第9回
高速化チューニングとその関連技術2

渡辺宙志

慶應義塾大学理工学部
物理情報工学科

Outline

1. 計算機の仕組み
2. プロファイラの使い方
3. メモリアクセス最適化
4. CPUチューニング

注意

今日話すことは、おそらく今後の人生にほとんど役にたちません

楽しんで

ただ、「こういうことをやる人々がいる」ということだけ知っておいてください

高速化とは (1/2)

高速化とは？

アルゴリズムを変えずに実装方法を工夫して実行時間を短縮すること

※本講義内での定義

➡ チューニング

遅いアルゴリズムを実装で高速化しても無意味

アルゴリズムが枯れているのが前提

高速化とは (2/2)

チューニングとは？

コンパイラや計算機にやさしいコードを書く事

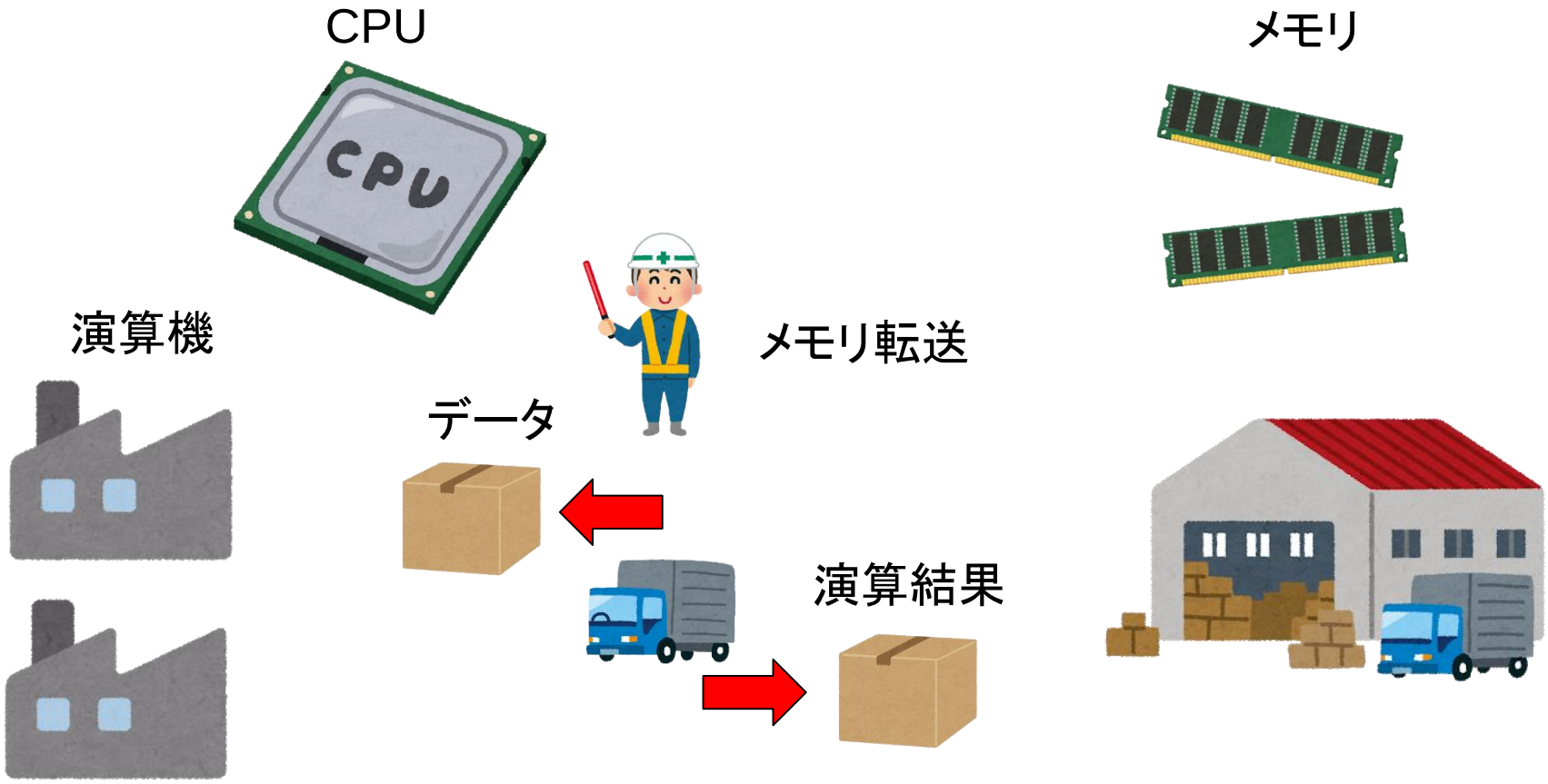
計算機の仕組みをある程度理解しておく必要がある

計算機の仕組み

チューニングをする前に知っておくべきこと

計算機とは何か？

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

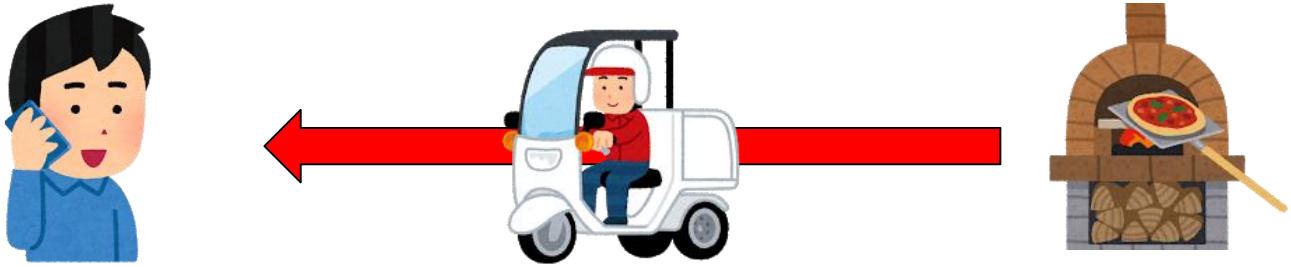


近年の計算機はメモリ転送がボトルネック

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



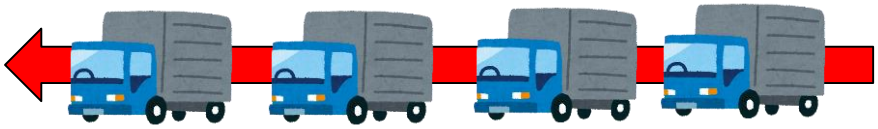
キャッシュアクセスで数サイクル～数十サイクル
メモリアクセスで数百サイクル程度

メモリ空間にランダムアクセスすると性能が出せない

※性能が出ない＝CPUが遊ぶ

スループット

計算能力に比較したデータ転送能力、いわゆるBytes/Flops (B/F)

