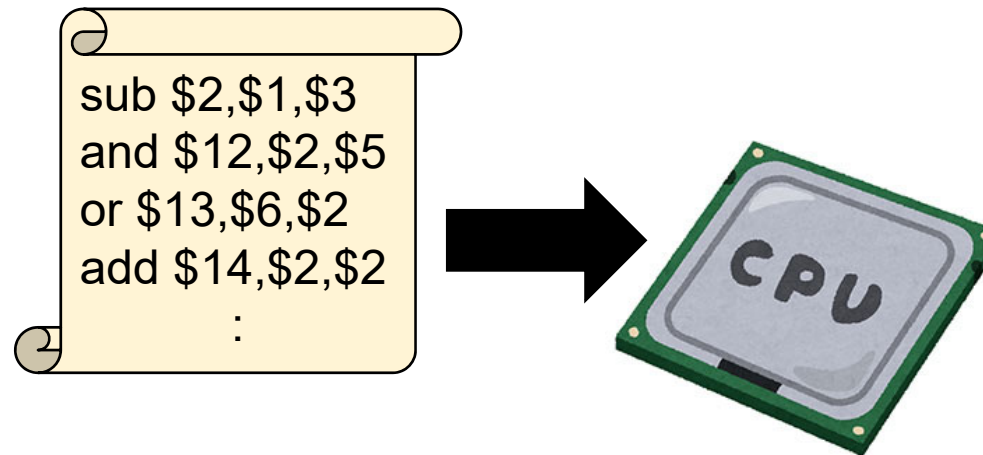
1. 機能ユニット間の並列性



- パイプライン処理により、連続した命令1～5が時間的に重なって実行 (並列実行)
- 機能ユニットが複数あるから可能に

機能ユニット間の並列性:ソフトウェアへの影響

- パイプライン処理の前提:機械語のソフトウェアには(原則)変更しなくてよい
例:パイプラインのなかった時代のソフトウェアをも、高速に実行
- 並列処理に伴う問題は、プロセッサ内部の責任
 - 命令間に依存関係(ハザード)があったら、内部で解決

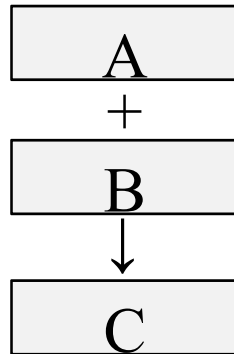


- しかし、2005年ごろまでには限界 → それ以降のプロセッサは、これ以外の並列性も備える

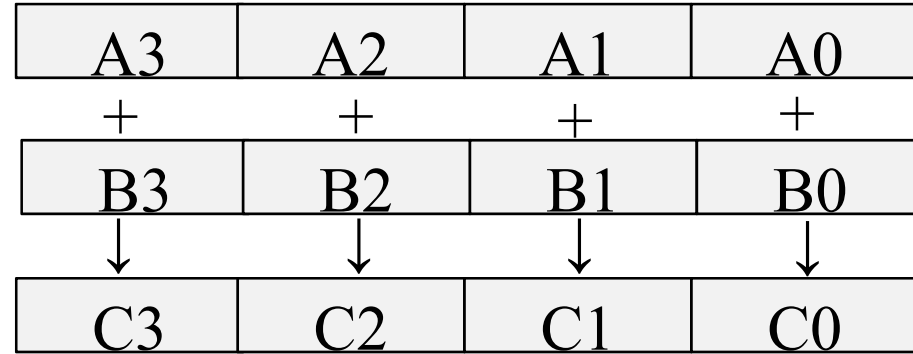
2. SIMD並列性

- SIMD (Single Instruction Multiple Data)
- 一つの機械語命令で、複数のデータの演算を行う

普通に加算命令



SIMD加算命令の例



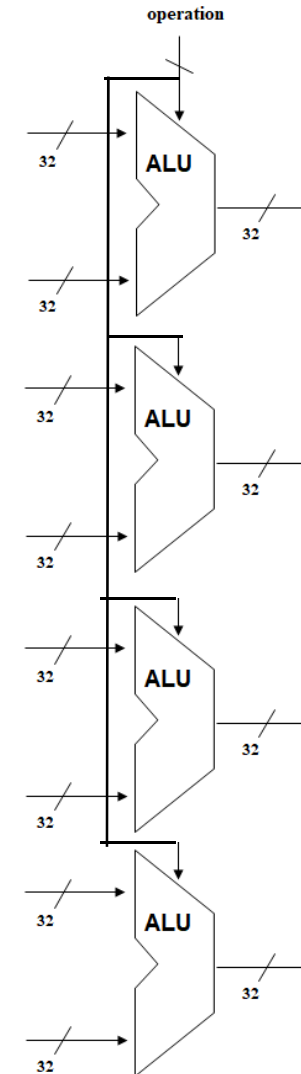
- 新しい機械語命令が必要
- 複数データの同時加算、乗算、論理演算、シフト...
 - int, float, double...それぞれ
 - 最近は、一命令で乗算と加算を行う機能も(FMA/Fused Multiply Add)
 - $\$w1 = \$w1 + \$w2 * \$w3$ (3オペランドの場合)
- 複数データの同時メモリ読み書き命令

SIMD並列性: 必要なハードウェア

- 長いレジスタ(**ベクトルレジスタ**とよく呼ばれる)の集合
 - ひとつあたり128, 256, 512bit (アーキテクチャによる)
 - たとえば128bitレジスタは、以下のように利用
 - 16bit整数 × 8, 32bit整数 × 4
 - 32bit float × 4, 64 bit double × 2
- SIMD演算可能なALU
 - たとえば32bit整数 × 4の演算は、ALU 4つ分
- 新命令に対応可能なデコーダ

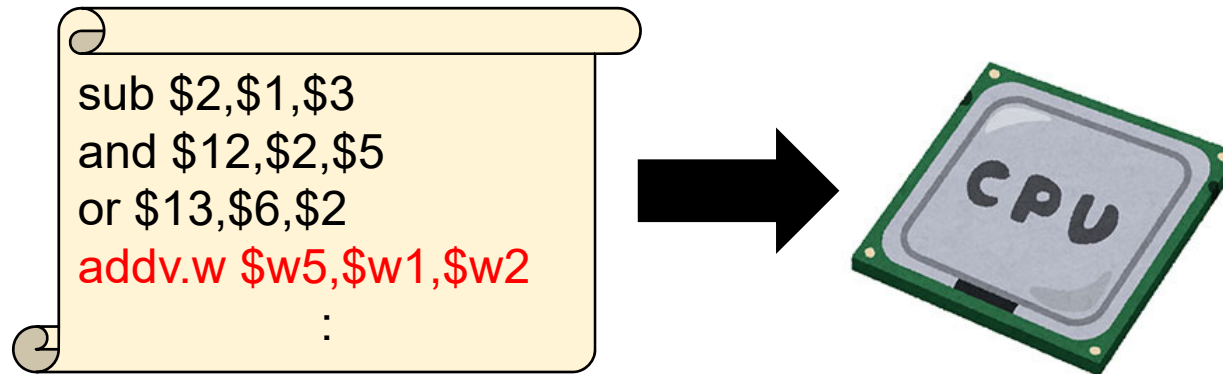
Intelの場合、下記のようにSIMD命令が追加されてきた

- MMX (1997)
- SSE (1999), SSE2, SSE3, SSE4.1, SSE4.2
- AVX (2011), AVX2, AVX512
 - AVX512は、512bitレジスタ



SIMD並列性:ソフトウェアへの影響

- 新しいSIMD命令を使わないと、高速化の効果を得られない！



ソフトウェア書き換えはユーザの責任

- 独立で同じ演算種類 (たとえばadd4つ)の演算を見つける
- ベクトルレジスタにデータを準備...

