

シミュレーション工学

分子動力学法(3) 実装と高速化の詳細

慶應義塾大学大学院理工学研究科基礎理工学専攻物理情報専修

渡辺宙志

はじめに

分子動力学法の実装

- 分子動力学法には様々な種類がある
- 短距離古典分子動力学法のコードは比較的単純

本講義の目的

- 効率の良いコードの実装には、ハードウェアの理解が必須であることを知る
- 単純なコードであっても、計算機の性能を引き出すのは面倒であることを知る

分子動力学法の概要

運動方程式を数値積分する

$$ma = F$$

1. 原子の初期配置と初期速度を決める
2. 原子の位置を更新
3. 原子間に働く力(力積)を計算し、運動量を更新
4. 2.3.のステップを繰り返す

分子動力学法の概要

物体の加速度は

物体に働く力に比例する

$$ma = F$$

物体に働く力の分類

外力

重力
外場による振動など

原子間力

二体力

ファンデルワールス力
バネなど

多体力

曲げ弾性など

分子動力学法の概要

原子間力の起源：原子や電子の量子力学的な相互作用



電子状態を真面目に計算する

→ **第一原理計算** (ab initio calculation)

第一原理計算で力を計算して原子を動かす

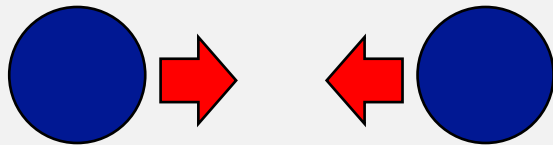
→ **第一原理MD** (ab initio MD, Car-Parrinello method)

※ 電子状態は密度汎関数理論(Density Functional Theory, DFT)で計算する

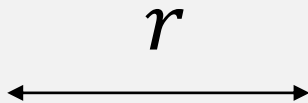
分子動力学法の概要

毎回電子状態を解くのは大変なのでモデル化

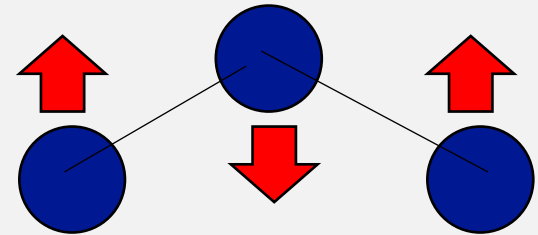
二体力



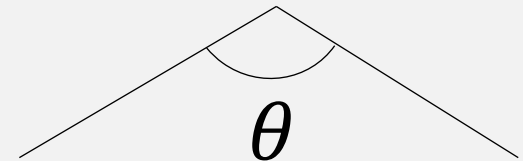
距離だけの関数で表現



多体力



原子の位置関係で表現



電子状態を解かず、原子の位置関係による経験的なポテンシャルで計算する方法を**古典分子動力学法**と呼ぶ

分子動力学法の概要

短距離力

距離が離れると急速に弱くなる力

ファンデルワールス力など

一定距離(**カットオフ距離**)で力の計算を打ち切る

Bookkeeping法などで計算コストを落とす(後述)

長距離力 ※本講義では紹介しません

離れた距離でも力が及ぶ

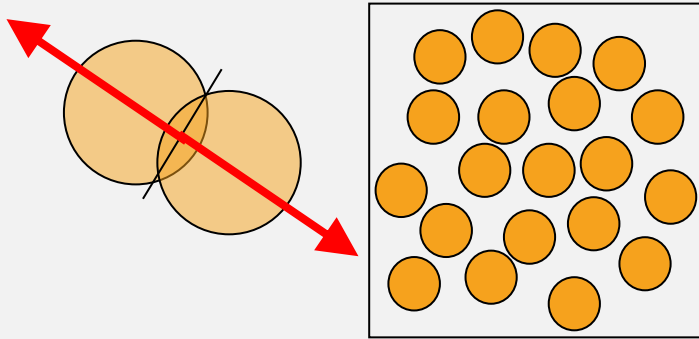
クーロン力や重力など

周期的境界条件の扱いに工夫が必要(Ewald和等)

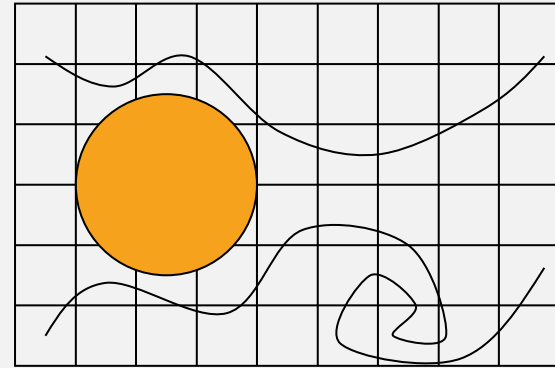
多重極展開などで計算コストを落とす

粒子法と分子動力学法

分子シミュレーション



格子シミュレーション



物体にかかる力を計算
→ラグランジュ描像

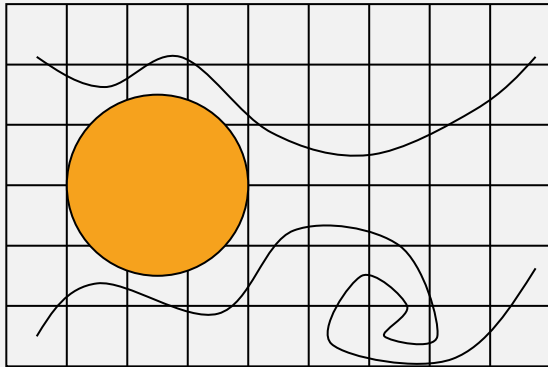
ニュートンの運動方程式

固定された空間で計算
→オイラー描像

ナビエ・ストークス方程式等

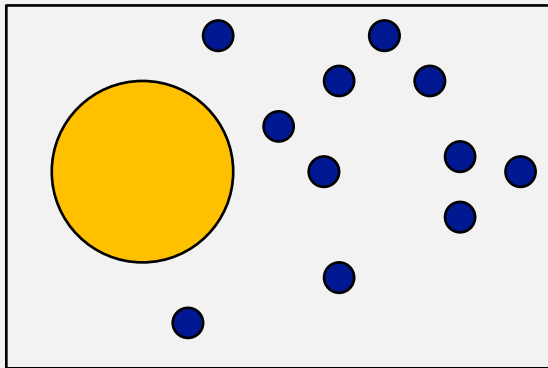
粒子法と分子動力学法

格子法



支配方程式を**固定した格子**上で
離散化して解く

粒子法



支配方程式を**流れに沿って動く参照点**上で離散化して解く

SPH (Smoothed Particle Hydrodynamics)
MPS (Moving Particle Semi-implicit)

粒子法と分子動力学法は扱う支配方程式が異なる

分子動力学法の概要のまとめ

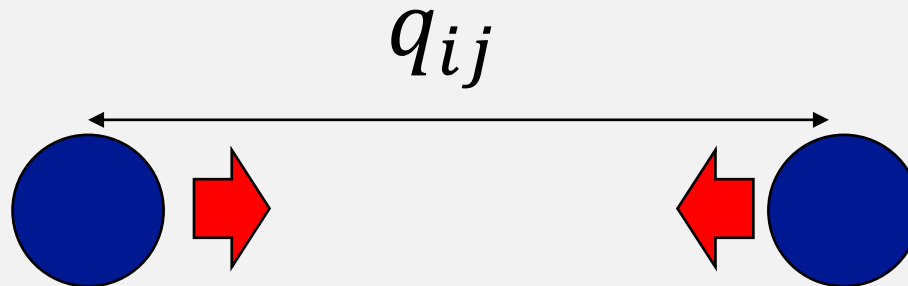
- 分子動力学法(Molecular Dynamics method, MD)は、原子に働く力を計算し、運動方程式を数値積分することで時間発展させる手法
- 原子の電子状態を真面目に解くのが**第一原理MD**、経験ポテンシャルで代用するのが**古典MD**
- 経験ポテンシャルを使う場合、力が遠距離まで届く場合と近距離で減衰する場合で扱いが大きく異なる
- **粒子法と分子動力学法は違う**(数値計算手法としての性質は似ているが、解いている支配方程式が異なる)

MDの実装の基礎

簡単のため、二体相互作用のみを考える
運動を支配するハミルトニアンは以下の通り

$$H = \sum_i \frac{\vec{p}_i^2}{2m} + \sum_{i < j} V(q_{ij})$$

$$q_{ij} \equiv |\vec{q}_{ij}| = |\vec{q}_i - \vec{q}_j|$$



MDの実装の基礎

位置の時間発展は簡単

$$H = \sum_i \frac{\vec{p}_i^2}{2m} + \sum_{i < j} V(q_{ij})$$

$$\left. \begin{aligned} \frac{dq_i^x}{dt} &= \frac{\partial H}{\partial p_i^x} = \frac{p_i^x}{m} \\ \frac{dq_i^y}{dt} &= \frac{\partial H}{\partial p_i^y} = \frac{p_i^y}{m} \\ \frac{dq_i^z}{dt} &= \frac{\partial H}{\partial p_i^z} = \frac{p_i^z}{m} \end{aligned} \right\}$$

まとめてベクトル形式で

$$\frac{d\vec{q}_i}{dt} = \frac{\partial H}{\partial \vec{p}_i} = \frac{\vec{p}_i}{m}$$

MDの実装の基礎

運動量の時間発展はちょっと面倒

$$H = \sum_i \frac{\vec{p}_i^2}{2m} + \sum_{i < j} V(q_{ij})$$

$$\frac{dp_i^x}{dt} = - \frac{\partial H}{\partial q_i^x} = - \sum_{i < j} \frac{\partial V(q_{ij})}{\partial q_i^x}$$

$$q_{ij} \equiv |\vec{q}_{ij}| = |\vec{q}_i - \vec{q}_j|$$

MDの実装の基礎

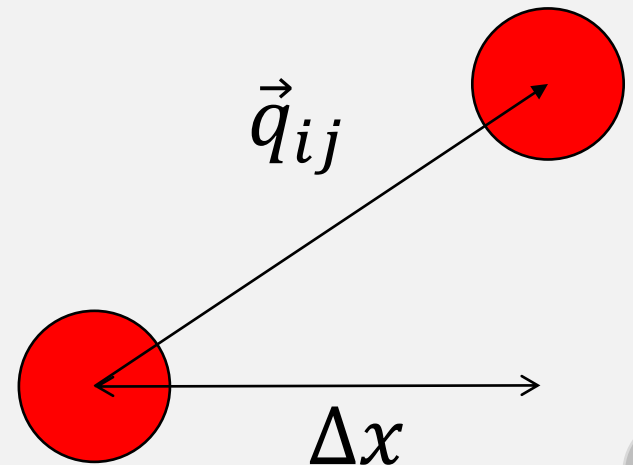
ij原子間に働く力のx成分 $f_{ij}^x = -\frac{\partial V(q_{ij})}{\partial q_i^x} = -\frac{dV(q_{ij})}{dq_{ij}} \frac{\partial q_{ij}}{\partial q_i^x}$

距離の定義 $q_{ij}^2 = (q_i^x - q_j^x)^2 + (q_i^y - q_j^y)^2 + (q_i^z - q_j^z)^2$

両辺を q_i^x で偏微分

$$2q_{ij} \frac{\partial q_{ij}}{\partial q_i^x} = 2(q_i^x - q_j^x)$$

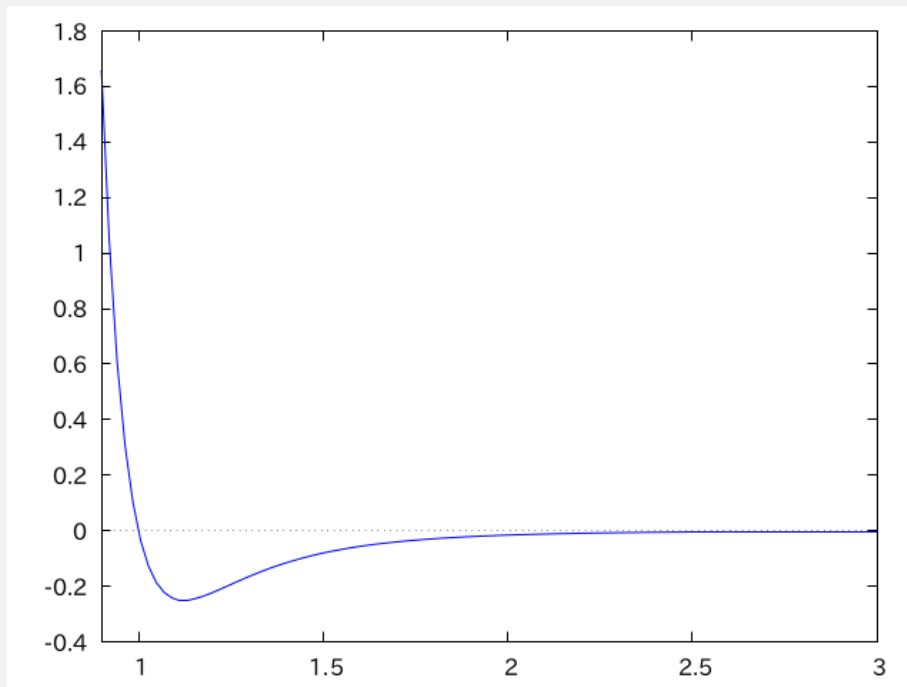
$$f_{ij}^x = -\frac{dV(q_{ij})}{dq_{ij}} \frac{\Delta x}{q_{ij}}$$