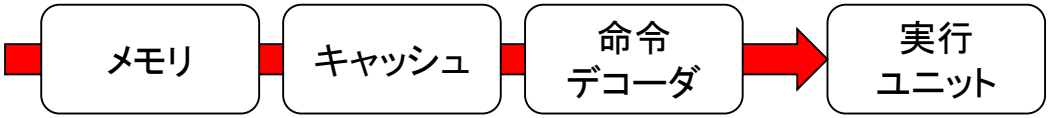


計算が「軽い」問題は、本質的に性能が出せない

パイプライン処理

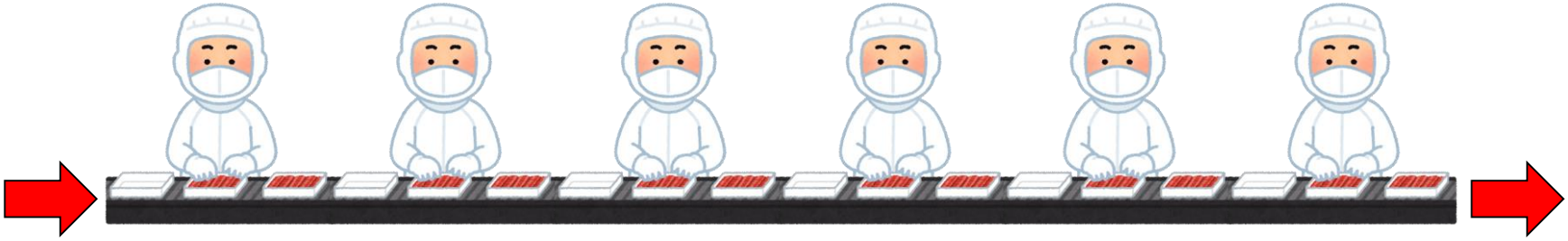
計算機がやること

メモリからデータと命令を取ってきて、実行ユニットに渡し、メモリに書き戻す



ひとつの四則演算に 3〜6サイクル程度かかる

パイプライン処理



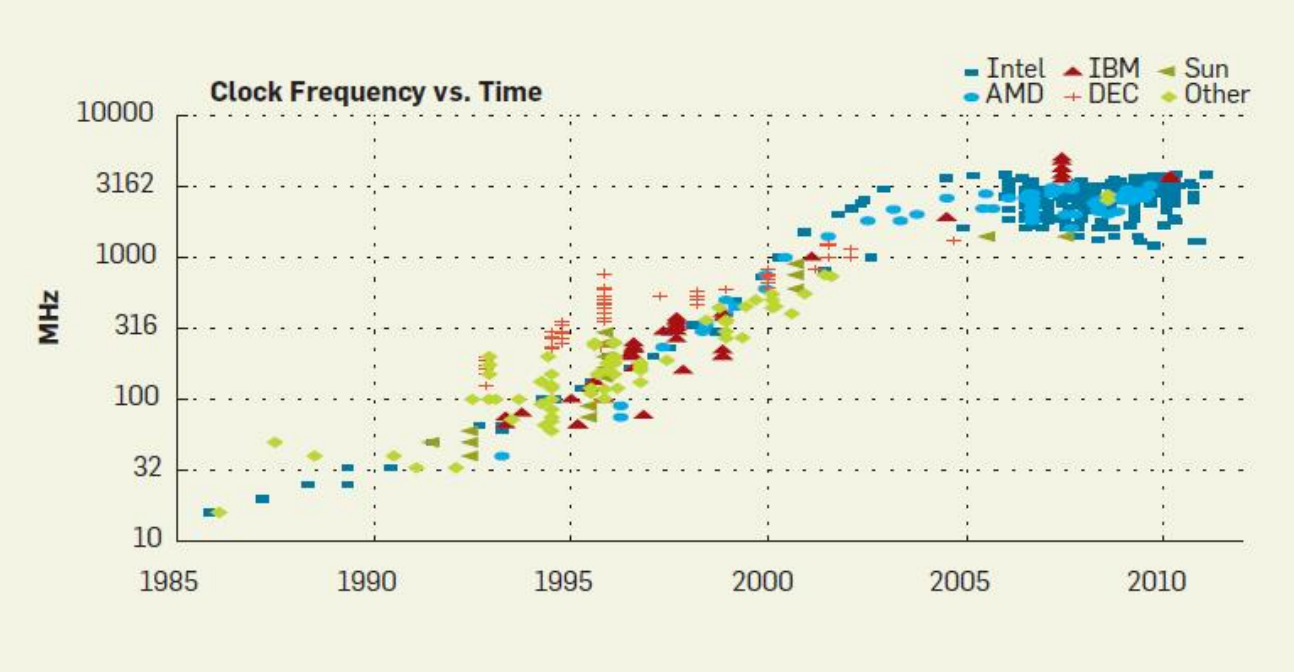
ベルトコンベア方式を採用
1つの演算には6サイクルかかるが、1000個の演算に1006サイクル
→ おおむね1サイクルに1つの計算が可能

性能 ＝ 動作周波数

CPUの動作周波数

パイプライン処理により、1サイクルに一回の演算が可能
後は動作周波数を上げれば上げるほど性能が上がる

➡ CPUの動作周波数向上は2000年頃から頭打ちに



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
➡ 一つのサイクルで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
振り分け