近年は、SIMD命令を出力可能なコンパイラも。ただし単純なケースに限る

コンパイル処理のイメージ

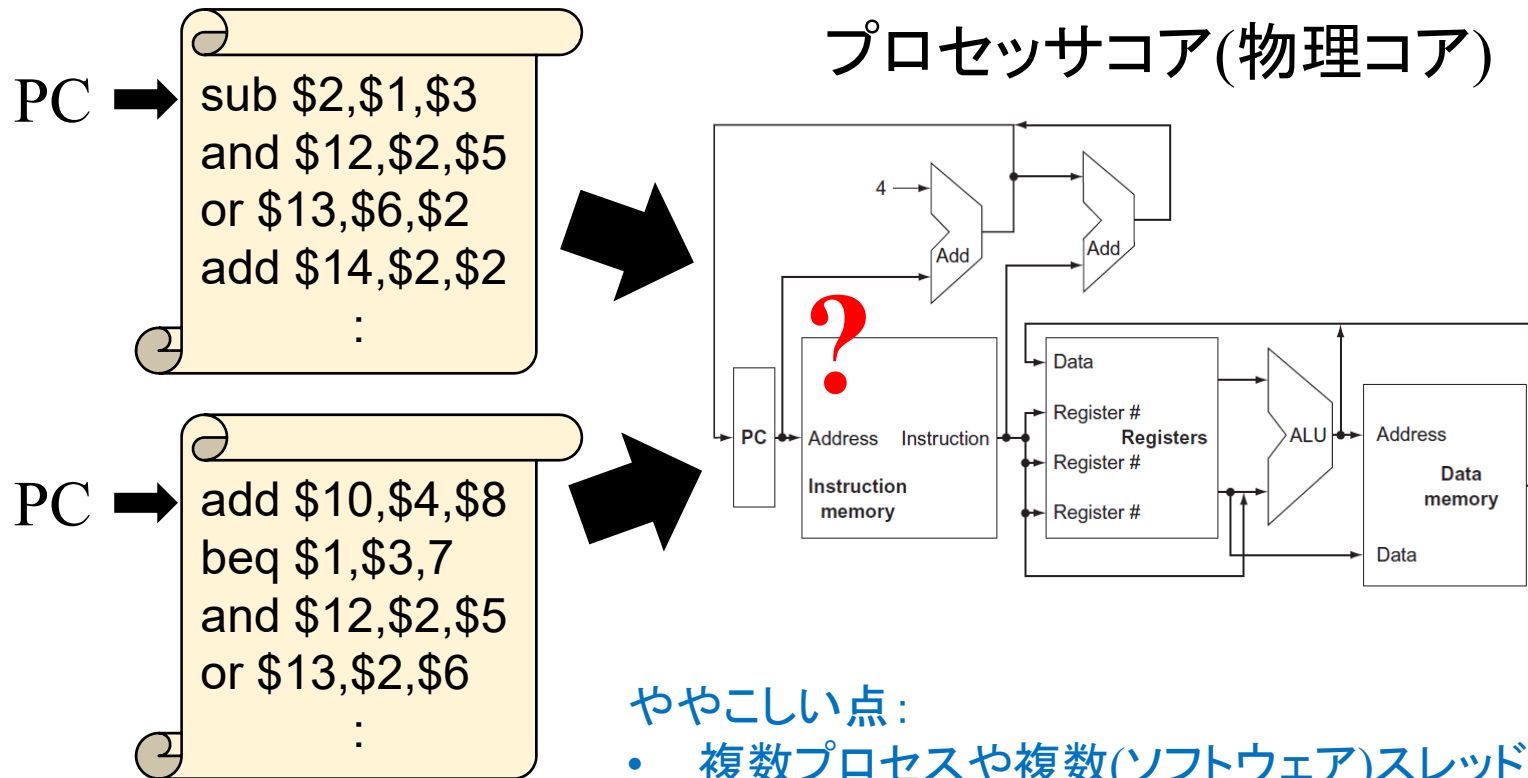
```
for (i=0; i<100; i++)  
  C[i] = A[i]+B[i];
```



```
for (i = 0; i<100; i+=4)  
  C[i~i+3]=A[i~i+3]+B[i~i+3];
```


3. ハードウェアスレッド並列性

- 物理的な1プロセッサコアが、複数の命令列を同時に(並列に)処理可能
 - 2つの場合が多い(特にIntel)が、4/8個の命令列の場合も
 - この場合、1つの**物理コア**に対して2/4/8個の**論理コア**が対応する、と言える
 - この技術をIntelではHyper-Threadingと呼んでいる



ややこしい点:

- 複数プロセスや複数(ソフトウェア)スレッドと、目的は近いが別物。ここではハードウェアによるもの

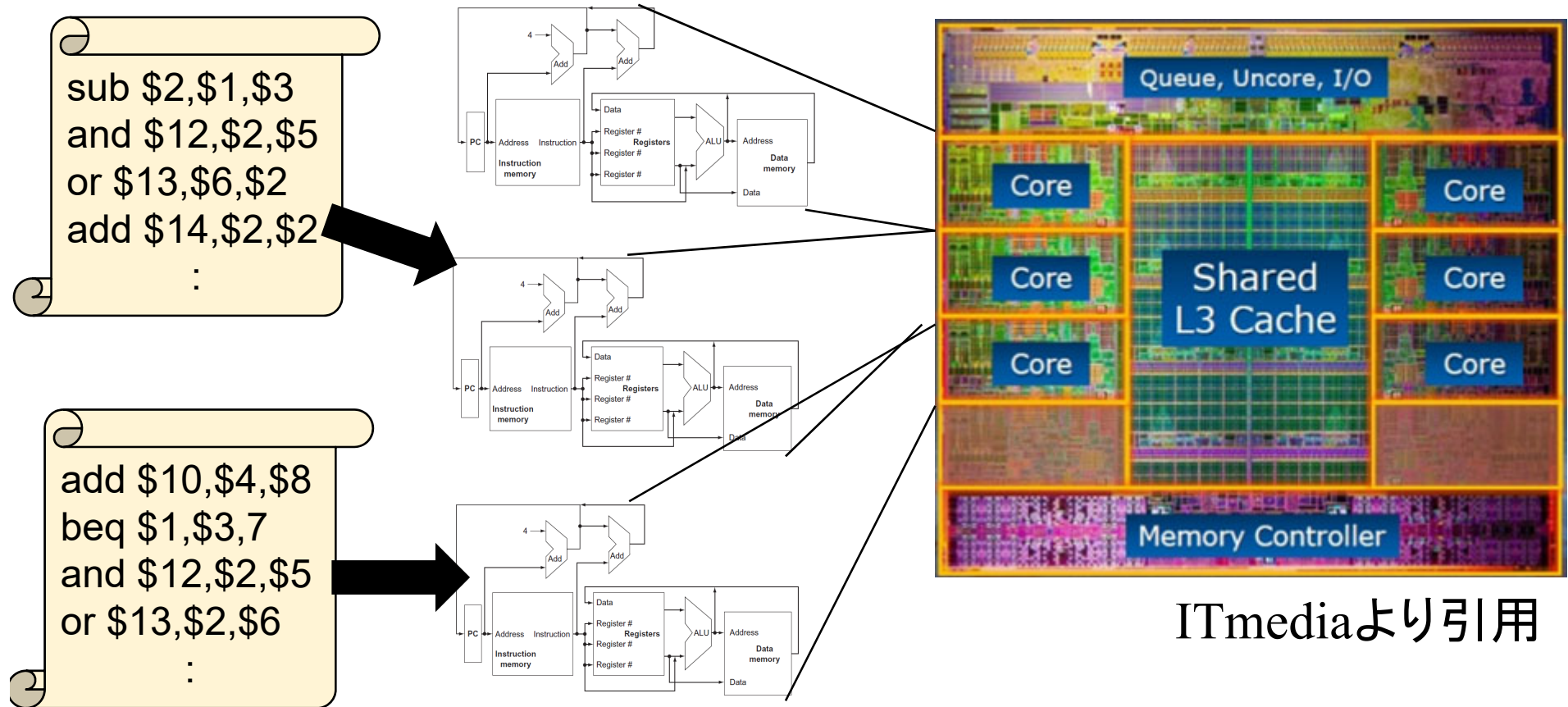
ハードウェアスレッド並列性の発想と 必要なハードウェア

- 現在のプロセッサは多数の機能ユニットを持ち、(パイプラインを駆使しても)常にすべての機能ユニットを埋め尽くすのは困難
 - 例: あるプログラムでは、整数ALUは忙しいが浮動小数点ALUはひま
 - 例: キャッシュミスすると100クロック以上ひまになってしまう
- 複数の命令列を並列実行するために必要なユニットの個数を増やす(たとえば2倍)
 - PC
 - レジスタファイル
 - デコード処理のユニット
- それ以外の個数はそのまま
 - ALU → 2つの命令列で、待合いながら使う
 - メモリ → 複数の読み書きがオーバーラップできる仕組みに

Intelプロセッサでは2002年ごろから採用

4. マルチコア並列性

- プロセッサ内に複数の(物理)コアを搭載する



ITmediaより引用

- 写真のプロセッサIntel Core i7-3960Xでは、6(物理)コアのそれぞれが2論理コアを持つので計12論理コア

マルチコア並列性のハードウェア (1)

- コアの間では原則、機能ユニットの共有は起らない
- 例外はメモリ:どのコアも、メモリの任意の場所をアクセスできる

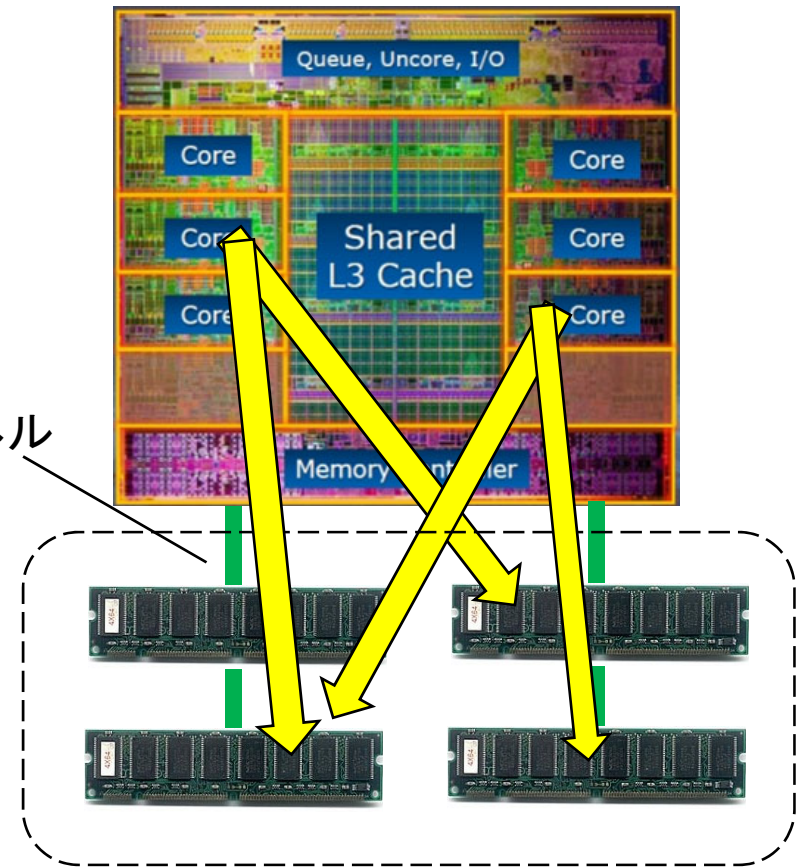
→ 共有メモリ (shared memory)と呼ぶ

- 各コアが実行するメモリ読み書き命令のうち、キャッシュミスが起こると、そのデータはメモリチャンネルを通る
- メモリチャンネルを複数設けるなど、様々な工夫

→ それでも、メモリのスループットは有限であり、混雑により速度低下が起こることはある

nコアを用いても、n倍未満の速度向上しか得られない理由の、大きな一つ

メモリチャンネル



メモリ:コア間で共有されている

マルチコア並列性のハードウェア (2)