Lennard-Jonesポテンシャル

$$V(r) = 4\varepsilon \left(\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right)$$

ε : 相互作用エネルギー
 σ : 原子直径



- 貴ガスの相互作用をモデル化
- 近距離で斥力
 - 中距離で引力
 - 遠距離で相互作用無し

Lennard-Jonesポテンシャル

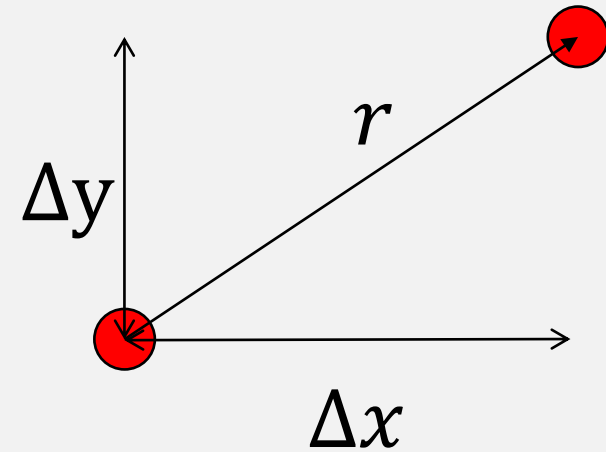
相対座標 $\vec{r} = (dx, dy, dz)$

ポテンシャル $V(r) = 4(r^{-12} - r^{-6})$

力 $f(r) = \frac{48}{r^{13}} - \frac{24}{r^7}$

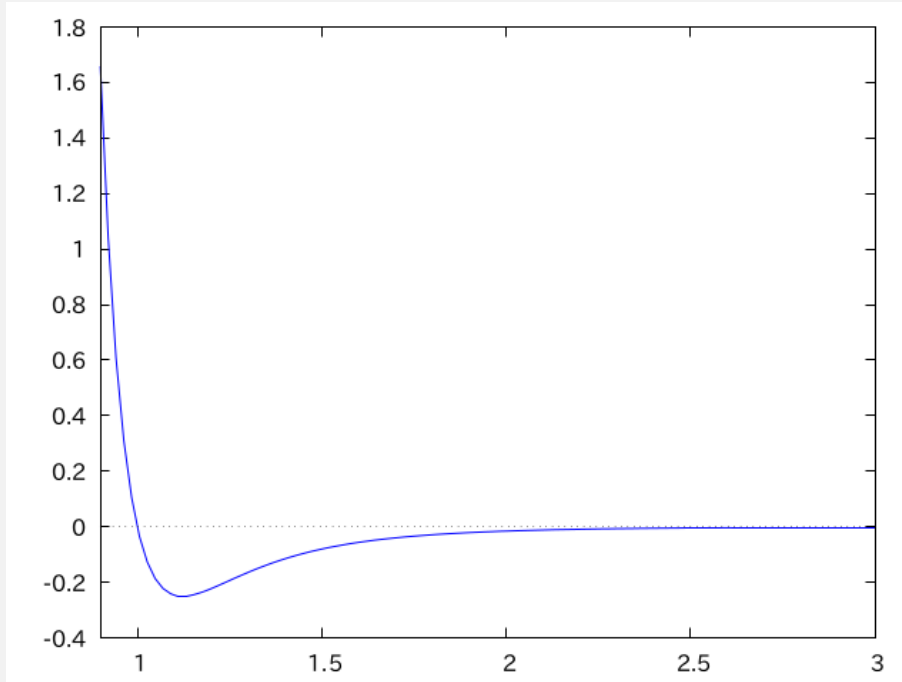
x方向の力積 $I_x = f \times \frac{\Delta x}{r} \times dt$

$$= \left(\frac{48}{r^{14}} - \frac{24}{r^8} \right) \Delta x dt$$



力積が距離の偶数次のみで記述できる

Lennard-Jonesポテンシャル

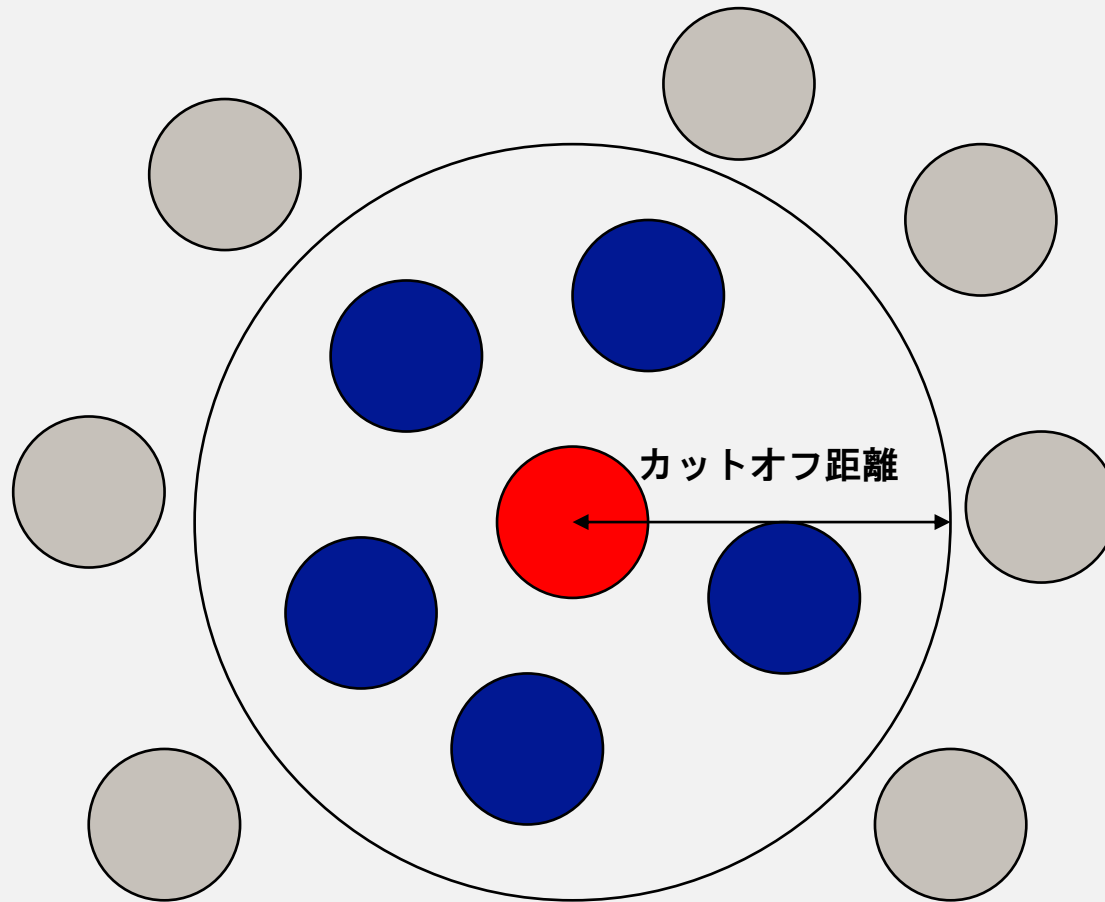


貴ガスの相互作用をモデル化

- 近距離で斥力
- 中距離で引力
- 遠距離で相互作用無し

原子は遠距離で相互作用しない
カットオフ距離を設定し、それ以遠は無視
カットオフ距離内の原子ペアのみ力を計算する

相互作用ペア探索



注目する原子から見てカットオフ距離内にいる
原子番号のリストを作りたい

相互作用ペア探索

単純な実装

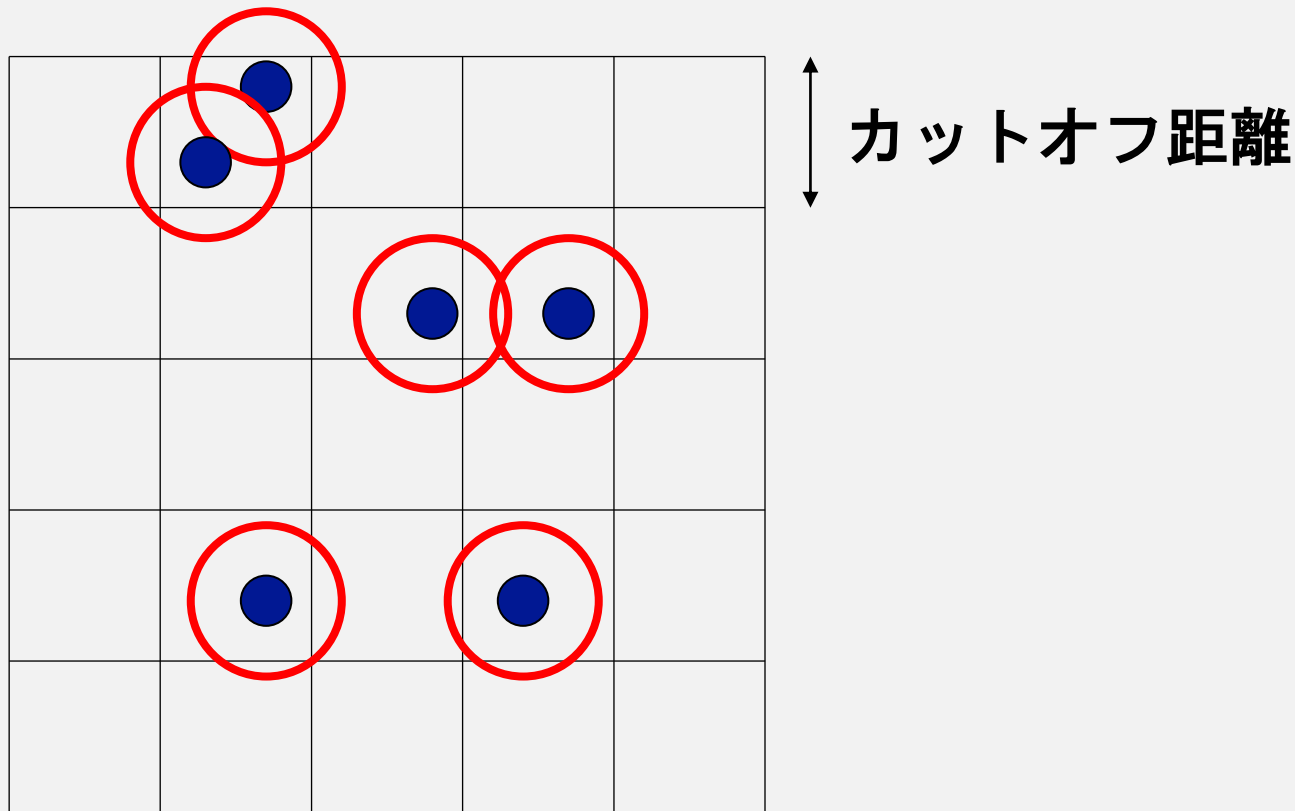
全てのペアについて距離をチェックする

```
for i in range(N-1):  
    for j in range(i+1,N):  
        dx = qx[i] - qx[j]  
        dy = qy[i] - qy[j]  
        dz = qz[i] - qz[j]  
        r2 = dx**2 + dy**2 + dz**2  
        if r2 < cutoff**2:  
            add_pair(i, j)
```

$$\text{計算量} = \text{ループの回転数} = \frac{N(N-1)}{2} = O(N^2)$$

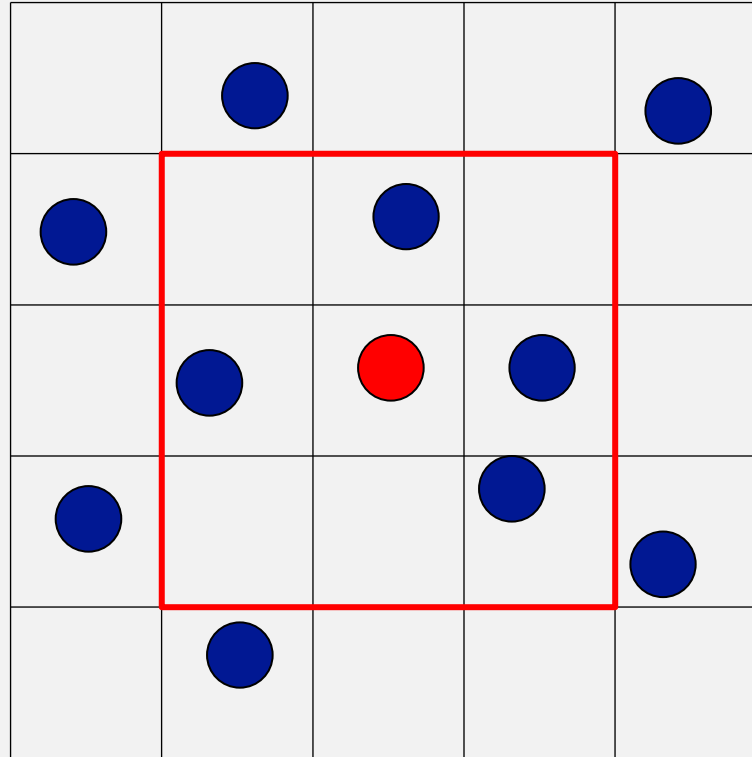
原子数が大きい領域で使い物にならない

相互作用ペア探索



系をカットオフ距離のセルに分割する
同じグリッドか隣接するセルの原子は相互作用する可能性がある
隣接しないセルに所属する原子は相互作用することはない

相互作用ペア探索



各原子が所属するセル番号の「住所録」を作る
ある粒子について、隣接する9つのセルのみ探索する

住所録作成、セルの探索どちらも $O(N)$

MD実装の基礎のまとめ

力の計算

- ポテンシャルが距離の偶数次のみを含む場合、力積が距離の偶数次のみで書ける
- ポテンシャルが奇数次を含む場合は逆数平方根の計算が必要

相互作用ペアリスト

- 相互作用距離にカットオフがある場合、相互作用する原子ペアを探索する必要がある
- 全探索すると $O(N^2)$ だが、グリッド探索により $O(N)$ となる