演算機

演算機

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

命令の後方互換性を保てる

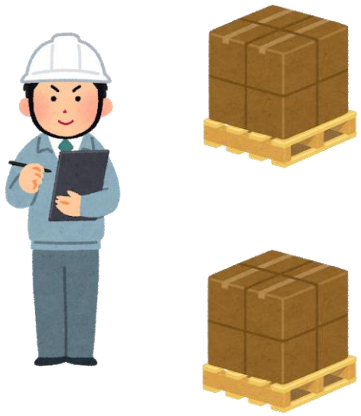
実行ユニットが増えると命令振り分けで死ぬ

解決案2: VLIW

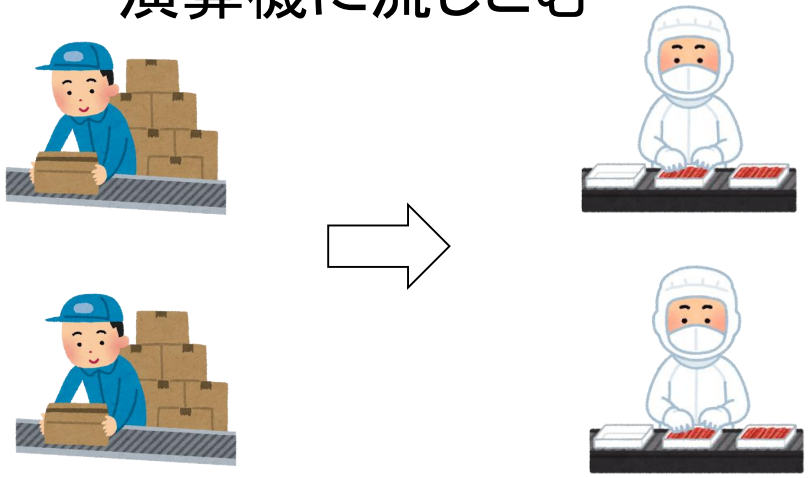
※ Very Long Instruction Word

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になり、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

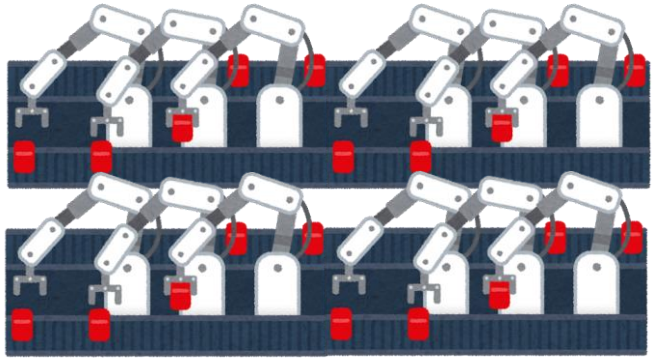
※ Single Instruction Multiple Data

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

依存関係をチェックせずに
ラインに流し込む

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

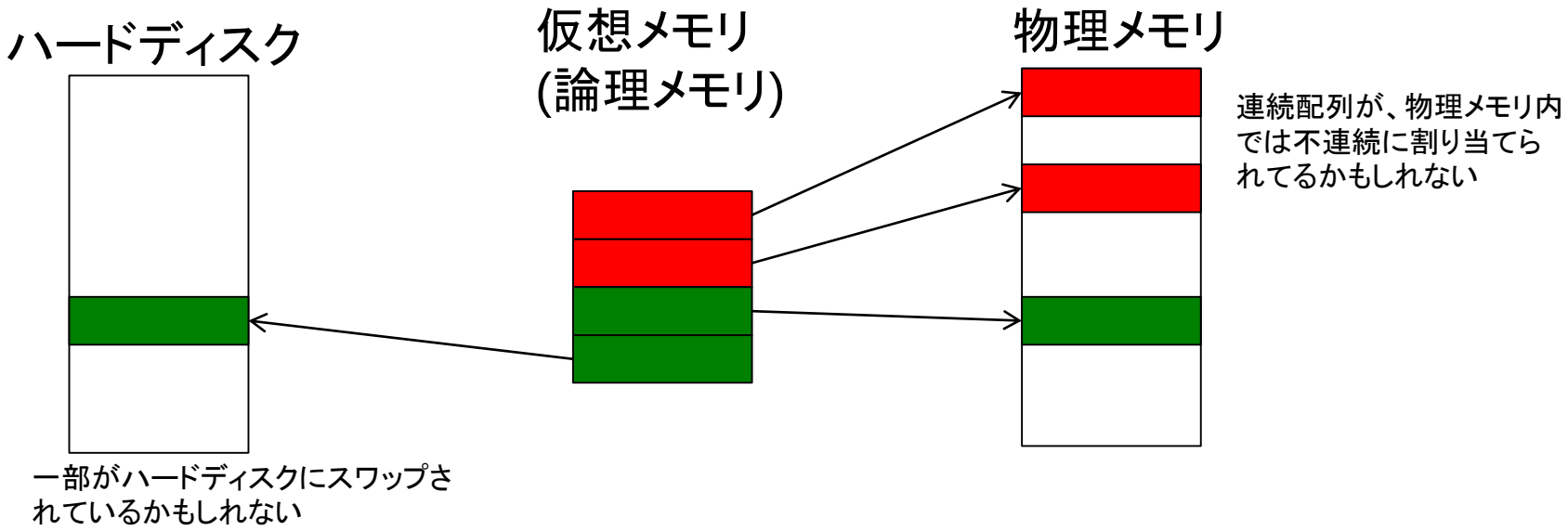
コンパイラによる自動SIMD化には限界がある
プログラムが大変

現在はスーパースカラ＋SIMDが主流
実行ユニットの数は増えず、SIMD幅が増えていく傾向

仮想メモリとページング (1/5)