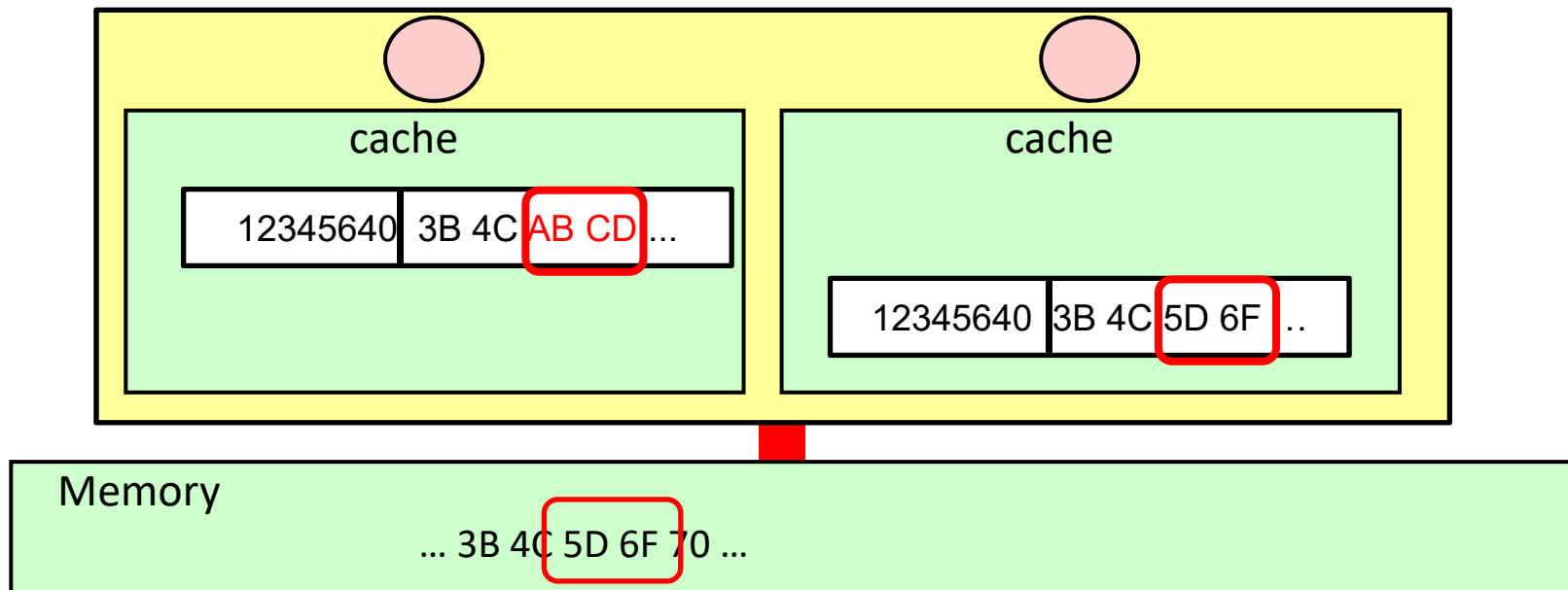
キャッシュの構造はどうなっている？

Intelの場合、

- L1/L2キャッシュはコアごと
- L3キャッシュはコア間共有

キャッシュの存在を(速度向上以外)ユーザになるべく見せない

→ データの一貫性が必要。[キャッシュコヒーレンスプロトコル](#)によって実現



同アドレスのデータが、複数キャッシュに複製される可能性
→ Write invalidateなどの方法で対応

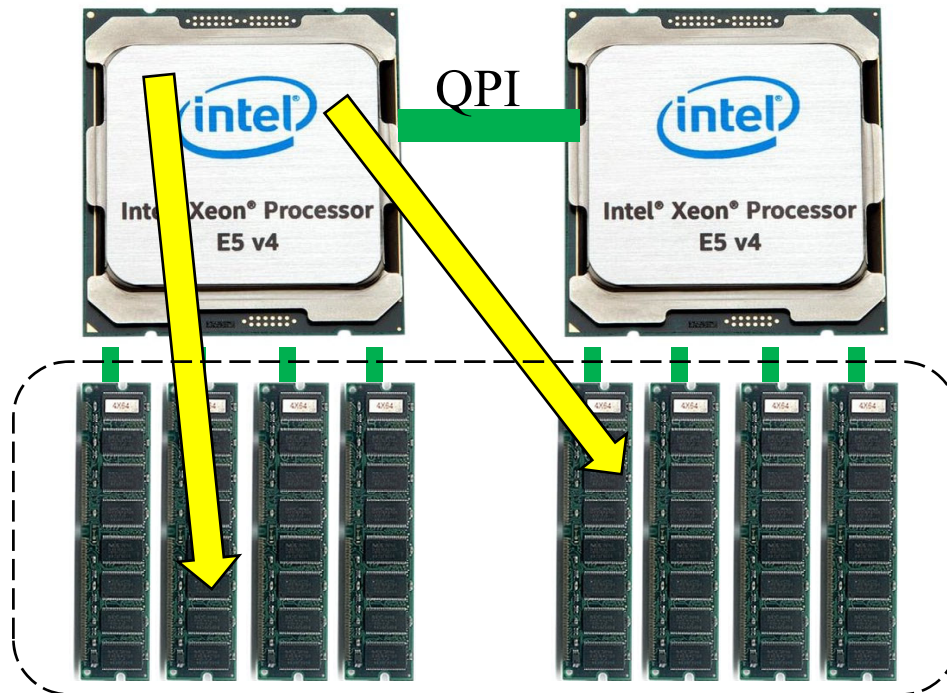
プロセッサ(Intel)の並列性の歴史

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/Speculation	Cores/Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103 W
Intel Core	2006	2930 MHz	14	4	Yes	2	75 W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	1	87 W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77 W

- SIMD並列性は1997ごろから
 - 当初はMMX (MultiMedia eXtension)...画像・動画処理向けとして
- Hyper-threadingは2002ごろから
- マルチコア並列性は2005ごろから

5. マルチプロセッサ並列性

- プロセッサ種類によっては、1台の計算機に複数のプロセッサを搭載できる
Intelの場合
 - Core iシリーズ: × (1Processorまで)
 - Xeon シリーズ: ○ (2P対応、4P対応、8P対応)
- どのコアも、メモリの任意の場所をアクセス可能 (共有メモリ)
 - 一貫性もokだが、やはりメモリのスループットの限界の影響あり

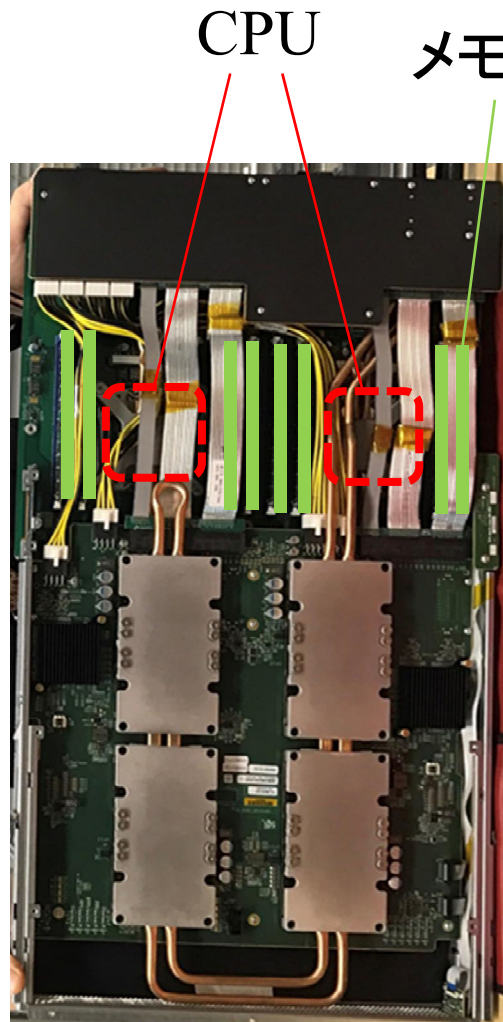


メモリ: 全プロセッサの全コア間で共有

共有メモリではあるが、
プロセッサをまたぐメモリ
アクセスはやや速度低下

このような構成を**NUMA**
(Non-Uniform Memory Access)
と呼ぶ

TSUBAME3の計算ノード(GPU除く)を見てみよう



- CPUはIntel Xeon E5-2680 v4 × 2個
- Xeon E5-2680 v4
 - 14物理コア × 2論理コア(Hyperthread)
 - AVX-512対応
 - 2.4GHz(通常), 1.9GHz(AVX-512時)
 - 3次キャッシュ 35MB
 - DDR4メモリチャネル 4本
- メモリ: 各メモリチャネルに、32GB DDR4-2400 (19.2GB/s) DIMMを1個ずつ
容量: $32\text{GB} \times 4\text{チャネル} \times 1\text{個} \times 2\text{プロセッサ} = 256\text{GB}$
バンド幅: $19.2\text{GB/s} \times 4\text{チャネル} \times 2\text{プロセッサ} = 153.6\text{GB/s}$
- 28物理コア/56論理コアが、このメモリを共有