分子動力学法の実装方法

1. 初期状態を生成する
2. グリッド探索により相互作用ペアを作成する
3. それぞれのペアについて力積を計算し、運動量を更新する
4. 更新された運動量にもとづいて位置を更新する
5. 3～5を繰り返す

分子動力学法の高速化

分子動力学法の実装(再掲)

1. 初期状態を生成する
2. グリッド探索により相互作用ペアを作成する
3. それぞれのペアについて力積を計算し、運動量を更新する
4. 更新された運動量にもとづいて位置を更新する
5. 3～5を繰り返す



上記の単純な実装では

まったく

性能が出ない

分子動力学法の高速化

Q. なぜ単純なコードでは性能が出ないのか？

A. 計算機が複雑化しているから



CPUは年々複雑化、多階層化している
計算機の仕組みを理解してコードを書かない
と性能が出ない

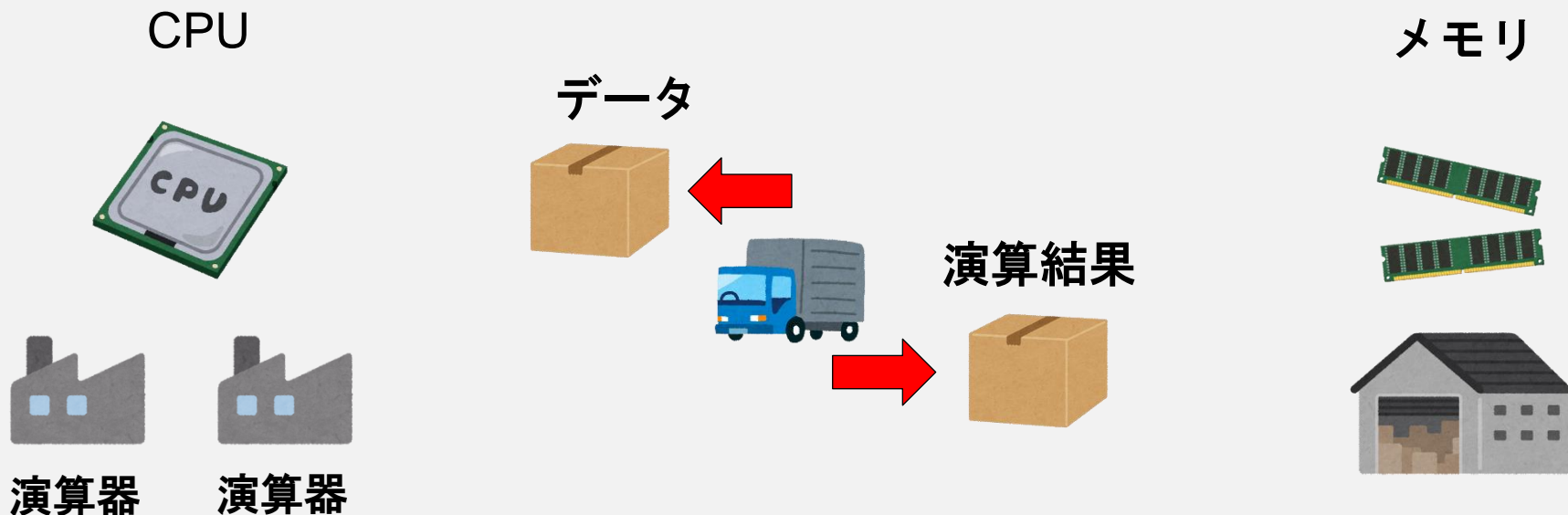
近年特に重要なのが**キャッシュ**と**SIMD**

計算機の仕組み

計算機とは

メモリからデータと命令を取ってきて
演算機に投げ
演算結果をメモリに書き戻す

装置のこと



計算機の仕組み

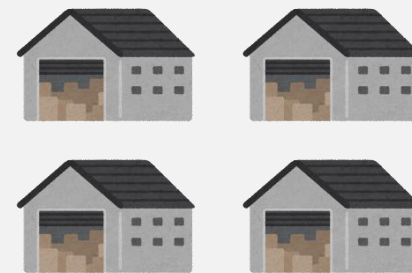
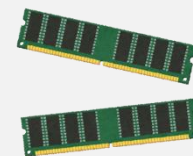
CPUの性能向上に比べて、メモリ転送能力が追いつかない

CPU



演算能力は向上

メモリ



メモリ容量も増大



転送能力は貧弱

メモリアクセス

レイテンシ

データを要求してから、データが届くまでの時間



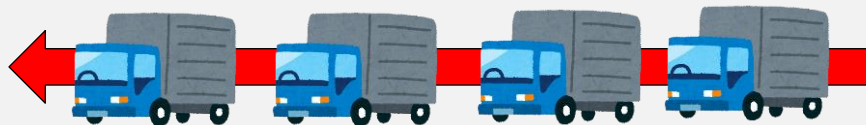
キャッシュアクセスで数サイクル～数十サイクル
メモリアクセスで数百サイクル程度

メモリ空間にランダムアクセスすると性能が出せない

スループット

※性能が出ない＝CPUが遊ぶ

計算能力に比較したデータ転送能力



計算が「軽い」問題は、本質的に性能が出せない

Byte/Flop



FLOPS (floating operations per second)

計算機の演算性能を表す指標
一秒間に浮動小数点演算を何回できるか



Byte/s

メモリの転送性能を表す指標
一秒間に何バイトを転送できるか

Byte/Flop (B/F)

メモリ転送性能と演算性能の比
1回演算する間に何バイト転送できるか

Byte/Flop

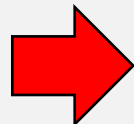
一次元の運動方程式で位置を更新するコード

```
for i in range(N):  
    r[i] = r[i] + v[i] * dt
```

ループ内のメモリ転送と計算量

1. $r[i]$ と $v[i]$ をロードする (16 Bytes)
2. 加算と乗算を実行する (2 FLOPS)
3. $r[i]$ をストアする (8 Bytes)

このコードでCPUが遊ばないためには、B/Fが12必要
現代のCPUの典型的なB/F値は0.5程度

 ピーク性能の1/24までしか出せない

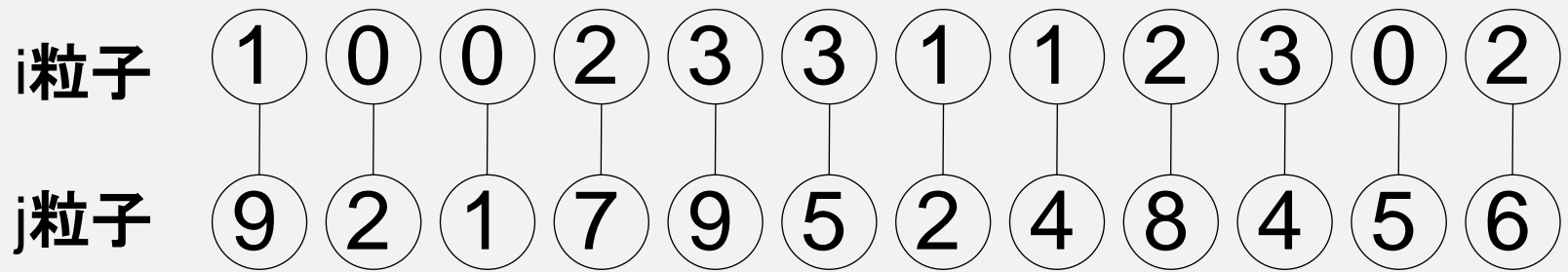
メモリアクセスの低減

力の計算はペアごとに行う

相互作用範囲内にあるペアは配列に格納

ペアの番号の若い方をi粒子、相手をj粒子と呼ぶ

相互作用ペア



2個の粒子の座標を取得 (48 Bytes)

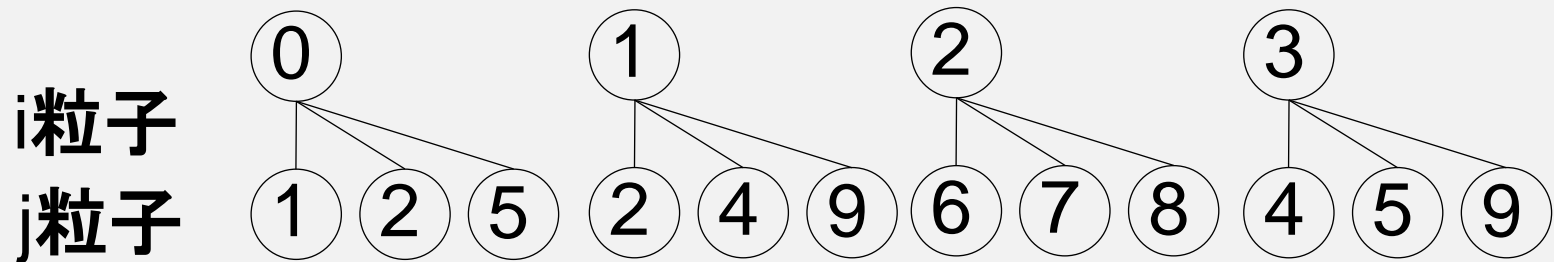
力積を計算し、運動量を更新 (約50 FLOP)

運動量の書き戻し (48 Bytes)

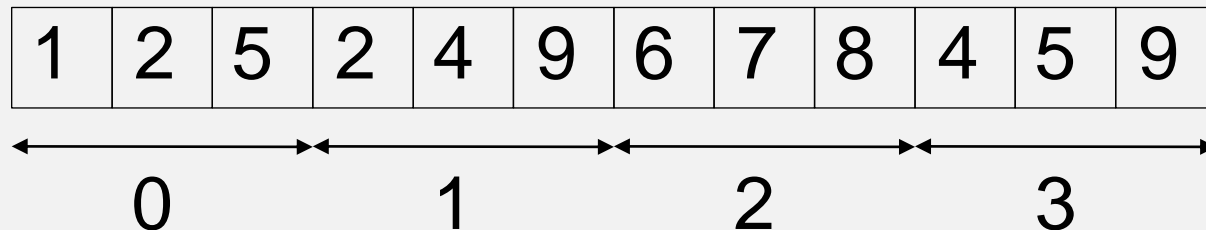
B/F=2.0を要求

メモリアクセスの低減

i粒子をキーとしてソート



メモリ表現



i粒子の情報がレジスタに乗る

→読み込み、書き込みがj粒子のみ

→ペア数が十分多ければメモリアクセスが半分に

キャッシュ

メモリは大容量だが遠くてデータ転送が間に合わない
→演算器の近くに小容量高速なメモリを置く(**キャッシュ**)

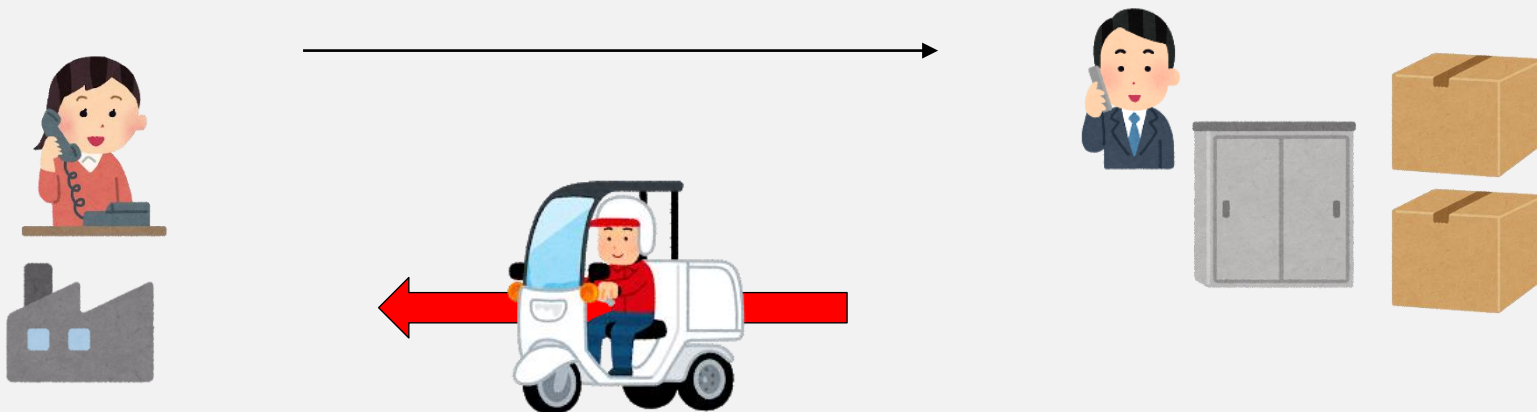


一度使ったデータは再利用する可能性が高いため、演算器の近くにデータを残しておく(**時間局所性**)

使ったデータの周辺のデータは今後使われる可能性が高いため、一緒に持ってくる(**空間局所性**)

キャッシュ

演算器は、まず欲しいデータがキャッシュにあるか確認する



キャッシュに欲しいデータがあった場合はすぐに手に入る(**キャッシュヒット**)



キャッシュに欲しいデータがなかった場合はメモリにアクセス(**キャッシュミス**)