仮想メモリとは

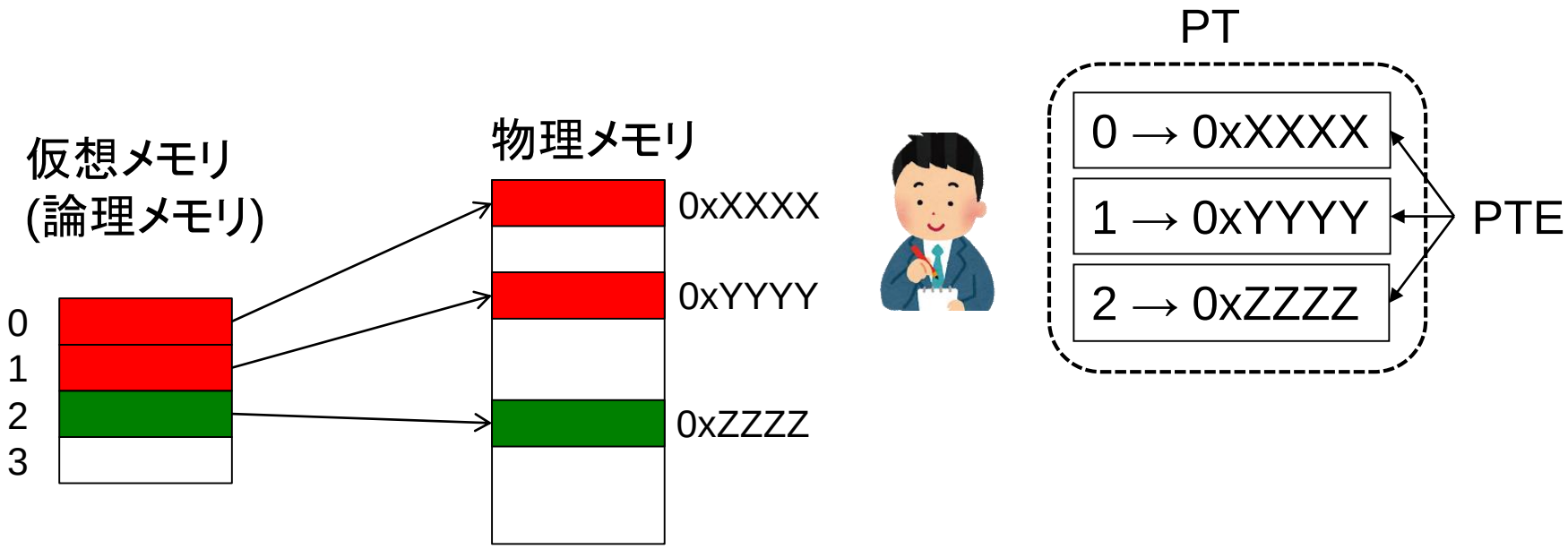
物理メモリと論理メモリをわけ、ページという単位で管理するメモリ管理方法



不連続なメモリ空間を論理的には連続に見せることができる
スワッピングが可能になる

仮想メモリとページング (2/5)

ページテーブル (PT): 論理ページと物理メモリの対応表
ページテーブルエントリ (PTE) : 論理ページと物理メモリの対応レコード



通常のページサイズは4096バイト→1GBは26万2144個のPTEで管理

PTEをすべてキャッシュに載せるのは不可能 → メモリ上で管理

仮想メモリとページング (3/5)

```
printf("%d¥n", a[i]);
```

(1) a[i]の場所おしえてください

ページテーブル

(2) 0xXXX番にあります

(3) a[i]の値を教えてください

配列のある場所

(4) 中身を送ります

一回データ要求するのに二回メモリアクセスする

仮想メモリとページング (4/5)

```
printf("%d¥n", a[i]);
```

(1) a[i]はさっき調べたっけ

TLB

(2) 0xXXXにあるんだった

ページテーブル

(3) a[i]の値を教えてください

配列のある場所

(4) 中身を送ります

TLB (Translation Lookaside Buffer)
ページテーブルエントリ専用のキャッシュ

仮想メモリとページング (5/5)

TLBミス

TLBもキャッシュなので、キャッシュミスがおきる
→ TLBミス、TLBスラッシング

a[i]の場所のメモある？

なかった

ラージページ

ページがたくさんあると、ページテーブルもたくさん用意しなければいけない
ページを大きくすれば、ページの数が減る

例えばページサイズを2MBにすると、1GBは512エントリですむ

cf: 4096バイトなら26万2144エントリ

TLBミスが減る

スワップできない
メモリ効率が悪くなる

計算機の仕組みのまとめ

- 計算機の仕組みは年々複雑化している
- 多くの場合はメモリ転送がボトルネック

ボトルネックは「レイテンシ」か「スループット」

本気でチューニングするなら

- TLBやラージページなども知っておいたほうが良い

プロファイラの使い方

チューニングの前にはプロファイリング

プログラムのホットスポット

ホットスポットとは

プログラムの実行において、もっとも時間がかかっている場所

多くの場合、一部のルーチンが計算時間の大半を占める
(80:20の法則)

チューニングの方針

- ☑ ホットスポットを探す

  チューニング前に、どの程度高速化できるはずかを見積もる
  チューニングは、ホットスポットのみに注力する

- ☑ ホットスポットでないルーチンの最適化は行わない

  高速化、最適化はバグの温床となるため
  ホットスポットでないルーチンは、速度より可読性を重視

プロファイラ

プロファイラとは

プログラムの実行プロファイルを調べるツール

サンプリング

どのルーチンがどれだけの計算時間を使っているかを調べる
ルーチン単位で調査
ホットスポットを探すのに利用

イベント取得

どんな命令が発効され、どこでCPUが待っているかを調べる
範囲単位で調査 (プログラム全体、ユーザ指定)
高速化の指針を探るのに利用

perfの使い方(サンプリング)

perfとは

Linuxのパフォーマンス解析ツール
プログラムの再コンパイル不要

使い方

```
$ perf record ./a.out
$ perf report
```

出力

手元のMDコードを食わせてみた結果

```
Samples: 155K of event 'cycles', Event count (approx.): 139410304992
 85.93%  mdacp  mdacp                [.] ForceCalculator::CalculateForceNext(Var
  5.26%  mdacp  mdacp                [.] MeshList::SearchMesh(int, Variables*, S
  2.10%  mdacp  libgomp.so.1.0.0     [.] 0x000000000000f05c
  1.50%  mdacp  mdacp                [.] ForceCalculator::UpdatePositionHalf(Var
  1.08%  mdacp  mdacp                [.] MDUnit::MakeBufferForBorderParticles(in
  0.92%  mdacp  mdacp                [.] PotentialEnergyObserver::Observe(Variab
  0.72%  mdacp  mdacp                [.] VirialObserver::Observe(Variables*, Mes
  0.53%  mdacp  mdacp                [.] MeshList::MakeList(Variables*, Simulati
  0.45%  mdacp  mdacp                [.] PairList::CheckByDisplacement(Variables
```

86%が力の計算
5%がペアリストの作成　といったことがわかる

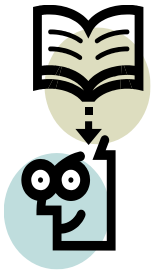
結果の解釈 (サンプリング)

一部のルーチンが80%以上の計算時間を占めている
→そのルーチンがホットスポットなので、高速化を試みる

多数のルーチンが計算時間をほぼ均等に使っている
→最適化をあきらめる

あきらめたらそこで試合終了じゃないんですか？

世の中あきらめも肝心です



※最適化は、常に費用対効果を考えること

プロファイラ(イベント取得型)

CPUがイベントをカウントする機能を持っている時に使える
(Hardware Counter)

取得可能な主なイベント:
- 実行命令 (MIPS)
- 浮動小数点演算 (FLOPS)
- サイクルあたりの実行命令数 (IPC)

} こういうのに気を取られがちだが

- キャッシュミス
- バリア待ち ←個人的にはこっちが重要だと思う
- 演算待ち

perfの使い方(イベントカウント 1/2)