TSUBAME3の計算ノードは何Flopsか？

- Flops:秒あたりの浮動小数点演算回数の単位
- ここでは、理論性能 (実測性能ではない) を考える

Xeon E5-2680 v4 1個のFlops (**double/FP64**)

14 (コア) × 8 (SIMD演算) × 2 (乗算+加算) × 1.9 GHz = 425.6GFlops

- 通常時クロックではなく、AVX時クロック1.9GHzで計算
- この式には、論理コア数は入らない (ハードウェアスレッドはALUを増やさないので)

同様に**single/FP32**のFlopsは、

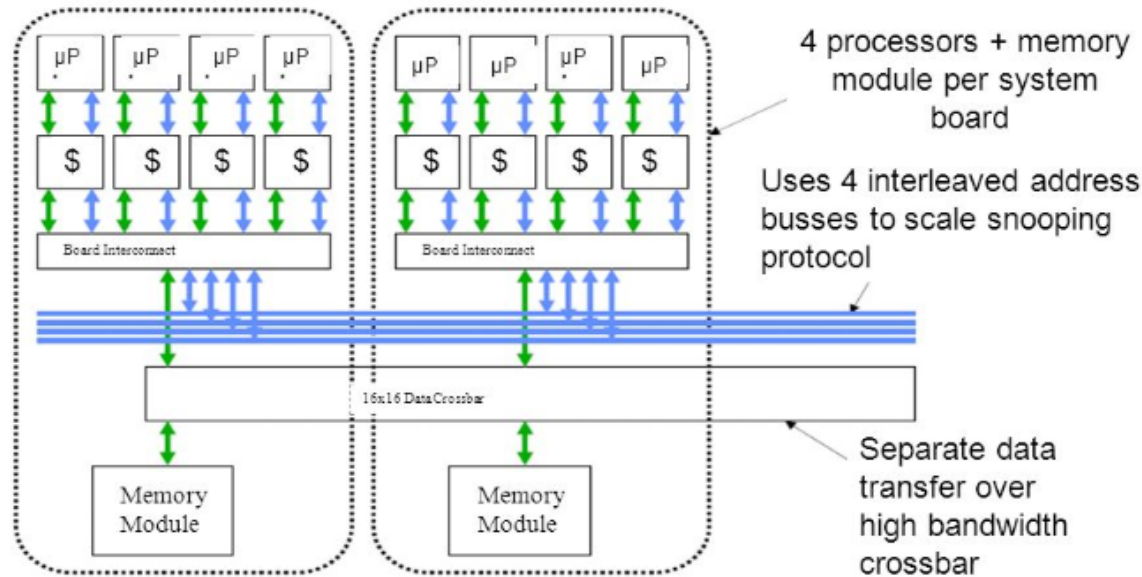
14 (コア) × 16 (SIMD演算) × 2 (乗算+加算) × 1.9 GHz = 851.2GFlops

1計算ノードには2個のプロセッサ → 851.2GFlops (FP64), 1702GFlops (FP32)

TSUBAME3.0には540台の計算ノードがあるので、0.46PFlops (FP64)
12PFlopsには及ばない? → 残り96%はGPUでまかなっている

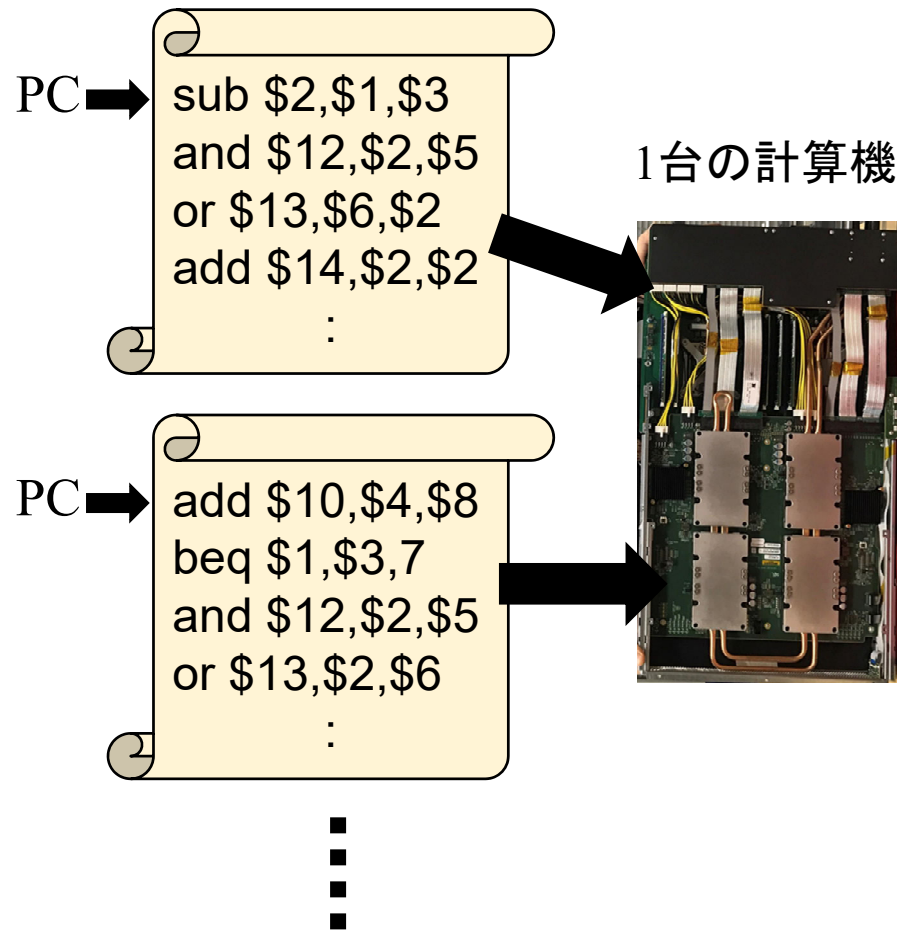
過去のマルチプロセッサ計算機

- 1990年代前半から、マルチプロセッサは存在した
 - 実は、マルチコア・ハードウェアスレッドより古い
- Sun Enterprise 10000 (1997)
 - 64個のUltra SPARCプロセッサ (1プロセッサあたり1コア)がメモリを共有



任意のプロセッサから、任意のメモリへのアクセス性能が均一
→ これは均一に遅い傾向にあり、**現在はマルチコアおよび、前述の
NUMA型マルチプロセッサが普及**

(3.~5.) ハードウェアスレッド・マルチコア・マルチ プロセッサ並列性の、ソフトウェアへの影響



- (プロセッサ数×コア数×ハイパースレッドの並列数)以上の**命令の流れ**が存在しないと、資源を埋められない

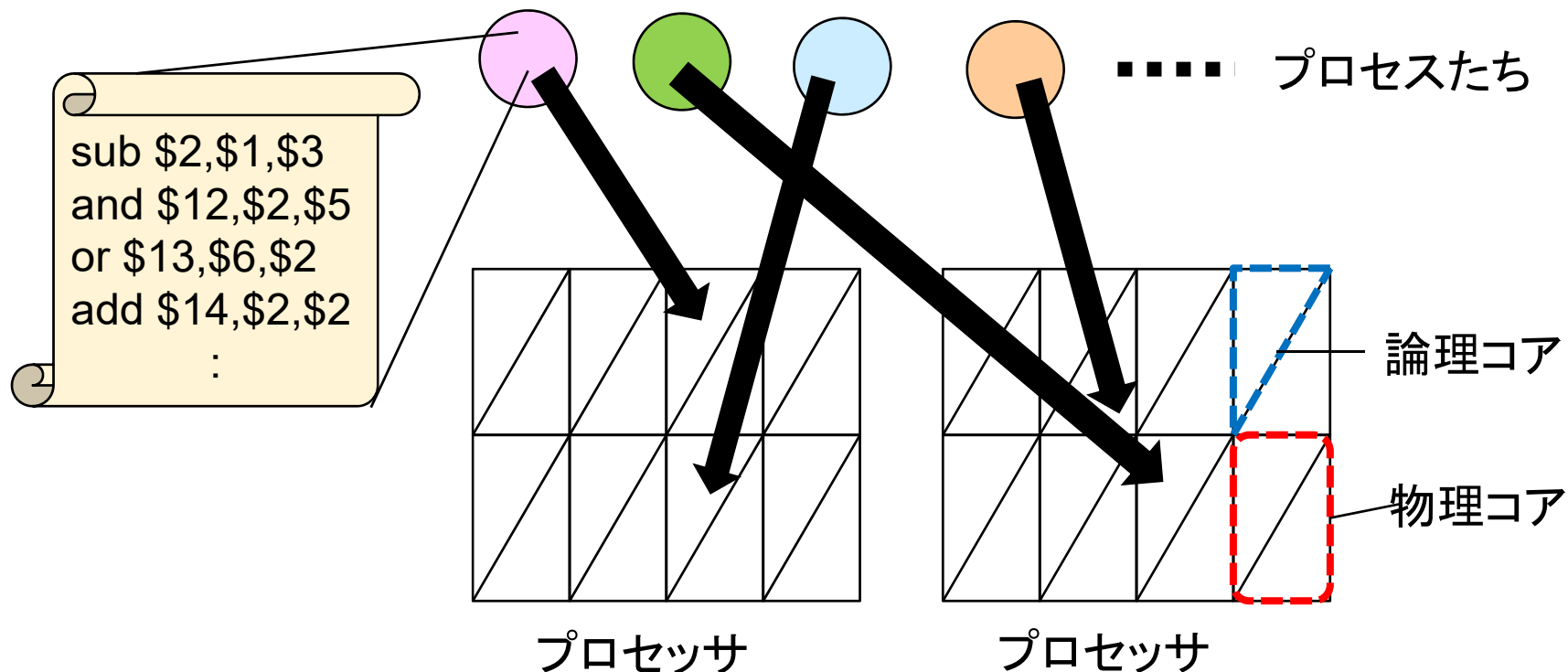
※ TSUBAME3計算ノード(GPU除く)の場合は、 $2 \times 14 \times 2 = 56$