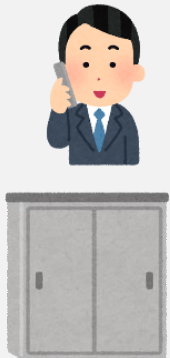
キャッシュ



14番地のデータをください



連続する塊を
まとめて取ってくる
(ライン)

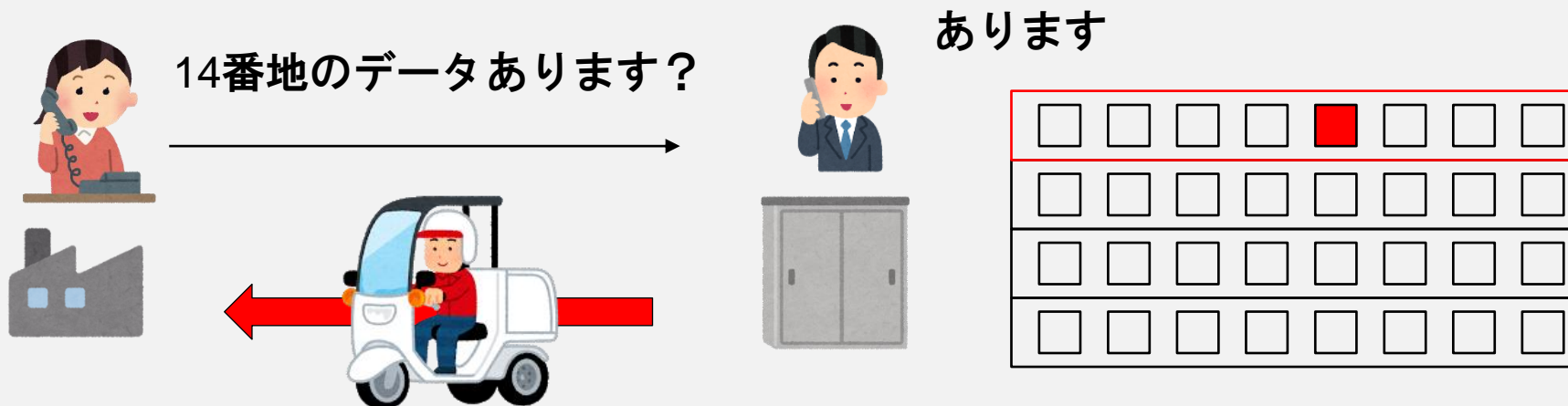


0	1	2	3	4	5	6	7	
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	00
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	10
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	20
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	30

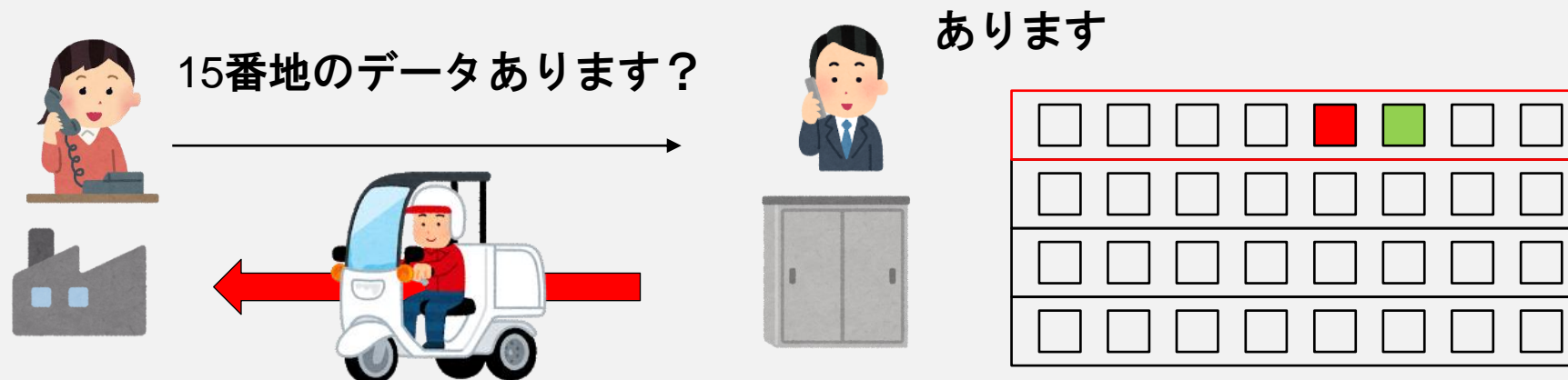
キャッシュからあふれたラインは
メモリに書き戻す

キャッシュ

時間局所性 一度使ったデータは、短時間にまた使われる傾向にある



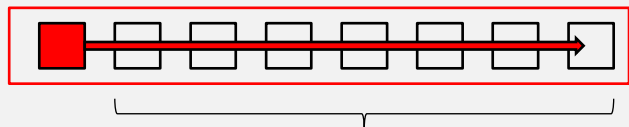
空間局所性 一度使ったデータの近くのデータが使われる傾向にある



キャッシュ

メモリの連続アクセス

ここへのアクセスでキャッシュミス

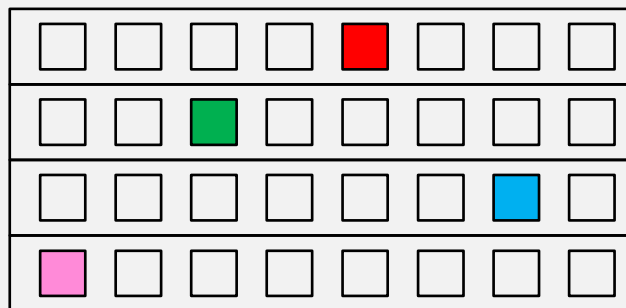


残りのデータは続けてキャッシュヒット

キャッシュ容量の大部分を活用
キャッシュヒット率が高い

0	1	2	3	4	5	6	7	
00								
10								
20								
30								

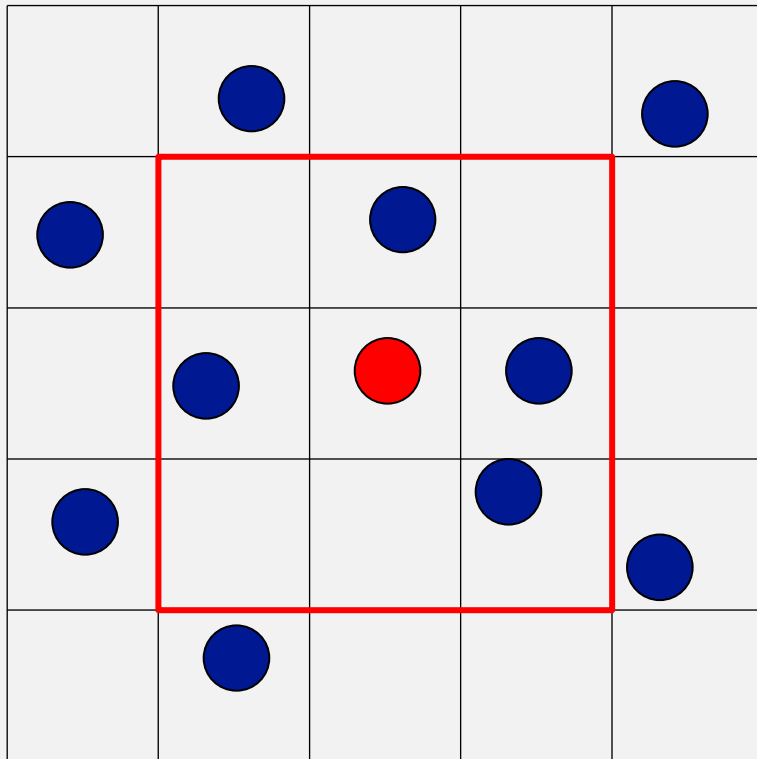
メモリのランダムアクセス



キャッシュ容量の一部しか活用してない
キャッシュミスが多発

0	1	2	3	4	5	6	7	
00								
10								
20								
30								

Bookkeeping法

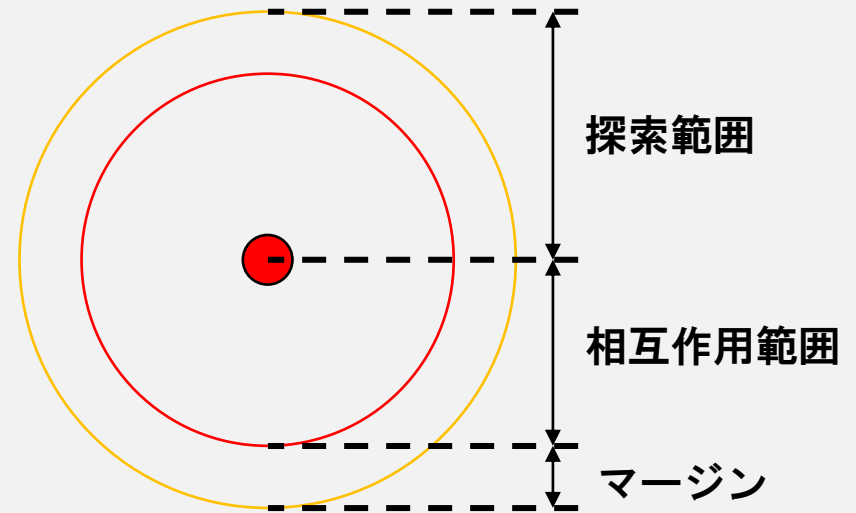


各粒子の所属するセル番号(住所)を登録
注目する粒子の近傍のセルに住むご近所
さんを住所録から逆引き

「住所録」へのアクセスは非連続的かつランダムアクセス
キャッシュヒット率が低く、極めて遅い
→相互作用ペアリストをなるべく作りたくない

Bookkeeping法

- ペアリスト構築時に、相互作用距離より離れた距離にいるペアを探索、リストに登録する
- 安全マージンを使い切るまで、同じペアリストを使いまわす
- この手法をBookkeeping法と呼ぶ



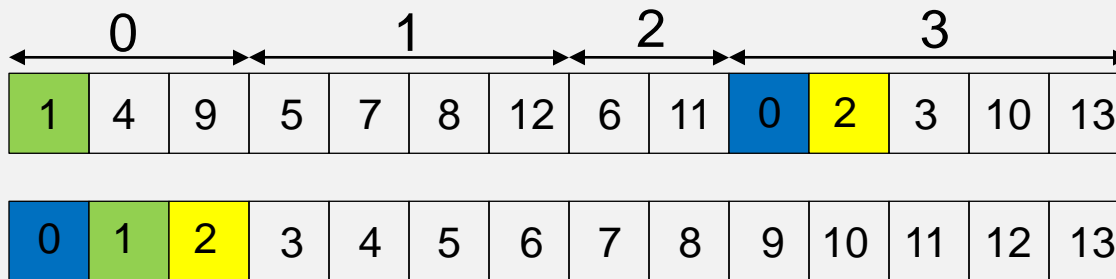
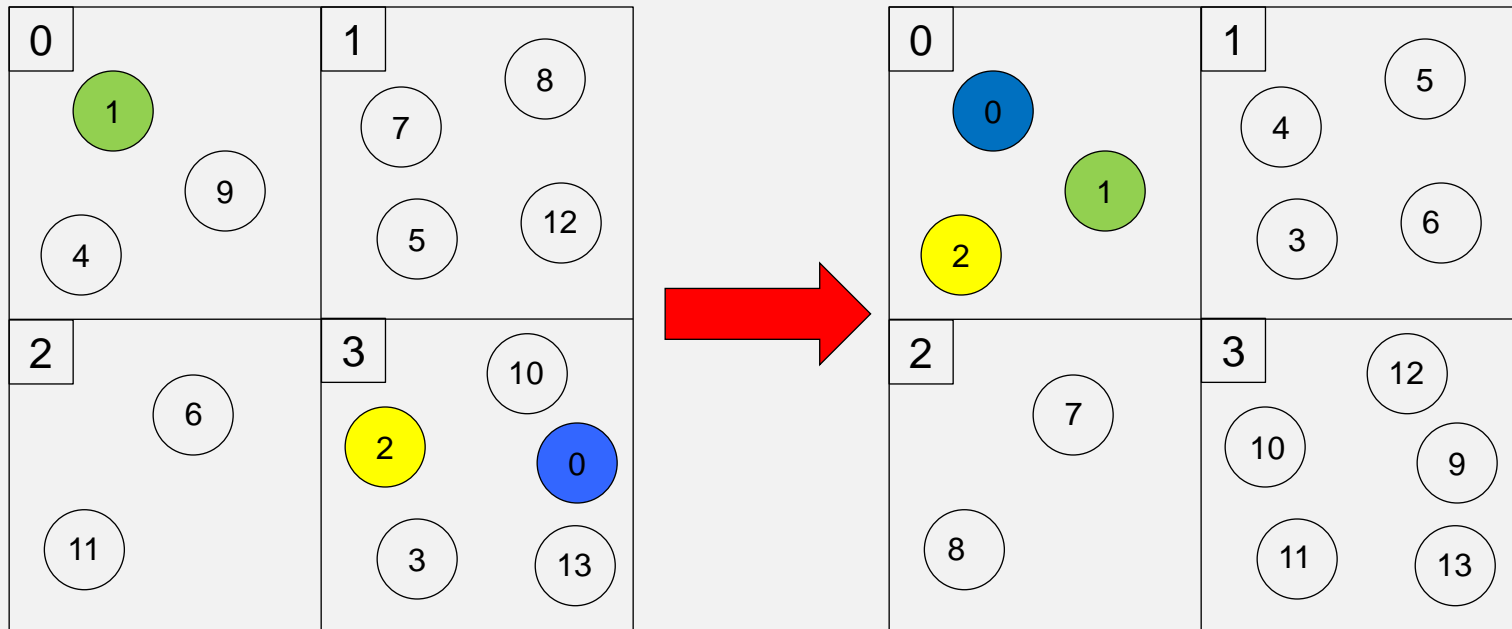
😊 重い処理であるペアリスト作成をサボることができる