使い方

$ perf stat ./a.out

出力

```
Performance counter stats for './a.out':

     38711.089599 task-clock                #    3.999 CPUs utilized
            4,139 context-switches          #    0.107 K/sec
                5 cpu-migrations            #    0.000 K/sec
            3,168 page-faults               #    0.082 K/sec
  138,970,653,568 cycles                    #    3.590 GHz
   56,608,378,698 stalled-cycles-frontend   #   40.73% frontend cycles idle
   16,444,667,475 stalled-cycles-backend    #   11.83% backend  cycles idle
  233,333,242,452 instructions              #    1.68  insns per cycle
                                            #    0.24  stalled cycles per insn
   11,279,884,524 branches                  #  291.386 M/sec
    1,111,038,464 branch-misses             #    9.85% of all branches

      9.681346735 seconds time elapsed
```

4CPUコアを利用

IPC = 1.68

分岐予測ミス

取得できるイベントは perf listで確認

perfの使い方(イベントカウント 2/2)

使い方 (イベント指定)

$ perf stat -e cache-misses,cache-references ./a.out

キャッシュミスの回数　　キャッシュの参照回数

出力

```
Performance counter stats for './a.out':

     1,019,158 cache-misses              #    5.927 % of all cache refs
    17,194,391 cache-references

     0.444152327 seconds time elapsed
```

17,194,391回キャッシュを参照にいって、そのうち
1,019,158 回キャッシュミスしたから、
キャッシュミス率は 5.927%だよ、という意味

結果の解釈 (イベントカウンタ)

バリア同期待ち

OpenMPのスレッド間のロードインバランスが原因
自動並列化を使ったりするとよく発生
対処：自分でOpenMPを使ってちゃんと考えて書く(それができれば苦労はしないが)

キャッシュ待ち

浮動小数点キャッシュ待ち
対処：メモリ配置の最適化 (ブロック化、連続アクセス、パディング・・・)
ただし、本質的に演算が軽い時には対処が難しい

演算待ち

浮動小数点演算待ち
A=B+C
D=A*E ←この演算は、前の演算が終わらないと実行できない
対処：アルゴリズムの見直し (それができれば略)

SIMD化率が低い

あきらめましょう
それでも対処したい人へ： 気合でなんとかする

プロファイリングで遅いところと原因がわかった？
よろしい、ではチューニングだ

メモリアクセス最適化

計算量を増やしてでもメモリアクセスを減らす

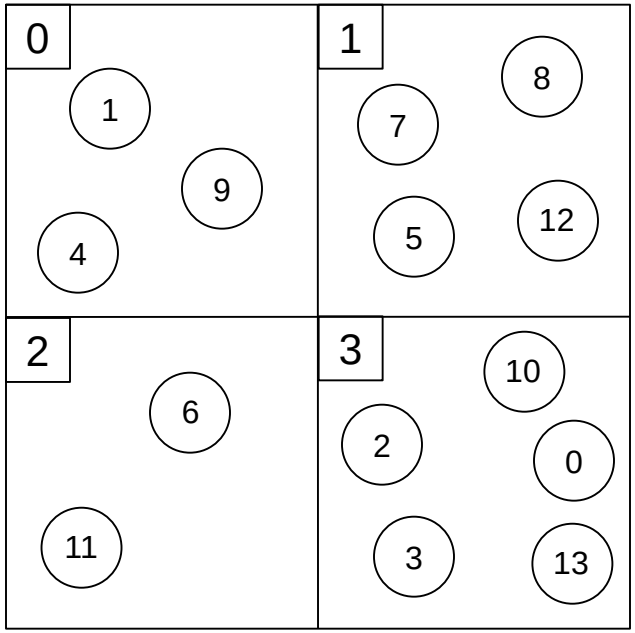
メモリ最適化1 セル情報の一次元実装 (1/2)

セル情報

相互作用粒子の探索で空間をセルに分割し、それぞれに登録する
ナイーブな実装→ 多次元配列

```
int GridParticleNumber[4];  どのセルに何個粒子が存在するか
int GridParticleIndex[4][10]; セルに入っている粒子番号
```

i番目のセルに入っているj番目の粒子番号 = GridParticleIndex[i][j];



GridParticleIndexの中身はほとんど空

1	4	9							
7	5	8	12						
6	11								
0	2	3	10	13					

広いメモリ空間の一部しか使っていない
→ キャッシュミスの多発

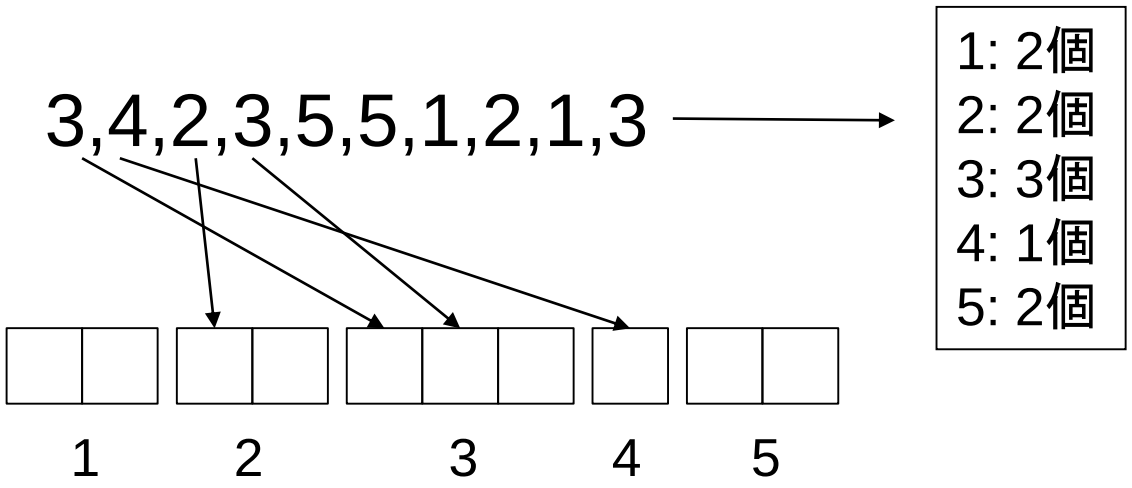
分布数えソート (Counting Sort)

原理

- あらかじめデータがとり得る値(離散)がわかっている時に使える
- 計算量はO(N)

アルゴリズム

- ある値をとるデータがそれぞれ何回出現するかを数える O(N)
- ある値を取るデータが入る予定の場所を調べる
- データを順番に「入る予定の場所」に詰めていく O(N)