以下の二つの方法

- 複数の**プロセス**を使う
 - 例: 一つの計算機上でWordとExcelとブラウザが動く
 - 例: 一つの計算機上で、シミュレータプログラムが同時に複数動く
- プロセスあたり、複数の(ソフトウェア)**スレッド**を使う
 - あるプログラムが、複数スレッド使うように作成されている
 - Java thread, pthread...

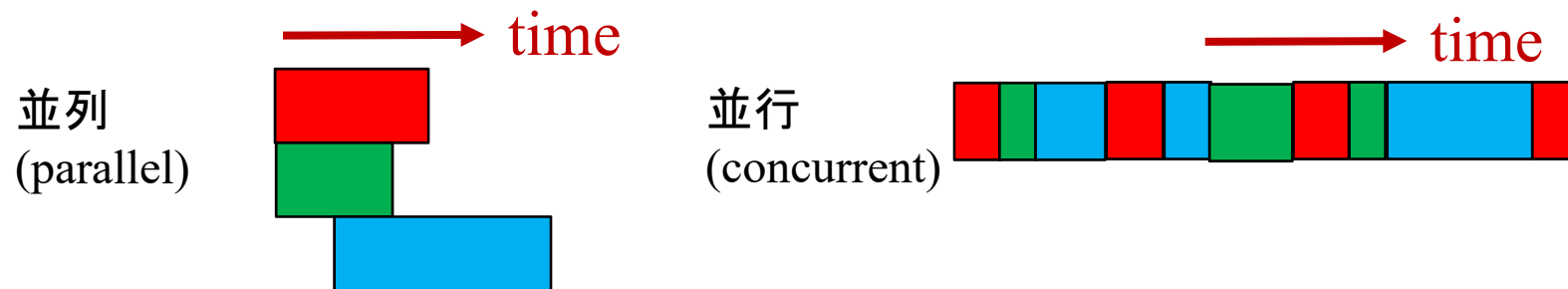
オペレーティングシステム(OS)が多数プロセスをつかさどる

- プロセス(process): (ざっくり言うと) 起動中のプログラム
 - 命令の流れを持つ
 - プログラムを実行開始すると増え、終了すると減る
 - 同じ実行ファイルを同時に複数実行もできる → プロセスは複数
- OSは、それぞれのプロセス(+後述のスレッド)がどの論理コアで動くか決める (スケジューリング)



プロセススケジューリングの性質

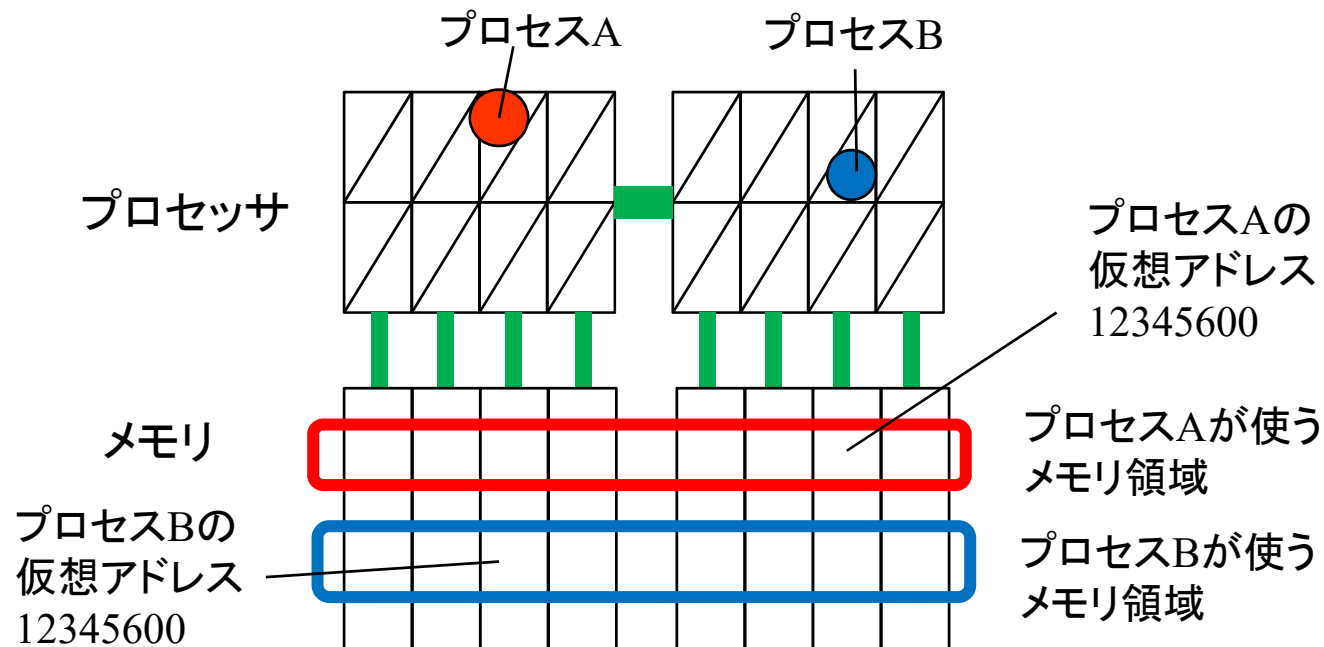
- プロセスの数がコア数より少なかったら？
 - 一部のコアがひまになる
- プロセスの数がコア数より多かったら？
 - 複数のプロセスが、1コアを時間分割して共有 ... 「並行」実行が起きる
 - 歴史的には、1コアしかなかった時代が長く、これが普通だった
 - 速度向上の面からはのぞましくない



- プロセスは、コアを必要としないタイミングもある
 - I/O待ちなど
 - 例: WordやExcelプロセスは、ほとんどの時間マウスやキーボード処理を待っていて、コア利用率は低い
- あるプロセスが、実行途中で別のコアに移動することもある
 - [Q] もし移動ができなかったら、どのような問題がある？

仮想アドレス

- あるプロセスがメモリ上に持つデータは、別のプロセスから読み書きできないようになっている ← 主にセキュリティなどのため
 - プロセスごとに、**仮想アドレス空間**(virtual address space)を持たせることにより、それを実現
 - プロセッサとOSの連携により実現
 - 本授業でこれまで出てきた「アドレス」は、原則、仮想アドレスのことだった
 - `lw $t1, 100($t2) ... ($t2+100)`は仮想アドレスとして解釈される
- ⇔ **物理アドレス**: 物理的なメモリ中の場所