😓 相互作用ペアリストに相互作用していないペアを含むため、力の計算前にチェックが必要

適切なマージンを取ると、全体で二倍以上早くなる

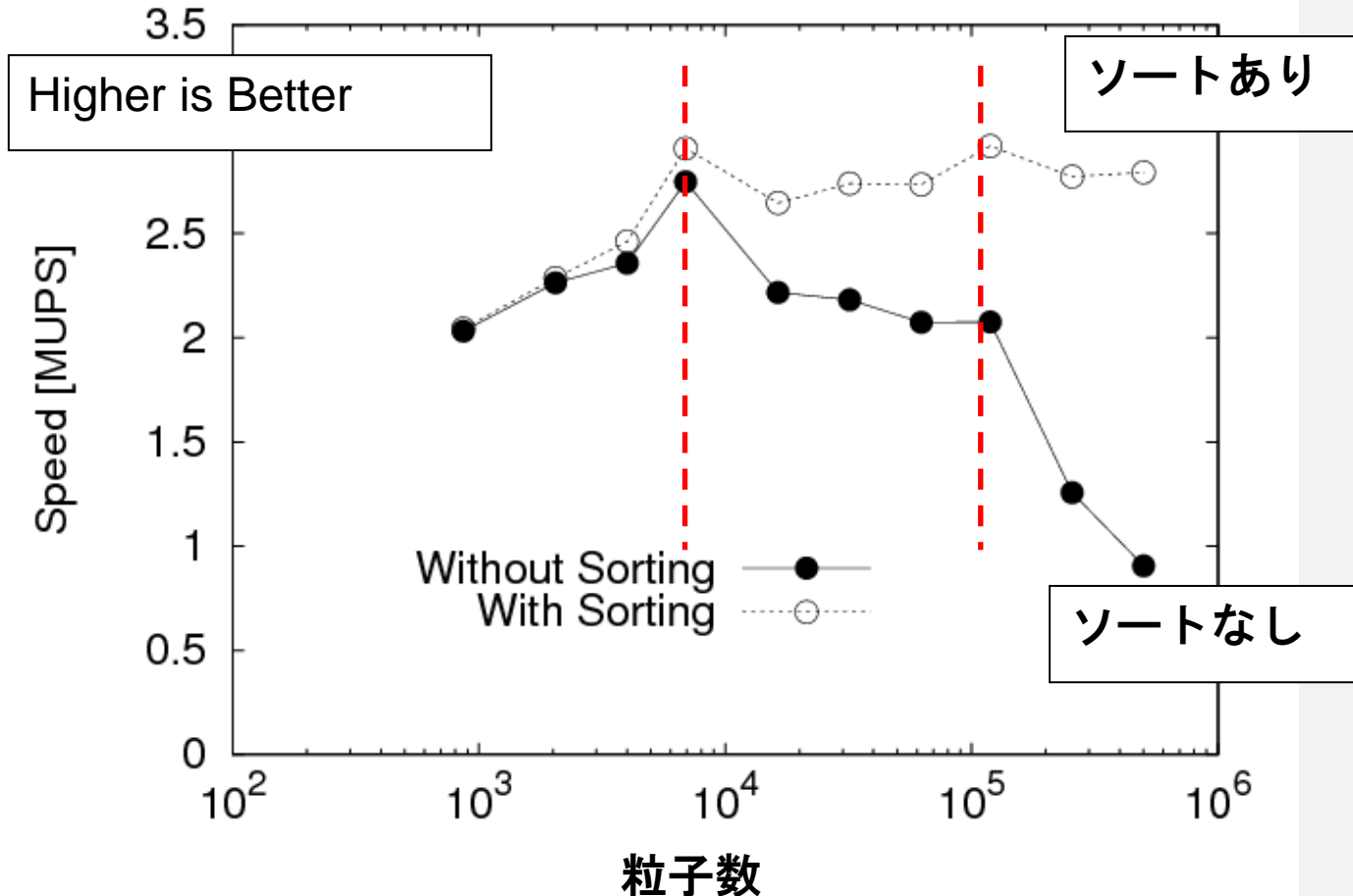
キャッシュの有効活用

時間発展を続けると、空間的には近い粒子が、メモリ上では遠くに保存されている状態になる→ソート



番号のふり直し

キャッシュの有効活用



ソートなし：粒子数がキャッシュサイズをあふれる度に性能が劣化する
ソートあり：性能が粒子数に依存しない

メモリアクセス最適化のまとめ

- 現代のCPUはメモリアクセスがボトルネックになっており、高い計算性能を活かすのが困難
- 高速小容量メモリであるキャッシュの活用が性能向上のカギ
- 使うデータがなるべくメモリ上で連続している方が性能が良い→ソートが有効

余計に計算してでもメモリアクセスを減らしたほうがトータルで高速になることが多い



SIMD化とは何か

科学計算に使われる汎用CPUは、ほぼSIMDを採用している

なぜSIMDが必要か？

SIMD化とは何か？

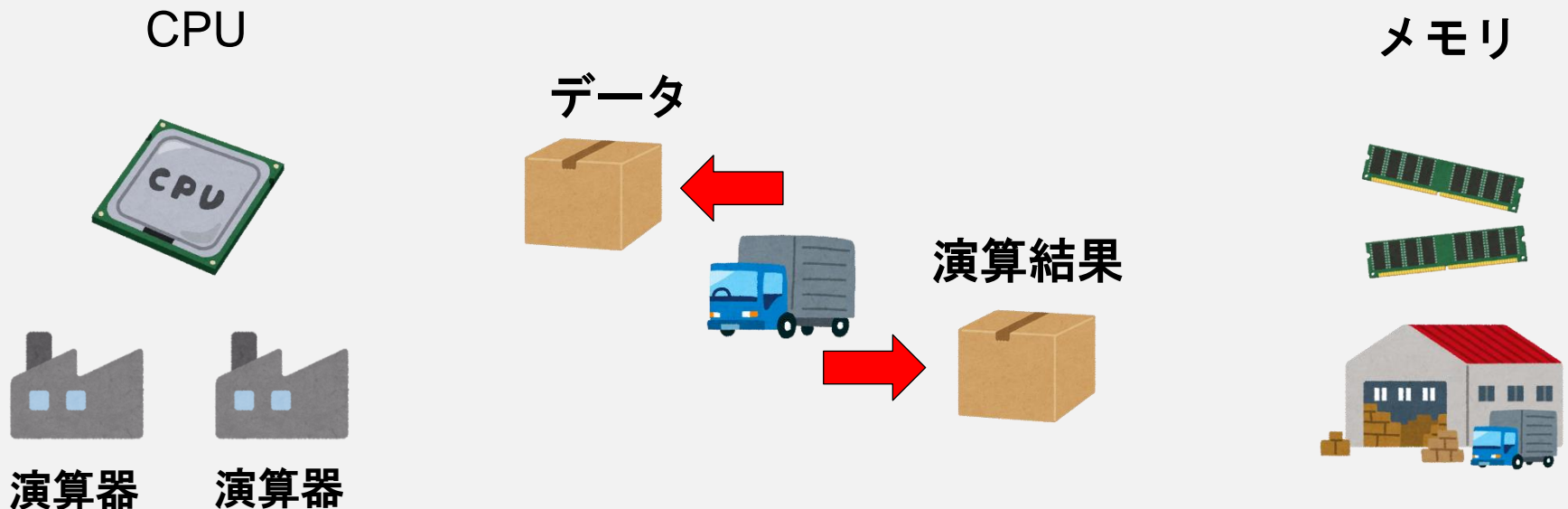
どうやってSIMD化するか？

計算機の仕組み(再掲)

計算機とは

メモリからデータと命令を取ってきて
演算機に投げ
演算結果をメモリに書き戻す

装置のこと

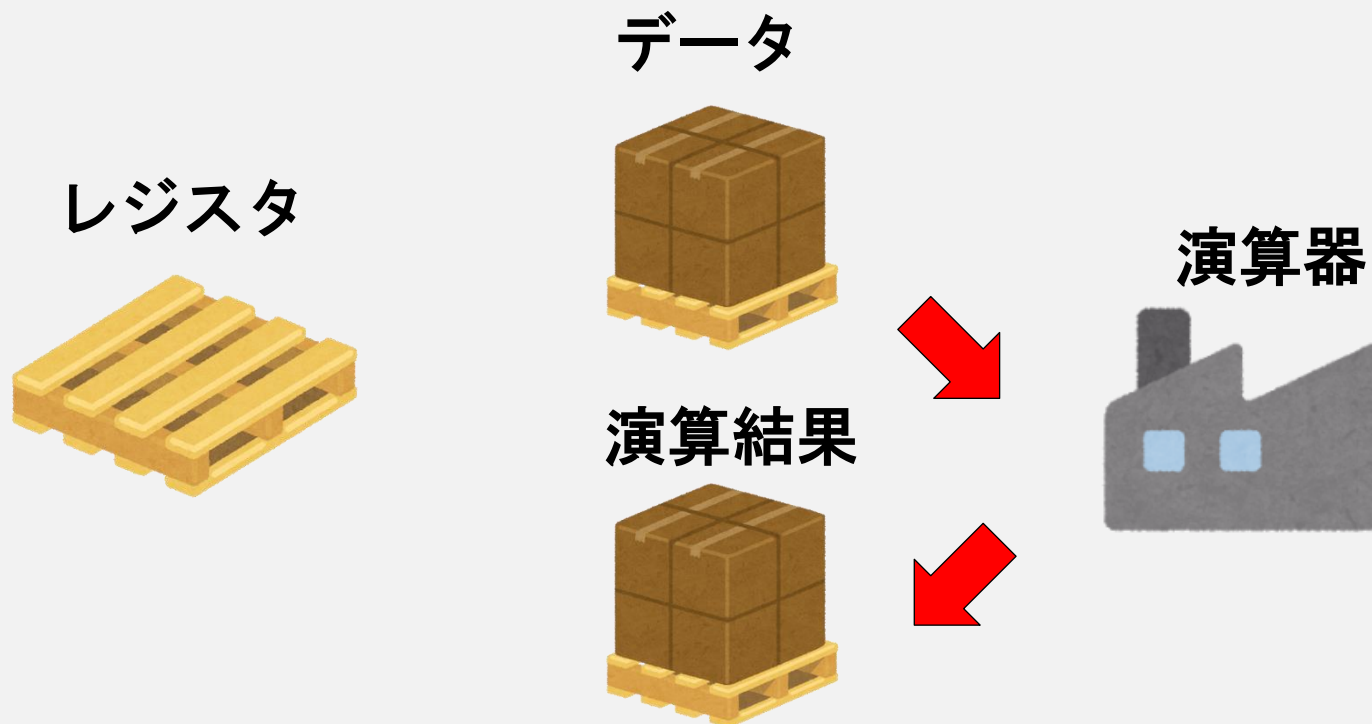


レジスタと演算器

計算機は

データをレジスタに載せて演算器に投げる

ことで計算する



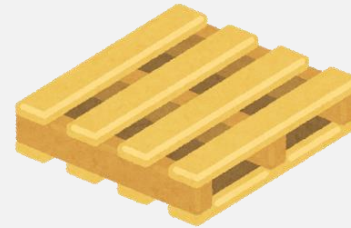
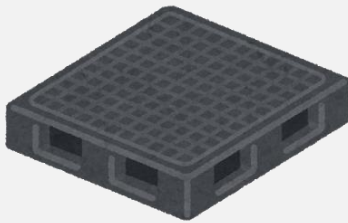
データをレジスタに載せて計算し、結果もレジスタに帰ってくる

レジスタと演算器

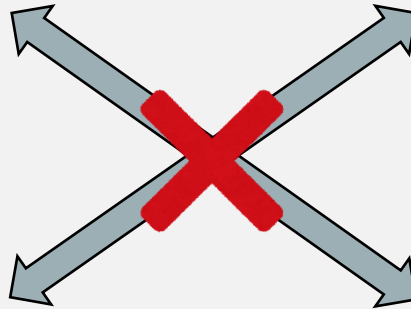
整数と浮動小数点数は異なるレジスタ、異なる演算器を使う

整数レジスタ

浮動小数点レジスタ



整数レジスタ
しか受け付けない



浮動小数点レジスタ
しか受け付けない



整数演算器

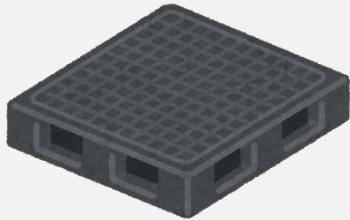
浮動小数演算器

※SIMDレジスタは異なるデータ型で共有することが多い

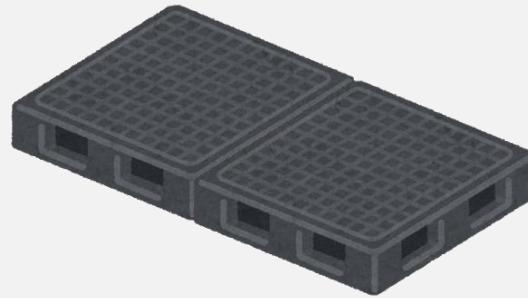
レジスタと演算器

レジスタには**長さ**がある
レジスタの長さは**ビット数**(bit)で表す

32ビットレジスタ



64ビットレジスタ



ハードウェアやソフトウェアの「ビット数」は
対応する**整数レジスタ**のビット数で決まる

ファミコン
8ビット



スーパーファミ
16ビット



プレステ
32ビット



NINTENDO64
64ビット



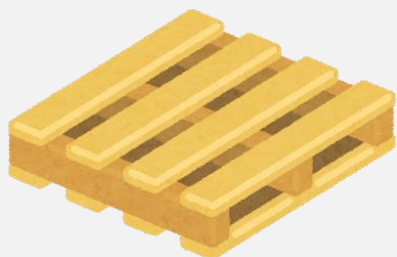
...