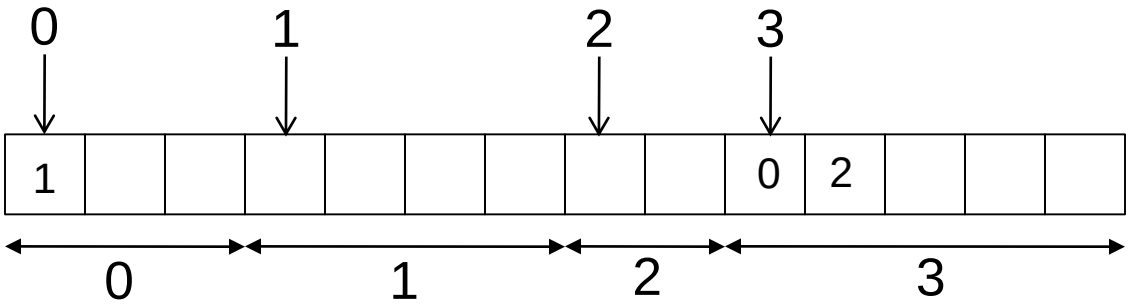


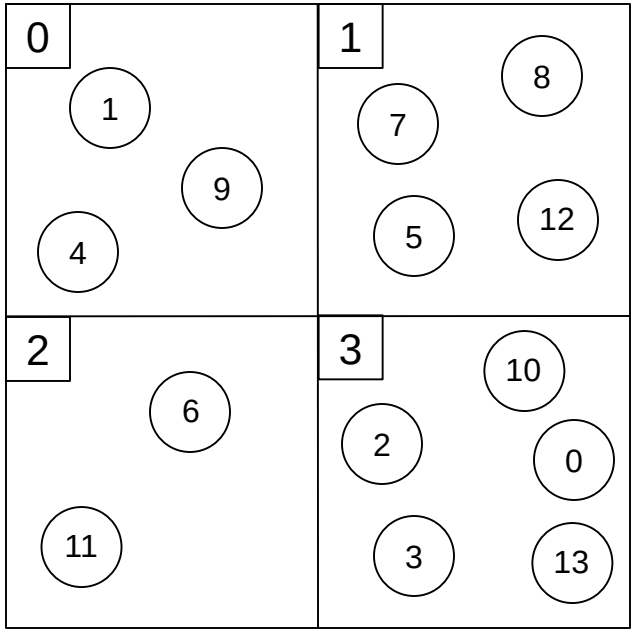
最適化のみならず、知っていると便利なソートアルゴリズムの一種

メモリ最適化1 セル情報の一次元実装 (2/2)

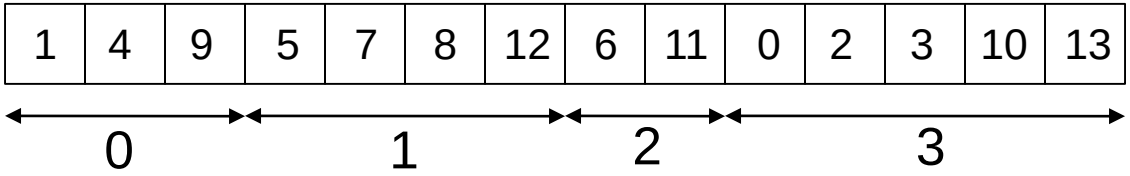
一次元実装

1. 粒子数と同じサイズの配列を用意する
2. どのセルに何個粒子が入る予定かを調べる
3. セルの先頭位置にポインタを置く
4. 粒子を配列にいれるたびにポインタをずらす
5. 全ての粒子について上記の操作をしたら完成





完成した配列 (所属セル番号が主キー、粒子番号が副キーのソート)



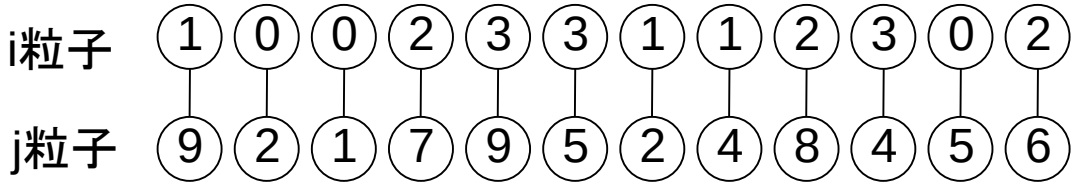
メモリを密に使っている (キャッシュ効率の向上)

メモリ最適化2 相互作用ペアソート (1/2)

力の計算とペアリスト

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

得られた相互作用ペア



相互作用ペアの配列表現

i粒子	1	0	0	2	3	3	1	1	2	3	0	2
j粒子	9	2	1	7	9	5	2	4	8	4	5	6

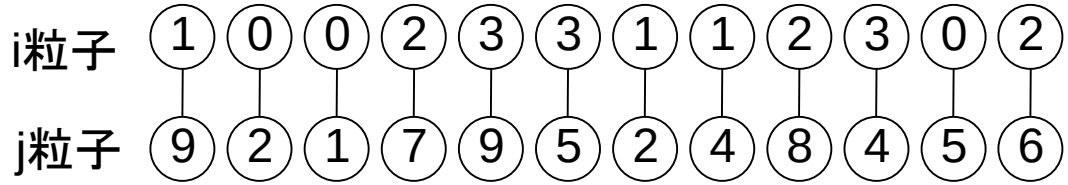
このまま計算すると

2個の粒子の座標を取得 (48Byte) 計算した運動量の書き戻し (48Byte)
力の計算が50演算程度とすると、B/F～2.0を要求 (※キャッシュを無視している)

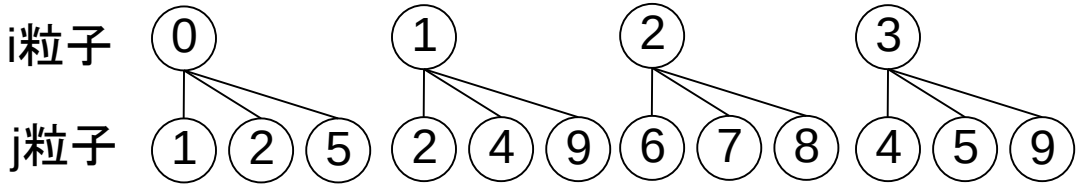
メモリ最適化2 相互作用ペアソート (2/2)