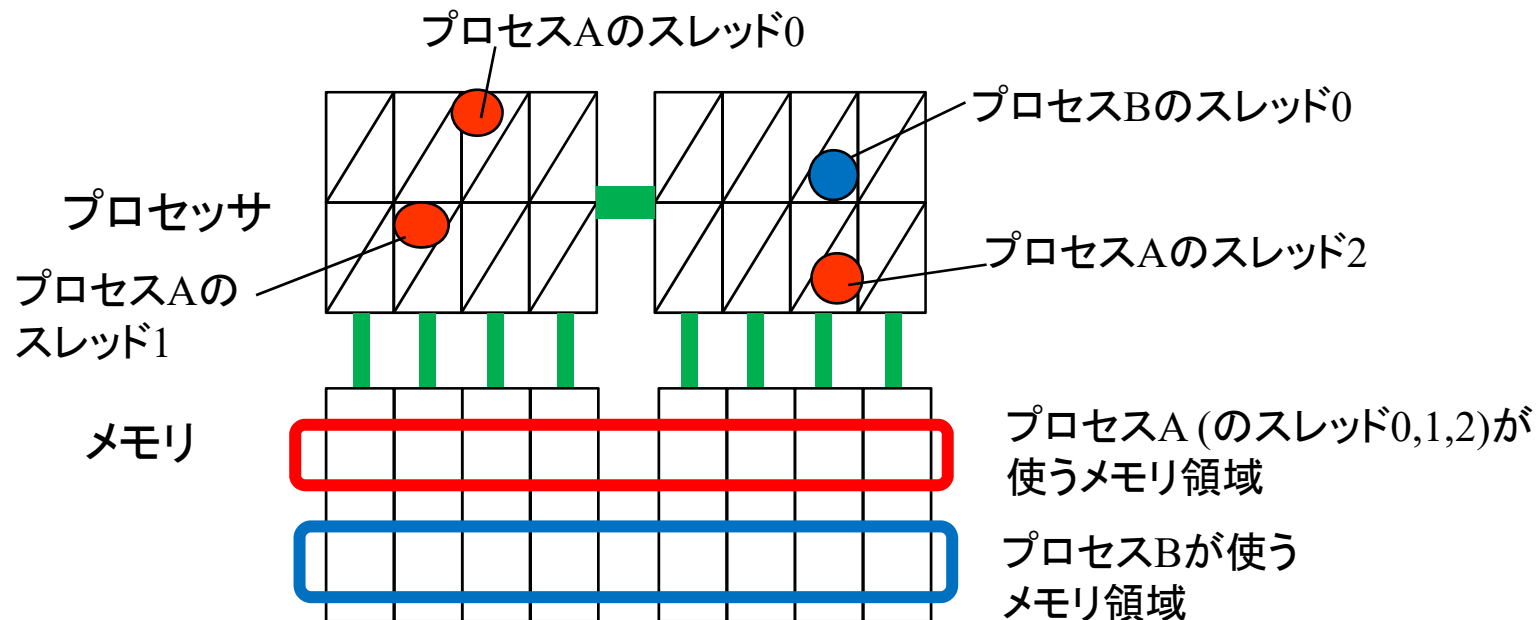


詳細は
OS授業で

スレッド(thread)とは

※ ここで説明するのはソフトウェアスレッド。ハードウェアスレッドとは違う

- **スレッド**:(ざっくり言うと)命令の流れ。下記のようなときに利用できる
 - 1プロセスの中で、複数のコアを使いたいとき
 - 1プロセスの中で、命令の流れ/PCが複数あるほうが良いとき
 - 例: ネットワークサーバなど
- 通常は1プロセス=1スレッドだが、プロセスの中でスレッドを増減可能
 - プロセス内のスレッドたちは、同じ仮想アドレス空間を見る
 - 厳密には、**OSがスケジュール対象とするのはスレッド**




スレッドプログラミングの難しさ

- プログラム中の、依存関係がなく、同時に実行可能な場所をユーザが見つける必要
- 依存関係による問題 (競合) がある場合は、ユーザが解決する必要
- コンパイラが一部対応する場合も

OpenMP対応コンパイラの
コンパイル処理のイメージ

```
#pragma omp parallel for  
for (i=0; i<100; i++)  
    C[i] = A[i]+B[i];
```



詳細は大学院授業「実践的
並列コンピューティング」で

スレッド0

```
for (i=0; i<25; i++)  
    C[i] = A[i]+B[i];
```

スレッド1

```
for (i=25; i<50; i++)  
    C[i] = A[i]+B[i];
```

スレッド2

```
for (i=50; i<75; i++)  
    C[i] = A[i]+B[i];
```

スレッド3

```
for (i=75; i<100; i++)  
    C[i] = A[i]+B[i];
```

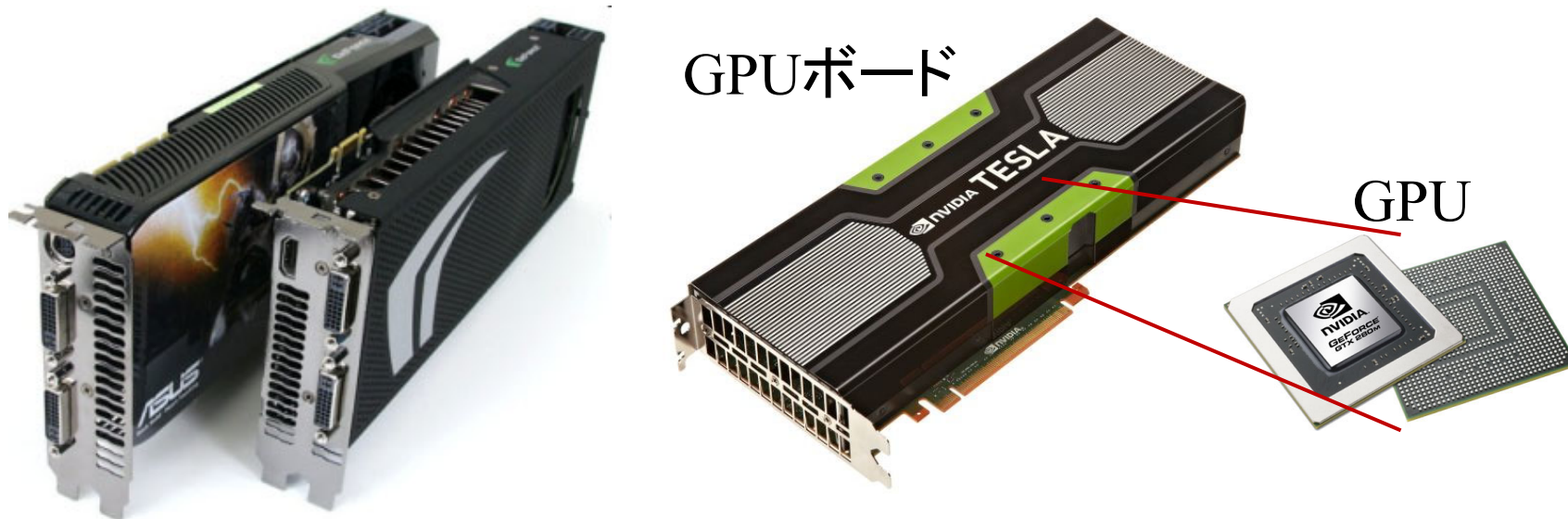

6. 異種プロセッサによる並列性

- これまで見たようにプロセッサ(CPU)は、パイプライン化・種々の並列化などの性能向上を遂げてきた
- しかし、まだ足りない部分を補うために、異なる種類のプロセッサを用いる場合がある
- 典型例：汎用CPU + GPU
 - 例：東工大TSUBAMEスパコン、米国ORNL Summitスパコン
- 汎用CPUでは演算性能 (主に浮動小数点演算) が不足な場合に、**GPU** (graphic processing unit) を併用する
 - TSUBAME3では全Flopsのうち4%がCPU, 96%がGPU



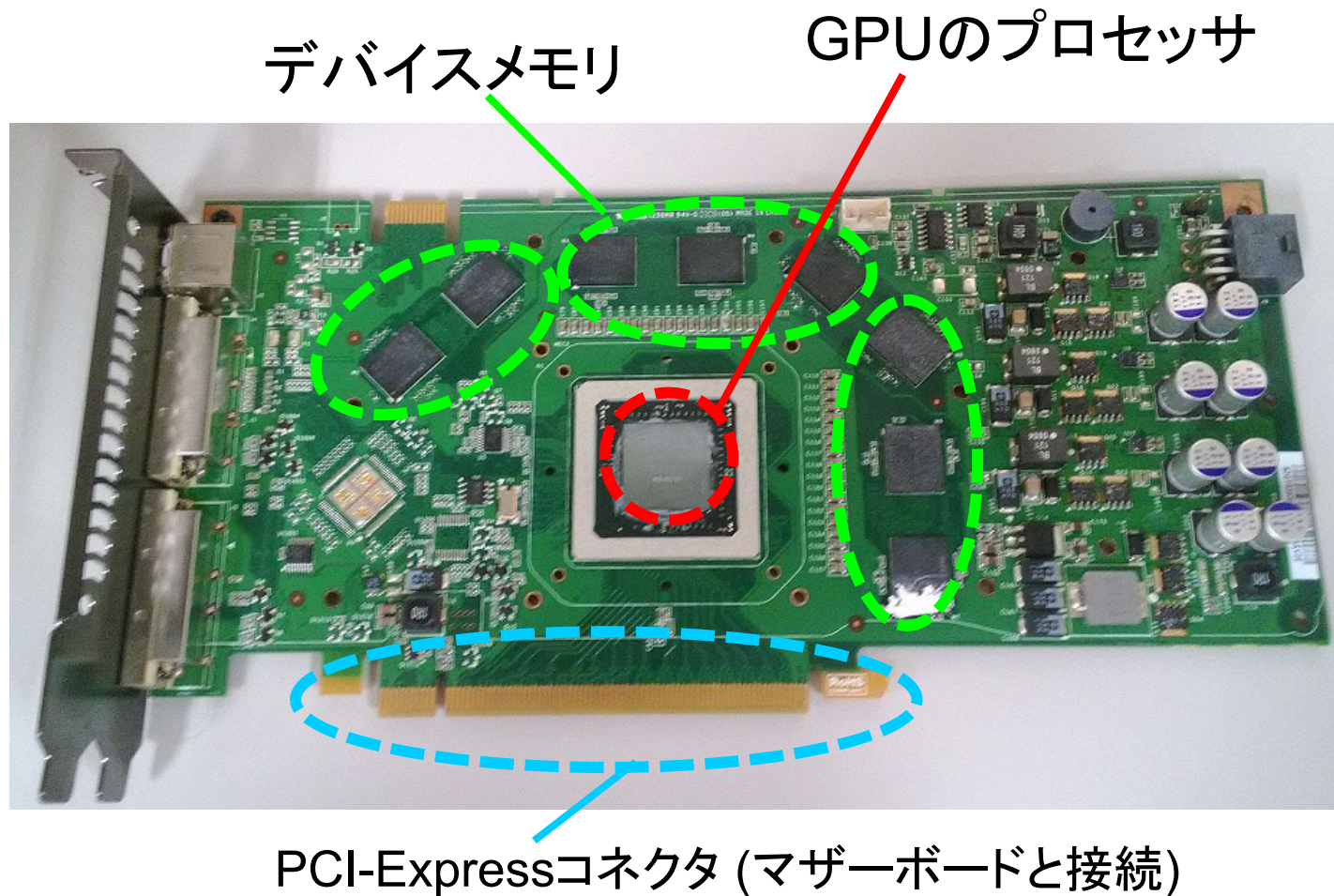
GPUコンピューティングとは

- グラフィックプロセッサ (GPU)は、グラフィック・ゲームの画像計算のために、進化を続けてきた
 - 現在、CPUのコア数は2～12個に対し、GPU中には数百コア
- そのGPUを一般アプリケーションの**高速化**に利用！
 - GPGPU (General-Purpose computing on GPU) とも言われる
- 2000年代前半から研究としては存在。2007年にNVIDIA社の**CUDA言語**がリリースされてから大きな注目