レジスタと演算器

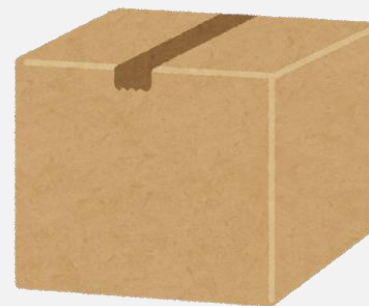
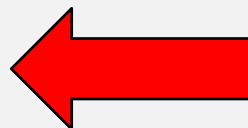
数値計算では、主に倍精度実数を用いる
倍精度実数は64ビットで表現される

64ビット浮動小数点レジスタ

倍精度実数 (64ビット)



ひとつ乗る

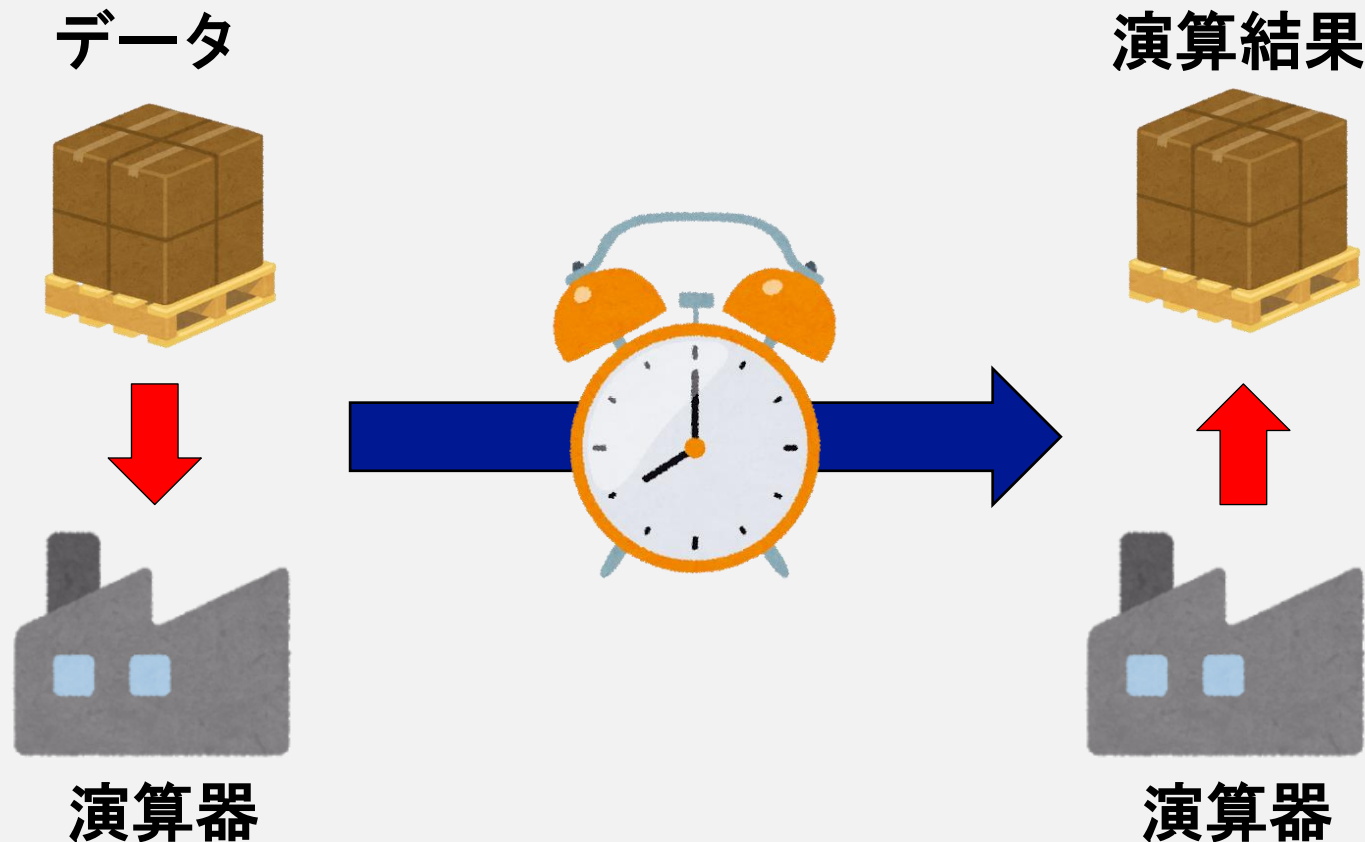


数値計算に用いるCPUの多くは64ビット整数レジスタと
64ビット浮動小数点レジスタを持つ

ただし、x86系のCPUは歴史的事情により64ビット浮動小数
点レジスタを持たず、128ビットSIMDレジスタを使う

パイプライン処理

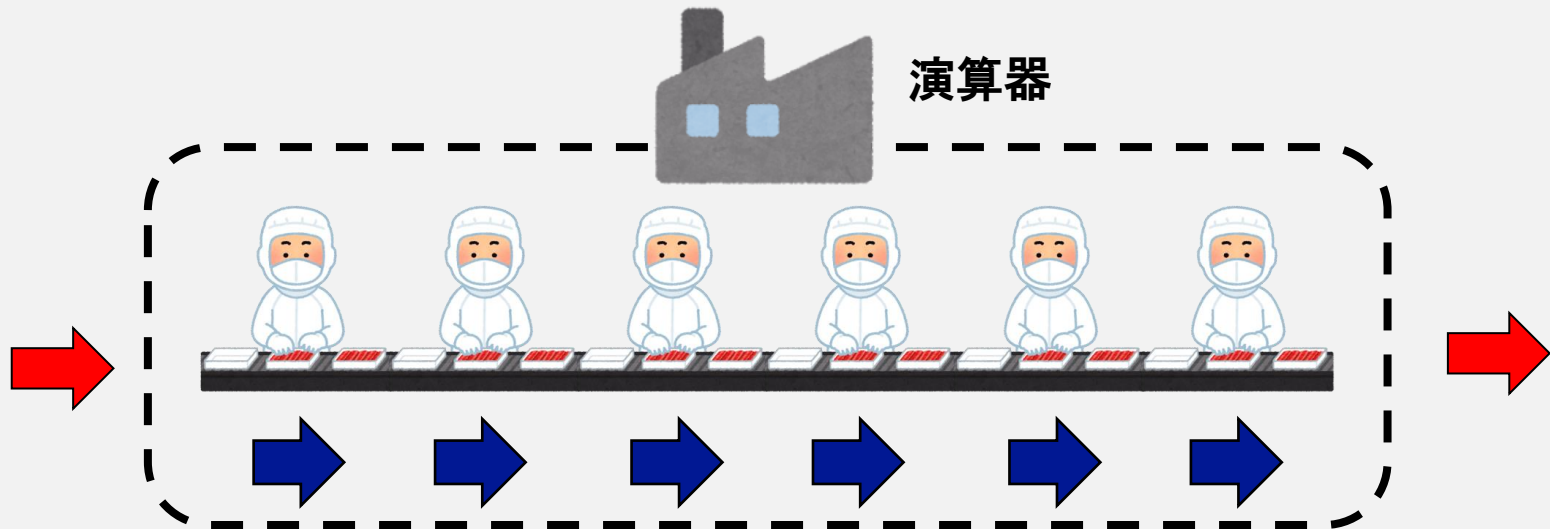
演算器に計算を投げてから結果が返ってくるまで時間がかかる
この時間を**レイテンシ**と呼び、サイクル数で測る



浮動小数点演算なら、加減乗算で3～6サイクル程度。除算は遅い(10～20サイクル)⁴⁸₇₁

パイプライン処理

全部で6工程ある作業を6人で分担すれば
1サイクルに1つ製品を作ることができる



1サイクルに1段右に動くベルトコンベア

演算器に入ってから出てくるまでは6サイクル(レイテンシ)
演算器から毎サイクル結果が出てくる(スループット)

CPUの動作周波数

パイプライン処理により、1サイクルに1回計算できるようになった

性能

=

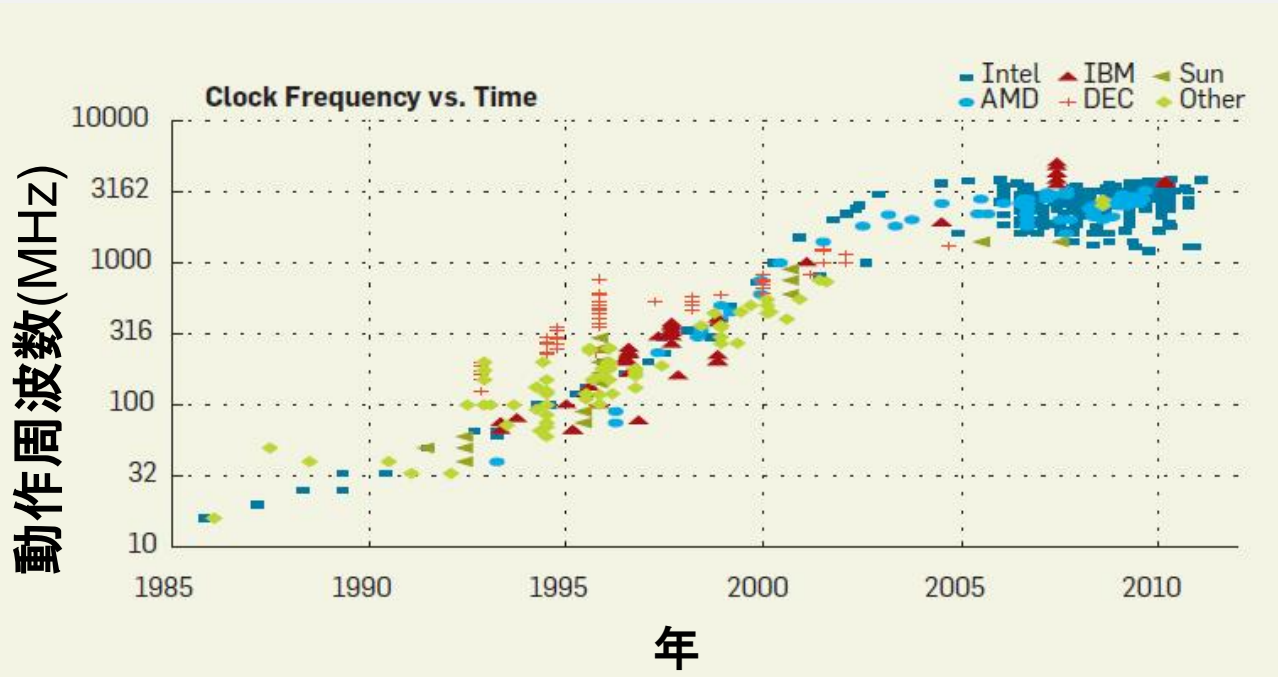
動作周波数

あとは動作周波数を上げれば上げるだけ性能があがる

・・・はずだった

CPUの動作周波数

CPUの動作周波数向上は2000年頃から頭打ちに



主に発熱が原因



<http://cacm.acm.org/magazines/2012/4/147359-cpu-db-recording-microprocessor-history/fulltext>