相互作用相手でソート

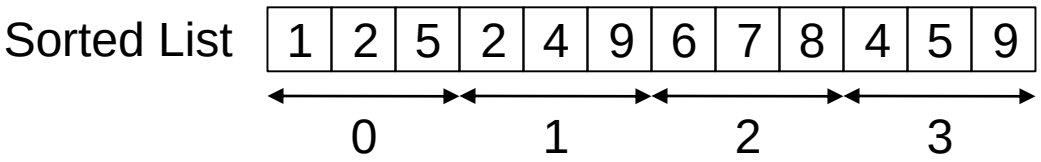
相互作用ペア



i粒子でソート



配列表現



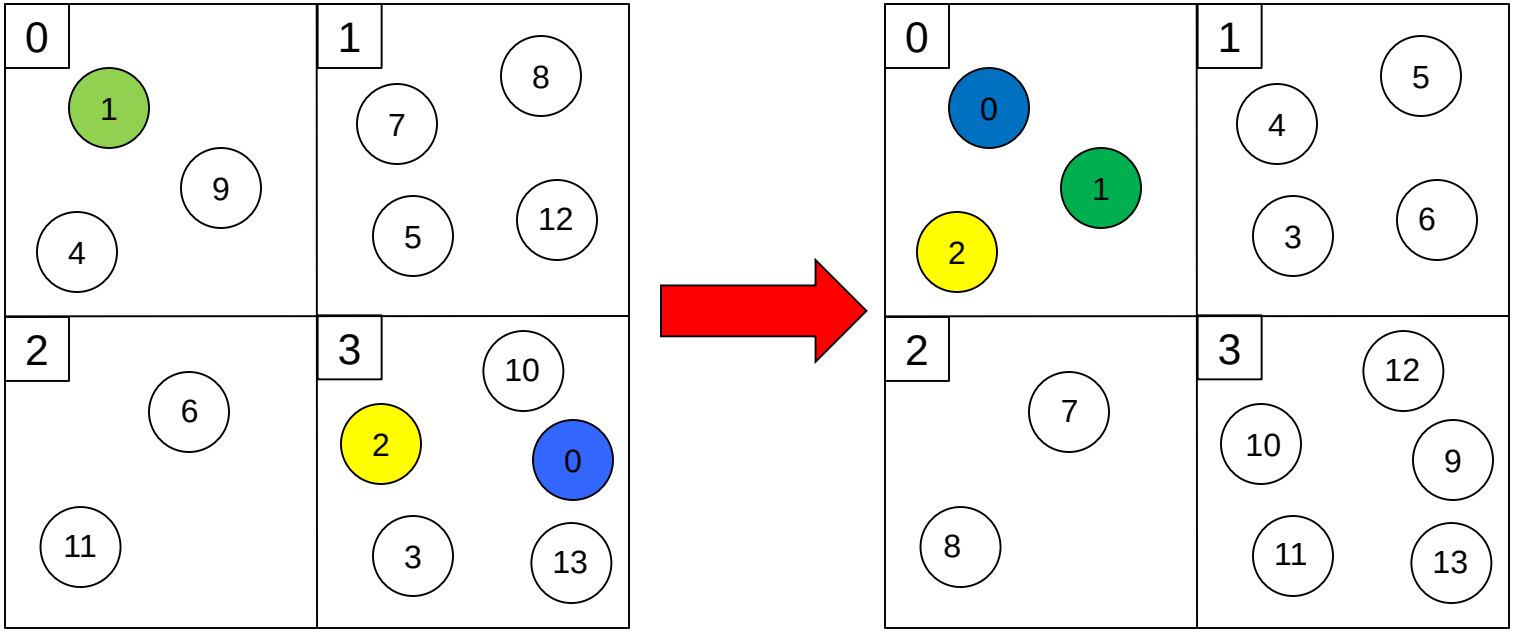
i粒子をキーとした分布数えソート

i粒子の情報がレジスタにのる
→ 読み込み、書き込みがj粒子のみ
→ メモリアクセスが半分に

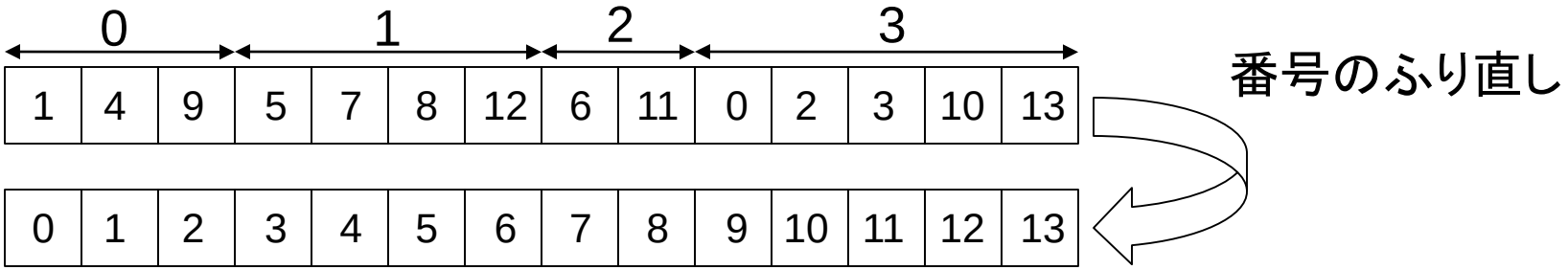
メモリ最適化3 空間ソート (1/2)