GPUボードの構成

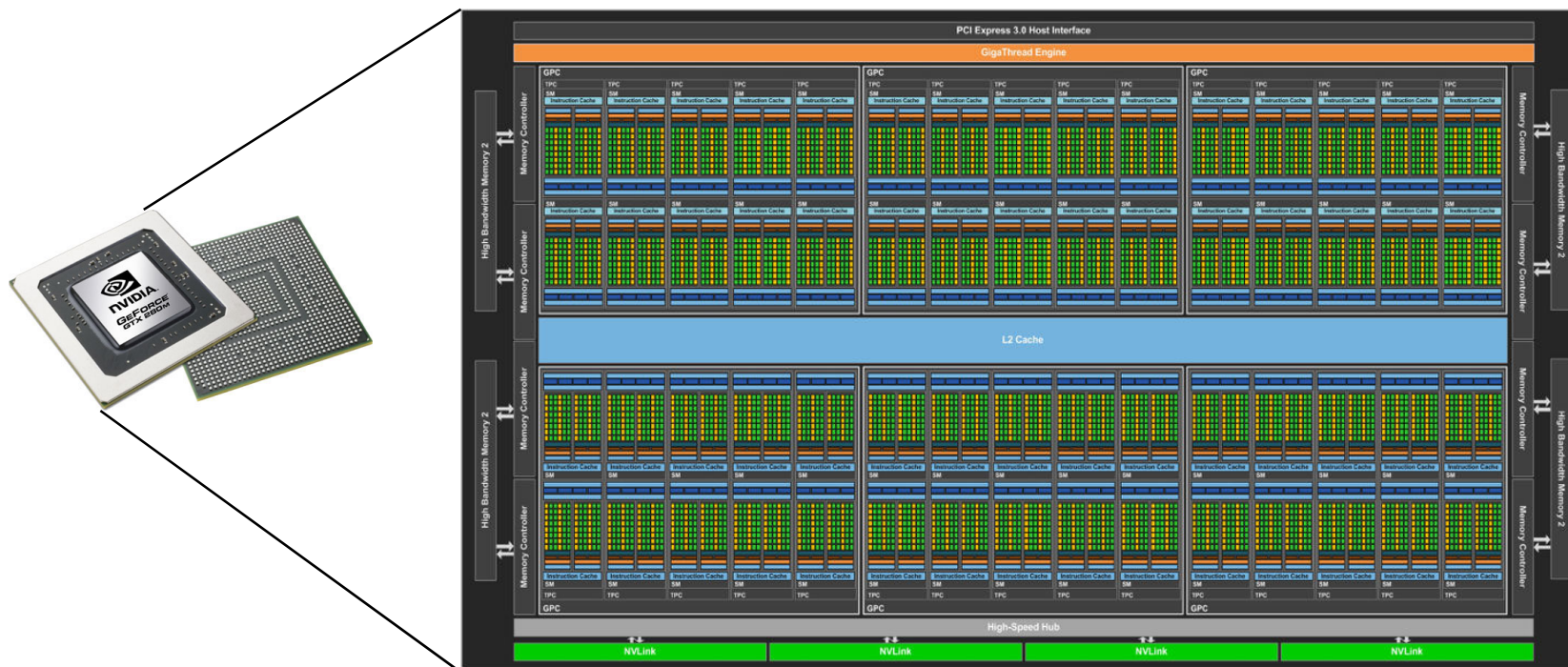
- ふたとヒートシンクを外した様子



GPUの特徴

- コンピュータにとりつける増設ボード
⇒単体では動作できず、CPUから指示を出してもらう
- 多数コアを用いて計算
⇒多数のコアを活用するために、多数のスレッドが協力して計算
TSUBAME3.0のTesla P100 GPUには、1つあたり3584 CUDA cores
- メモリサイズは比較的小さい
⇒CPU側のメモリと別なので、「データの移動」もプログラミングする必要
TSUBAME3.0のTesla P100 GPUには、1つあたり16GBのメモリ

GPUの多数コア



NVIDIA
資料より

- GPUと言えども、CPUと同じ半導体(トランジスタ)の集合で作られている
- なぜ数千コア (CUDA core)もつめ込めるのか？ コアの定義も違う

Tesla P100の場合

- **1GPU = 56 SMs = 112 processing blocks = 3584 CUDA cores**

機能上はこれが
一般的なコアに近い

さらにOSが動かないなど、単純構成

単体のPCを(事実上)持たず、
SIMDの強化版に近い

Tesla P100 GPUのFlops

double/FP64の場合

$56 \text{ (SM)} \times 2 \text{ (block)} \times 16 \text{ (演算)} \times 2 \text{ (乗算+加算)} \times 1.48 \text{ (GHz)} = 5.3 \text{ TFlops}$

single/FP32の場合

$56 \text{ (SM)} \times 2 \text{ (block)} \times 32 \text{ (演算)} \times 2 \text{ (乗算+加算)} \times 1.48 \text{ (GHz)} = 10.6 \text{ TFlops}$

FP16の場合

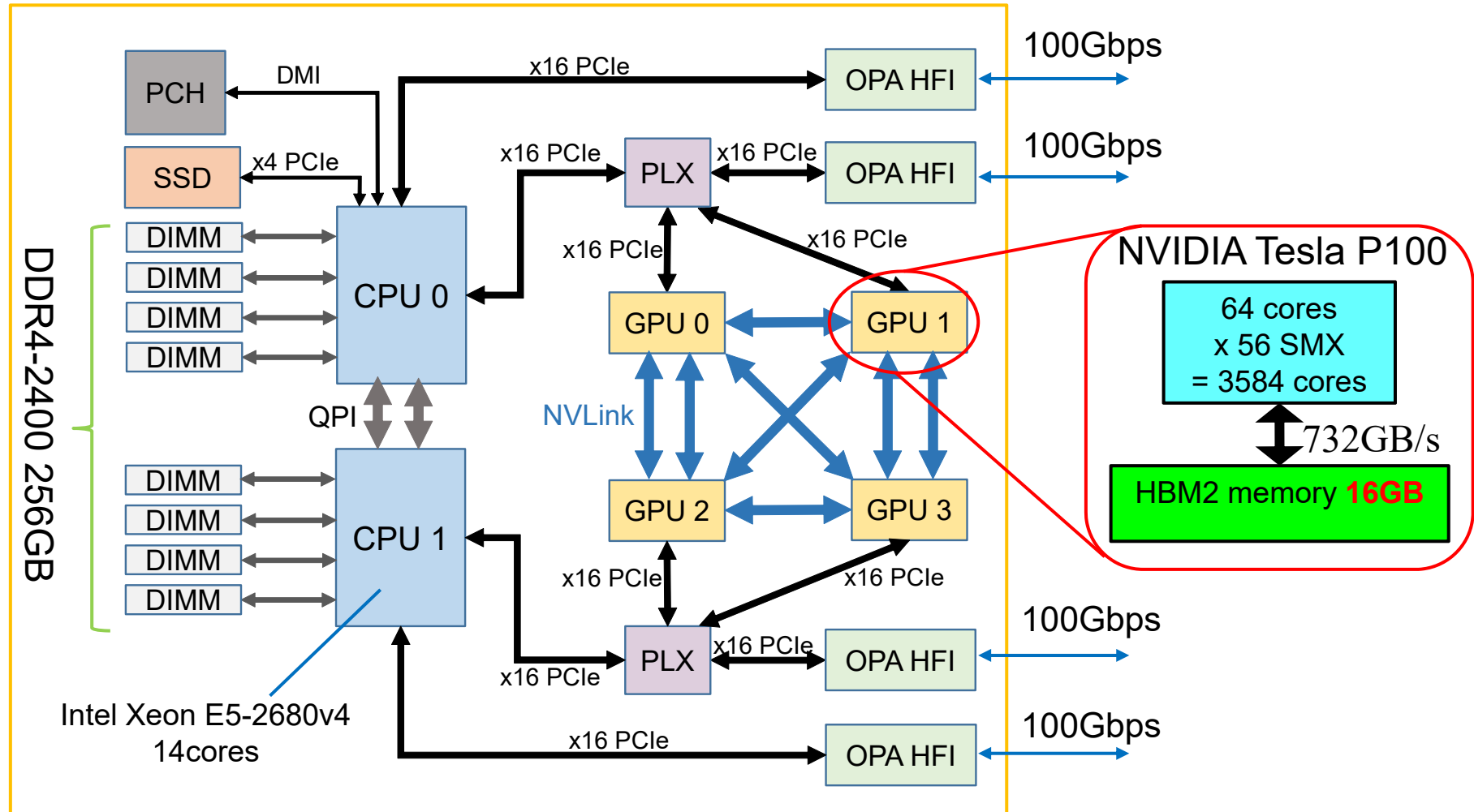
$56 \text{ (SM)} \times 2 \text{ (block)} \times 64 \text{ (演算)} \times 2 \text{ (乗算+加算)} \times 1.48 \text{ (GHz)} = 21.2 \text{ TFlops}$