動作周波数を上げずに演算性能を上げたい

➡ 1サイクルに複数の命令を実行するしかない

解決案1: スーパースカラ

ハードウェアにがんばらせる

データフェッチ

依存関係チェック



データと命令を複数持ってきて
複数の生産ラインに振り分ける

演算機



演算機



この人が過労死する



命令の後方互換性を保てる



実行ユニットが増えると命令振り分けで死ぬ

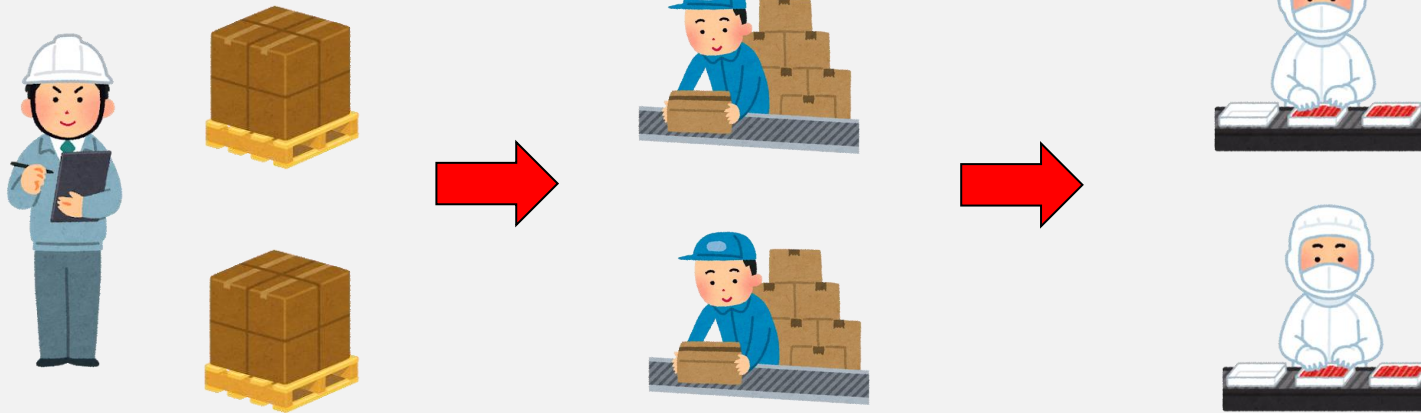
解決案2：VLIW

※ Very Long Instruction Word

ソフトウェアにがんばらせる

コンパイラがデータと
命令を並べておく

それをノーチェックで
演算機に流しこむ



依存関係チェックが不要→ハードウェアが簡単に



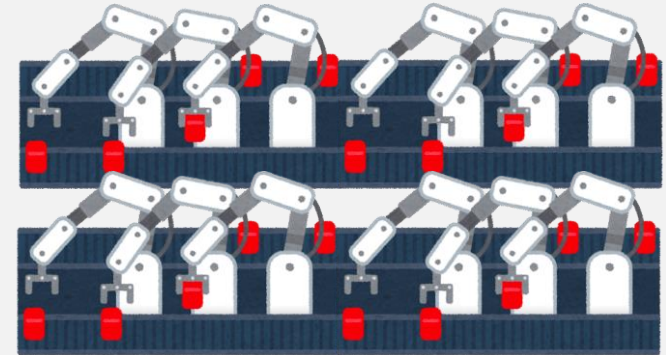
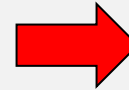
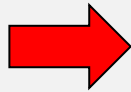
神のように賢いコンパイラが必要
後方互換性を失う

組み込み向けでは人気も
HPC向けとしてはほぼ絶滅

解決案3：SIMD

プログラマにがんばらせる

プログラマが
データを並べておく



一度に2～8演算を行う



ハードウェアは簡単
後方互換性も保てる



コンパイラによる自動SIMD化には限界がある
プログラムが大変

「なぜSIMDが必要か」のまとめ

パイプライン処理により、1サイクルに1命令実行できる
CPUの動作周波数は限界に達しており、これ以上あがらない



1サイクルに複数の命令を実行するしかない



ハードやソフトにがんばらせる方法も限界



人間ががんばるしかない ←イマココ

SIMDの仕組み

1時間に1個製品ができる製造ラインがある
ただし、コンベア速度はもう上がらない

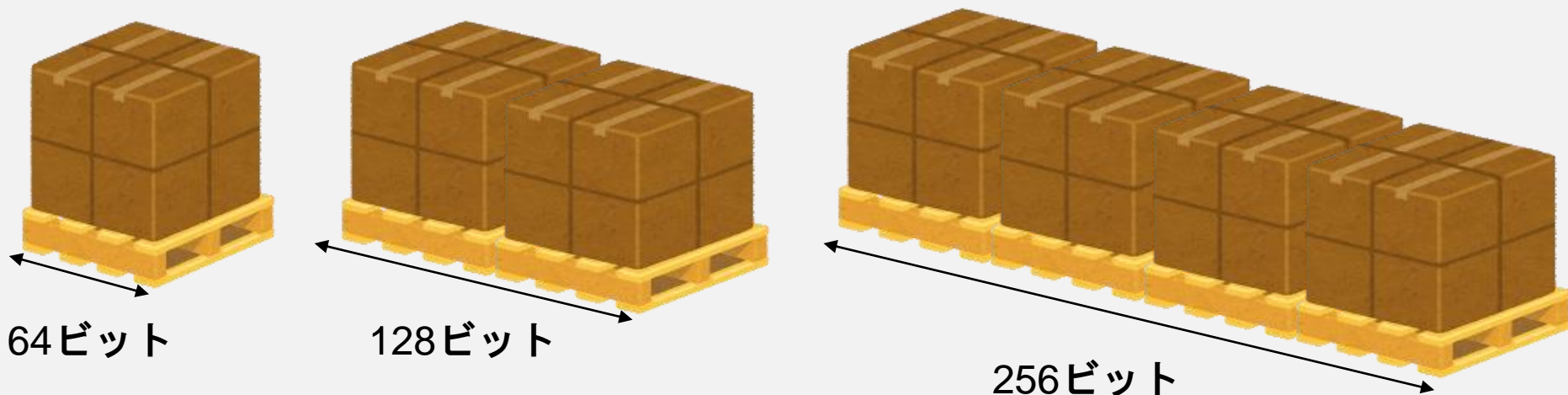


じゃあ製造ラインの幅を倍にすれば良いじゃん



SIMDの仕組み

- 64ビットレジスタは倍精度実数を1つ載せることができる
- 128ビットレジスタなら、2つ載せることができる
- 256ビットレジスタなら、4つ載せることができる



ビット幅が広く、データを一度に複数載せることができるレジスタをSIMDレジスタと呼ぶ

SIMDの仕組み

2つのレジスタに4つずつ値を載せる(256ビットの場合)

1	3	4	8
---	---	---	---

3	8	2	5
---	---	---	---

「同じ位置」同士で同時に**独立な**演算をする

1	3	4	8
---	---	---	---

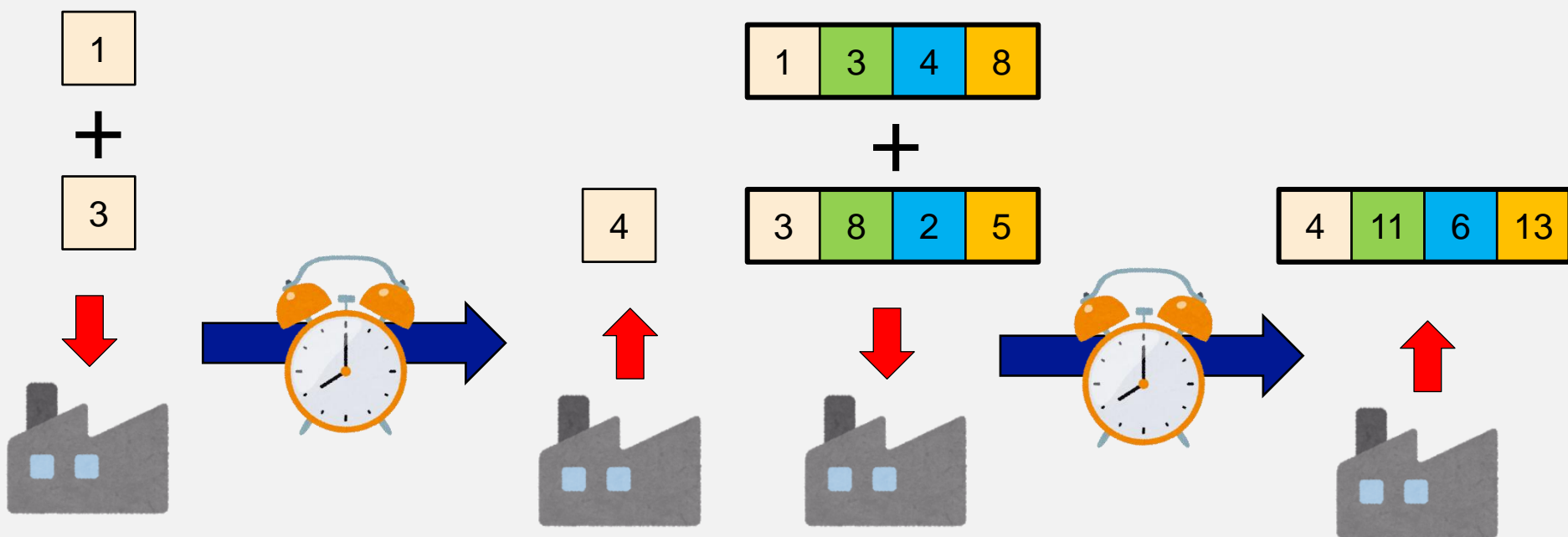
+

3	8	2	5
---	---	---	---

4	11	6	13
---	----	---	----

SIMDの仕組み

一つの計算をするのと同じ時間で
複数の計算を同時に実行できる

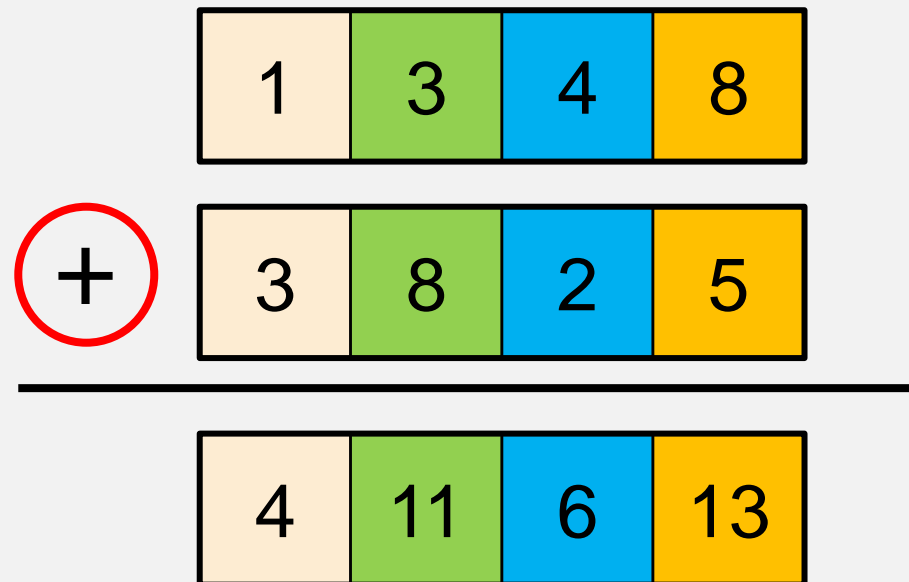


CPUの理論ピーク性能は「SIMD幅を使い切った時」の値

➡ SIMDが使えていなければ、数分の一の性能しか出せない

SIMDの仕組み

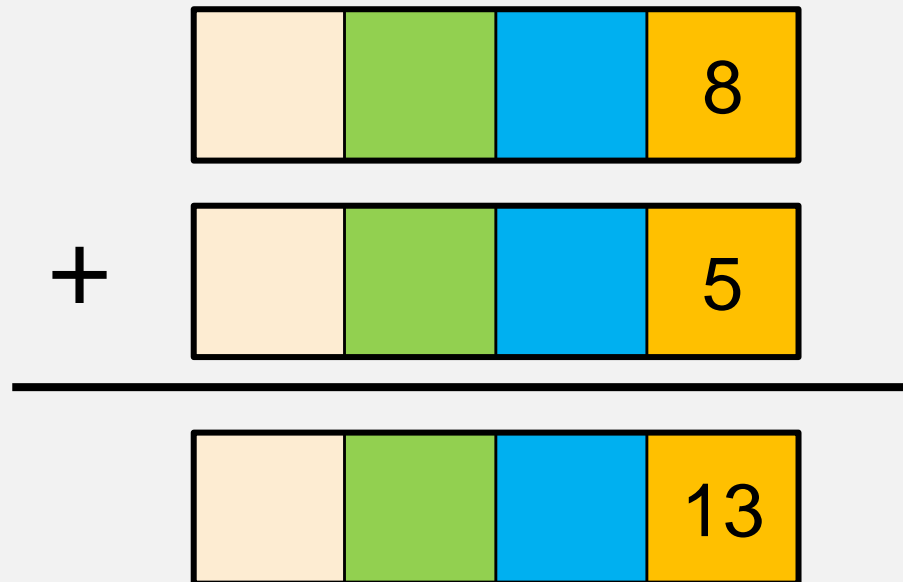
各位置ごとに異なる演算はできない



複数のデータ (Multiple Data)に
単一の演算 (Single Instruction)を実行するから
SIMD (Single Instruction Multiple Data)

SIMDの仕組み

使っていない位置は無駄になる



なるべく SIMDレジスタにデータを詰め込んで一度に計算したい

SIMD化とは

SIMDレジスタをうまく使えていないプログラムを
SIMDレジスタを活用するように修正し
性能を向上させること

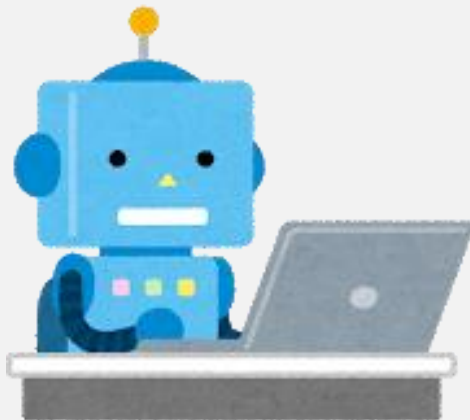


SIMDベクトル化 (SIMD Vectorization) と も

SIMD化とは

1. コンパイラにSIMD化してもらう
2. 自分でSIMD化する

基本的にこの二択



単純なSIMD化

注目する粒子(赤)と相互作用する粒子を
4つつまとめて計算する (馬鹿SIMD化)