深層学習の需要の高まりに応じて高速FP16に対応

TSUBAME3のCPUは0.425TFlops(FP64), 0.85TFlops (FP32)

- NVIDIA GPUの中では、TeslaはGPGPU向けのシリーズ
 - FP64: FP32: FP16 = 0.5: 1: 2
- コンシューマ向けGeForceシリーズは、FP32が主
 - FP64: FP32: FP16 = 0.0625: 1: 0 (未対応) など

TSUBAME3.0の計算ノード



$$0.425\text{TFlops} \times 2(\text{CPU}) + 5.3\text{TFlops} \times 4(\text{GPU}) = 22\text{TFlops}$$

$$22\text{TFlops} \times 540(\text{node}) = 11.9\text{PFlops}$$

ここまでのまとめ

プロセッサ/計算機の性能向上技術のうち、並列性を活用するものを取りあげた

1. 機能ユニット間の並列性

- パイプライン処理により活用
- 隣接した命令間の実行は重なる

2. SIMD並列性

3. ハードウェアスレッド並列性

4. マルチコア並列性

5. マルチプロセッサ並列性

6. 異種マルチプロセッサ並列性

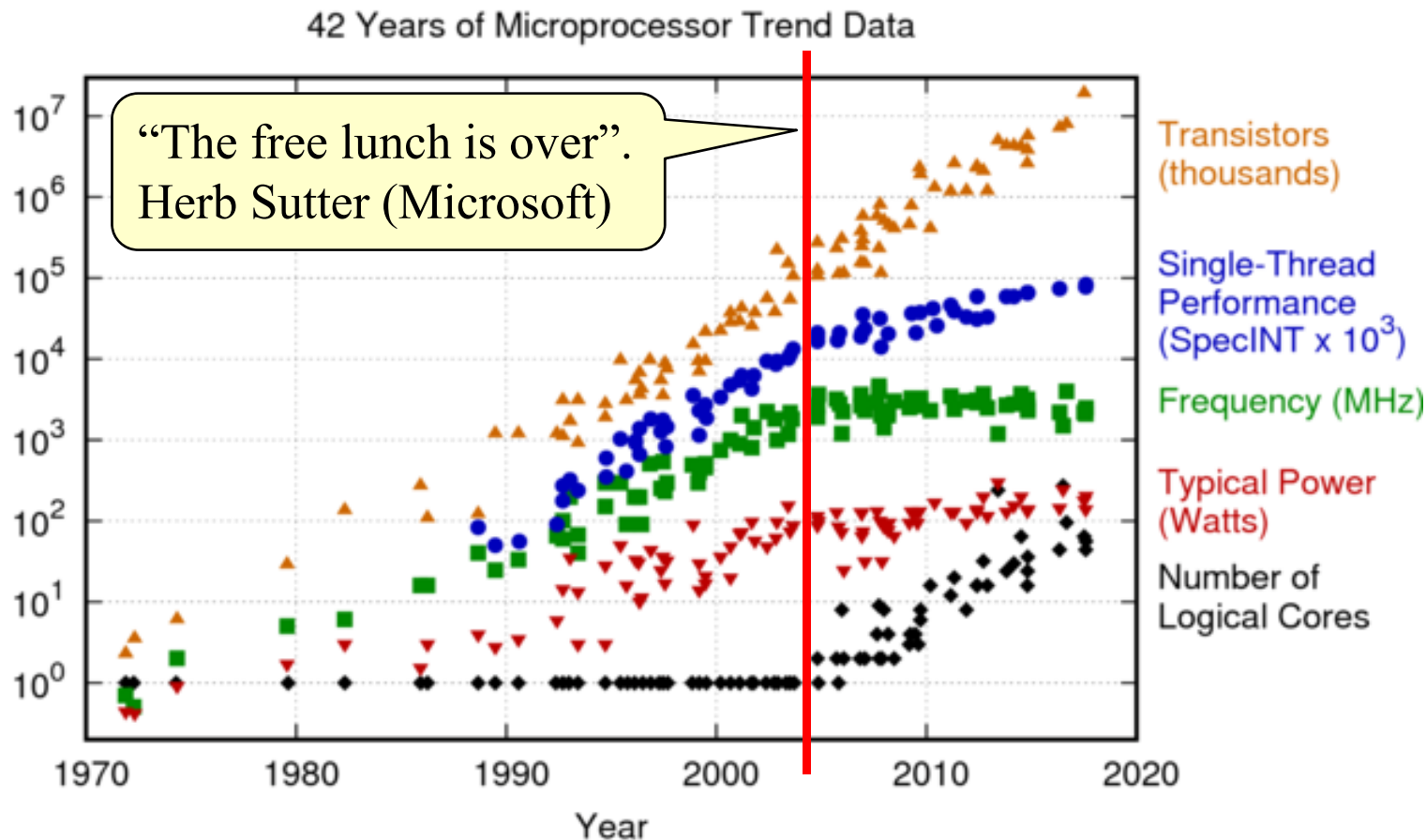
- CPUとGPUなど

7. マルチノード並列性 (次回ネットワークの回で)

1.は、既存の機械語プログラムを変更せずに効率化するものだったが、

2.～7.に対応するためには、プログラム側の対応が必要

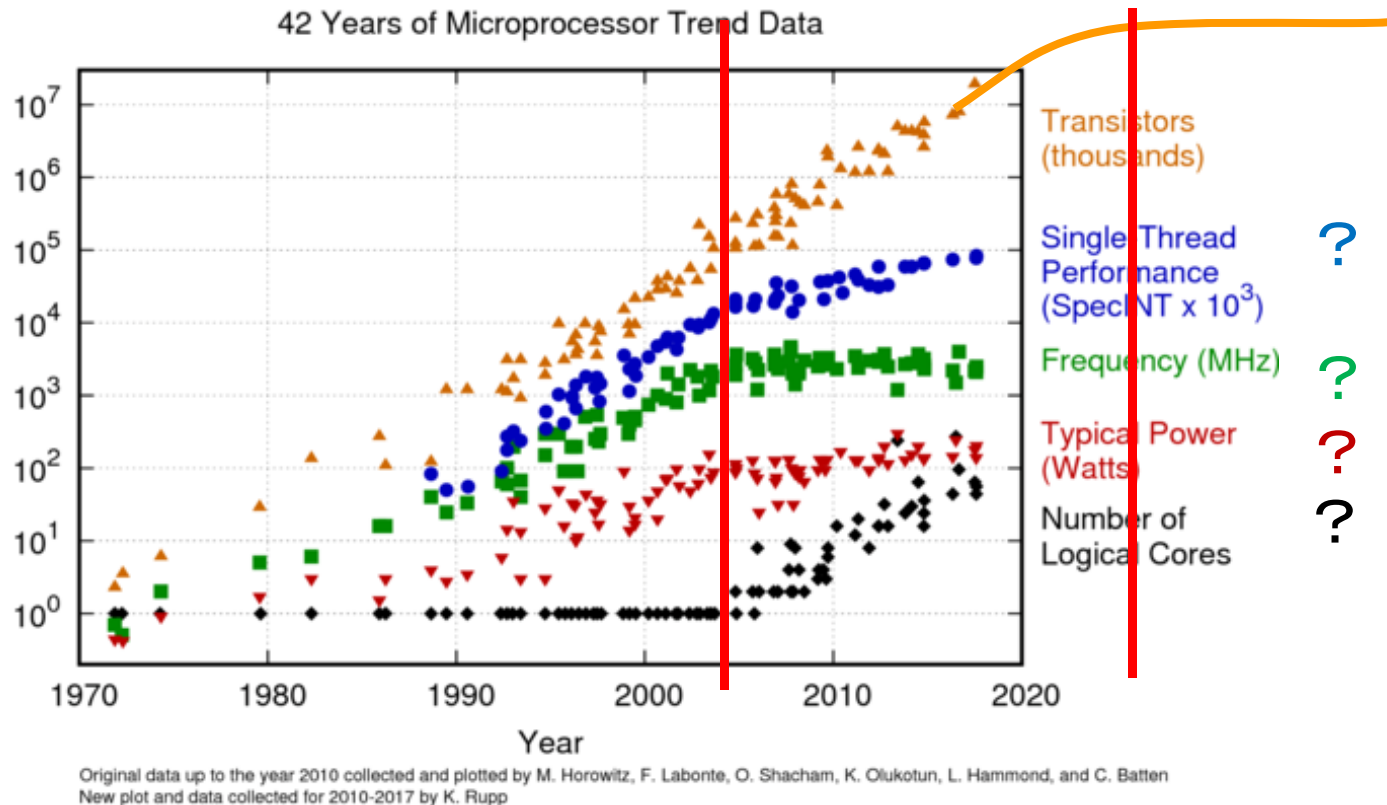
プロセッサの進化の歴史



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

- 継続的なトランジスタ数の増加(**ムーアの法則**)。それを、以下のように活用してきた
- 2005年まで: プログラムの変更不要な性能向上。クロック周波数・パイプライン...
- 2005年以降: コア数増加などの、プログラム変更をまきこむ並列化

ムーアの法則の終焉？



- 2025～2030年ごろに、継続的なトランジスタ数の増加が止まると推測されている
 - トランジスタ・配線の微細化の限界
- それでも、IoT・AI・自動運転などの計算需要の増加は止まらない

→ どうする？ 情報理工学の巨大な課題