4つの座標データをレジスタにロード

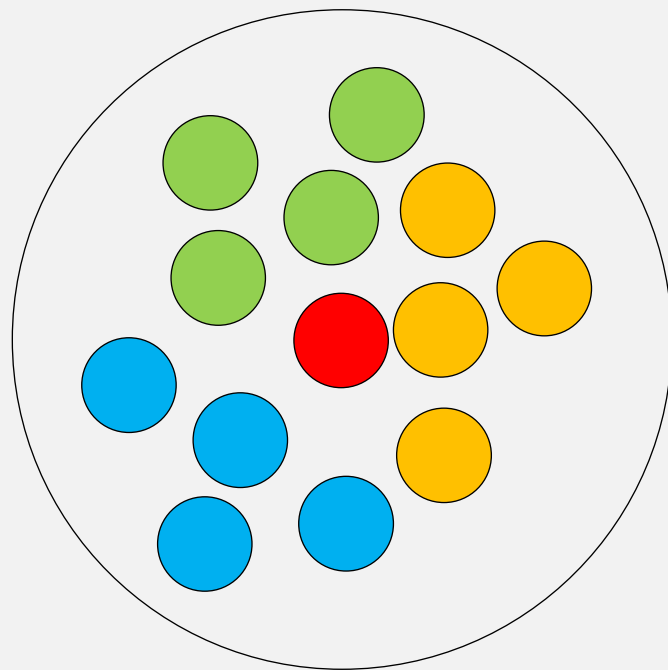
$$\hat{q}_x^i \quad \begin{array}{|c|c|c|c|} \hline q_x^{i+3} & q_x^{i+2} & q_x^{i+1} & q_x^i \\ \hline \end{array}$$

$$\hat{q}_x^j \quad \begin{array}{|c|c|c|c|} \hline q_x^{j+3} & q_x^{j+2} & q_x^{j+1} & q_x^j \\ \hline \end{array} \quad \text{※ y, z も同様}$$

$$\hat{dx} = \hat{q}_x^j - \hat{q}_x^i$$

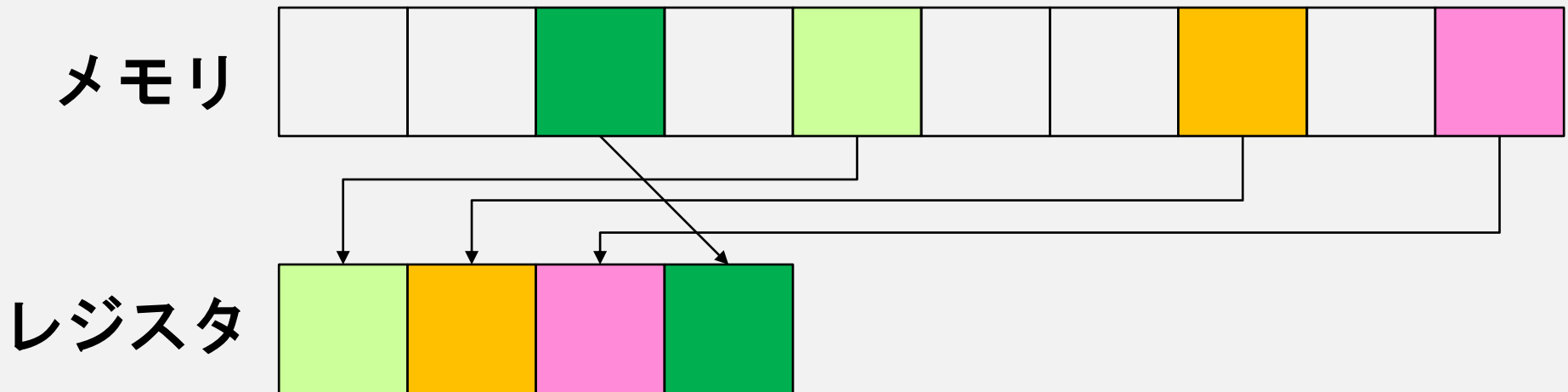
$$\hat{r}^2 = \hat{dx}^2 + \hat{dy}^2 + \hat{dz}^2$$

距離が4つパックされた



単純なSIMD化

相互作用粒子のインデックスは連続ではない
→ 4つのデータをバラバラにレジスタに転送



➡ 滅茶苦茶遅い

AVX2を用いたSIMD化

メモリ上にデータを以下のように並べる
(パディング付き Array of Structure, AoS)

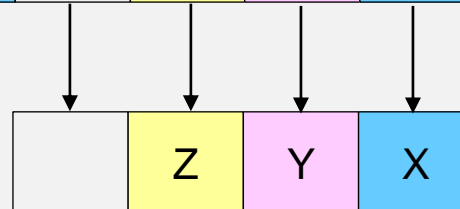


3成分のデータを256bitレジスタに一発ロードできる
ただし1成分(64bit)は無駄になる

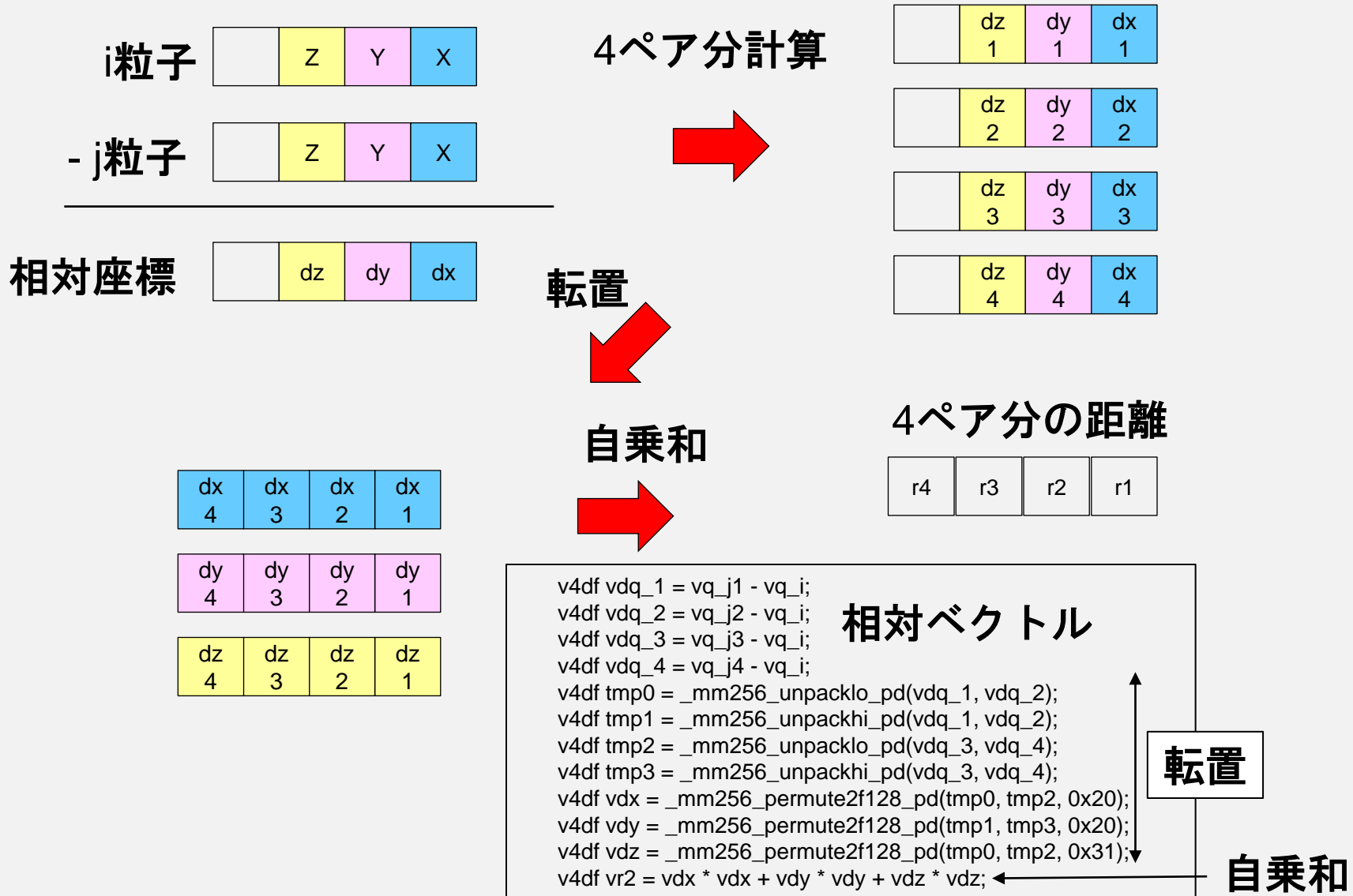
メモリ



レジスタ

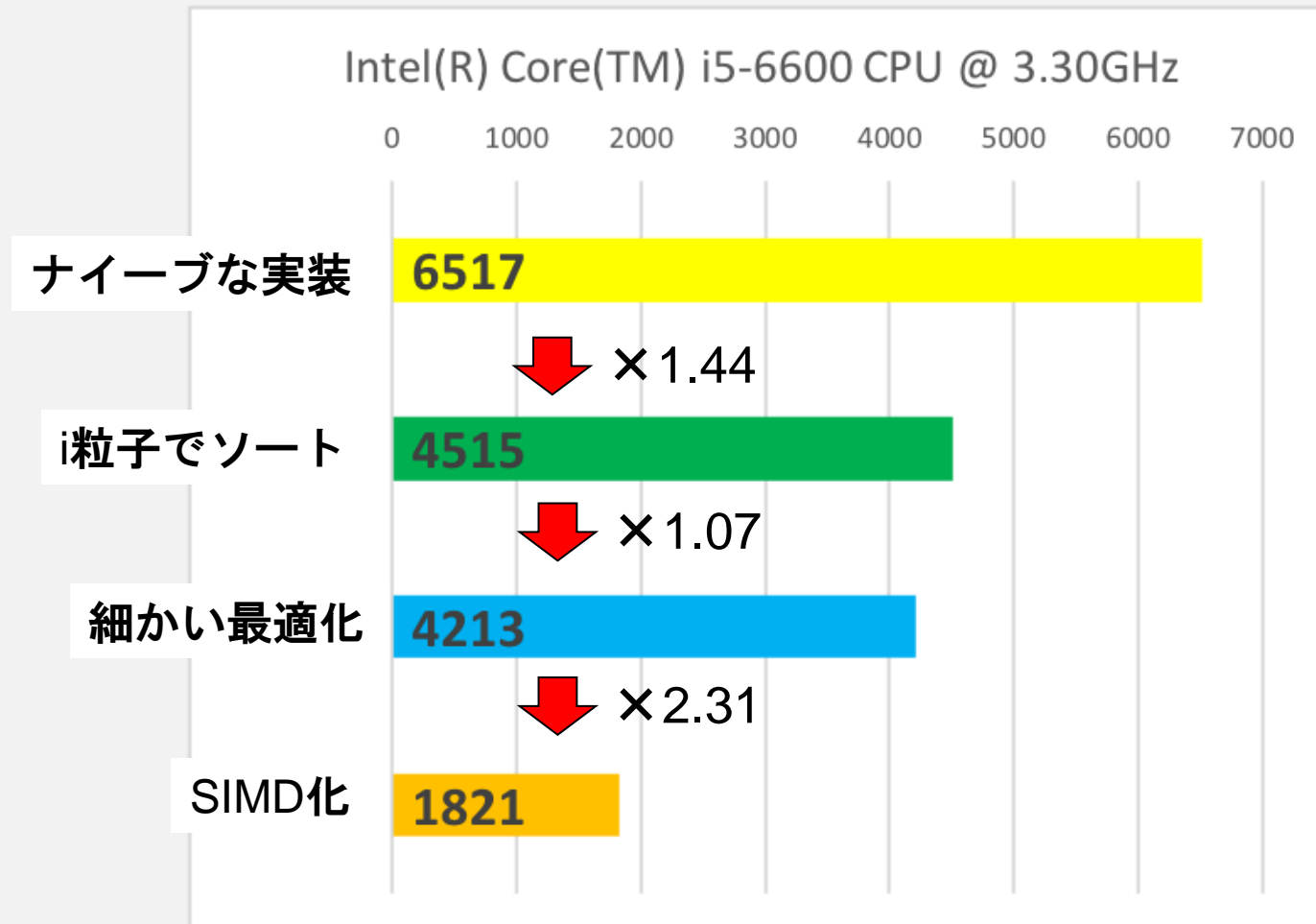


AVX2を用いたSIMD化



AVX2を用いたSIMD化

原子数: 12万, 数密度: 1.0, カットオフ: 3.0
100回の力の計算にかかった時間 [ms]



SIMD化のまとめ

- SIMDは**1サイクル**に**複数の命令**を実行するための工夫の一つ
- SIMDは**幅広レジスタ**に複数のデータを載せて同時に**独立**な計算を実行する
- SIMD化の目的はSIMDレジスタを活用することで**性能を向上**させること
- SIMD化は、**コンパイラに任せる**方法と、**自分で書く**方法がある
- SIMD化により数倍性能が変わるが、データ構造を変えたり、アセンブリを書いたりする必要がある

本日の講義のまとめ

- 現代のCPUは、構造が複雑化、多階層化しており、性能を出すのが困難
- 特にメモリアクセスがボトルネックになりやすいため、性能を出すためにはキャッシュの有効活用が必要
- 現代CPUの性能はSIMD幅を使い切った場合の値であり、SIMDレジスタを活用できないコードは性能が出ない
- 単純なコードであっても、高い計算効率を達成するには、アルゴリズム(ソフトウェア)とハードウェアの両方の知識が必要

講義全体のまとめ

シミュレーションとは宣言である

- 世の中がこのモデルで表される
- 知りたい物理量はこう定義する

何が定義で、何が非自明かを考える

効率的なシミュレーションコードを書くのは難しい