空間ソート

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

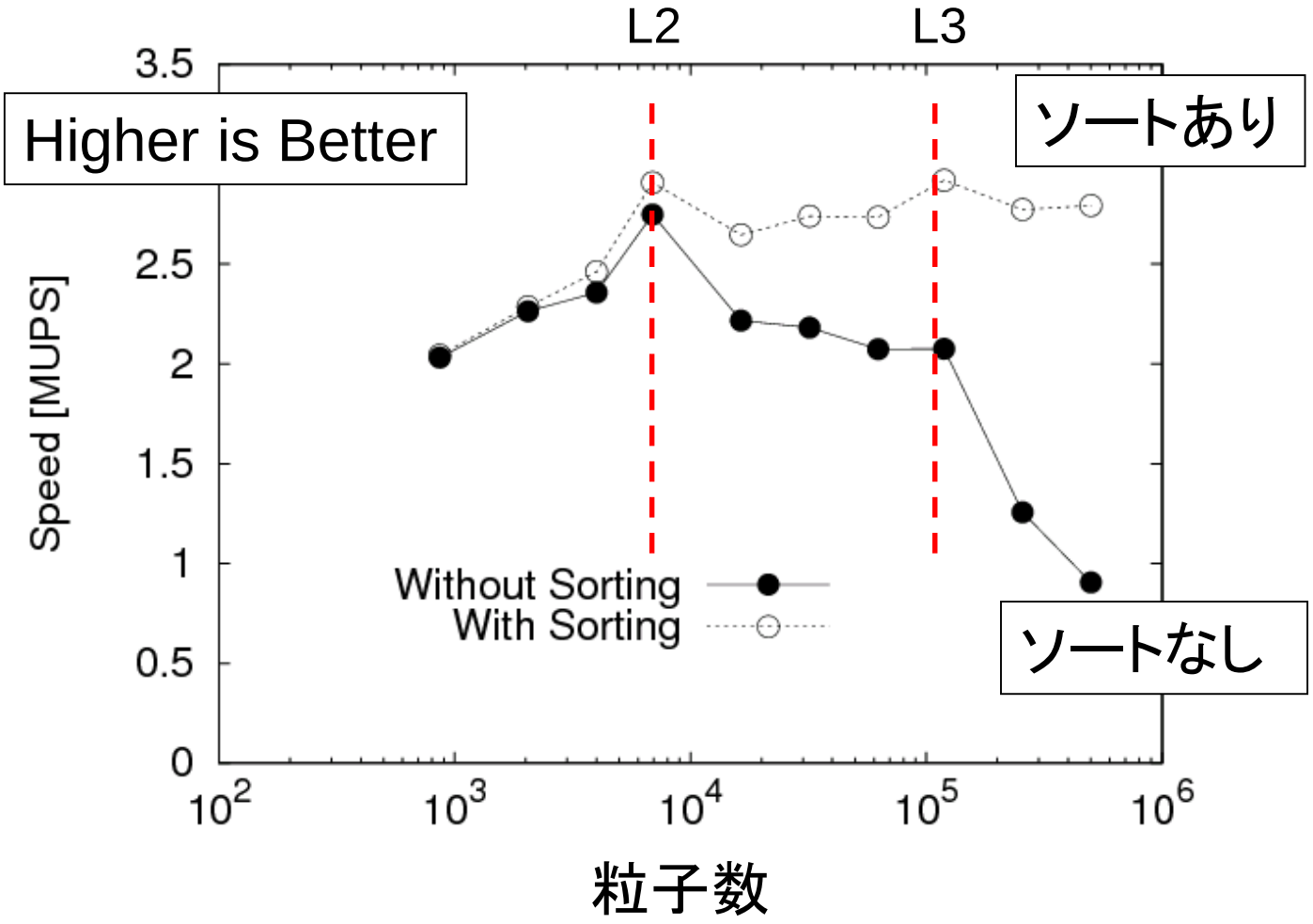


ソートのやりかたはセル情報の一次元実装と同じ



メモリ最適化3 空間ソート (2/2)

空間ソートの効果



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

- 使うデータをなるべくキャッシュ、レジスタにのせる
- 計算量を犠牲にメモリアクセスを減らす
  → ソートが有効であることが多い
- 計算サイズの増加で性能が劣化しない
  → キャッシュを効率的に使えている

- メモリアクセス最適化の効果は一般に大きい
  不適切なメモリ管理をしていると、100倍以上遅くなることも
  → 100倍以上の高速化が可能
  → アーキテクチャにあまり依存しない
  → PCでの最適化がスパコンでも効果を発揮

必要なデータがほぼキャッシュに載っており、CPUの
計算待ちがほとんどになって初めてCPUチューニングへ

CPUチューニング

条件分岐削除
SIMD化

アセンブリ、読んでますか？

ソース

```
const int N = 10000;
void func(double a[N], double b[N]){
  for(int i=0;i<N;i++){
    a[i] = a[i] + b[i];
  }
}
```

```
$ g++ -O2 -S test.cpp
$ c++filt < test.s
```

・「-S」オプションでアセンブリ出力
・c++filtは、変換された名前を元に戻すツール

アセンブリ

```
func(double*, double*):
LFB0:
  xorl  %eax, %eax
  .align 4,0x90
L2:
  movsd (%rdi,%rax), %xmm0
  addsd (%rsi,%rax), %xmm0
  movsd %xmm0, (%rdi,%rax)
  addq  $8, %rax
  cmpq  $80000, %rax
  jne L2
  ret
```

```
i= 0

xmm0 = a[i]
xmm0 += b[i]
a[i] = xmm0
i++
if (i<10000)
goto L2
```

CPUチューニングをするならアセンブリ見るのは必須

条件分岐削除 (1/5)

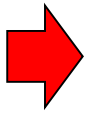
条件分岐削除とは

- if文などの条件により、ジャンプを伴う処理を削除する最適化
- 主にマスク処理を行う

条件分岐ジャンプの例

a[i]が負の時だけ b[i]を加えたい

```
const int N = 100000;
void
func(double a[N], double b[N]){
  for(int i=0;i<N;i++){
    if(a[i] < 0.0) a[i] +=b[i];
  }
}
```



L1:
(1) a[i] と 0を比較
(2) 0より大きければL2へジャンプ
(3) a[i] = a[i] + b[i]
L2:
(4) i = i + 1
(5) iがNより小さければL1へジャンプ

※ 実際のアセンブリとは構造が異なる

なぜ問題となるか？

比較結果がわかるまで、次に実行すべき命令が決まらないから
最近は投機的実行などがサポートされているが、うまくいかない場合もある

条件分岐削除 (2/5)

修正後

```
void
func2(double a[N], double b[N]){
  for(int i=0;i<N;i++){
    double tmp = b[i];
    if(a[i] >= 0.0) tmp = 0.0;
    a[i] += tmp;
  }
}
```

修正前

```
const int N = 100000;
void
func(double a[N], double b[N]){
  for(int i=0;i<N;i++){
    if(a[i] < 0.0) a[i] +=b[i];
  }
}
```

L1:
(1) テンポラリ変数tmpにb[i]を代入
(2) もしa[i]が正ならtmpを 0クリア
(3) 問答無用で a[i]にtmpを加算 ← ループ判定ジャンプ以外のジャンプが消えた
(4) i = i + 1
(5) iがNより小さければL1へジャンプ

実際にそれぞれどんなアセンブリが出るか、
「g++ -O2 –S」で試してみること

条件分岐削除 (3/5)

実際のコード

```
const int pn = particle_number;
 for (int i = 0; i < pn; i++) {
   const int kp = pointer[i];
   const int np = number_of_partners[i];
   const double qix = q[i][X];
   const double qiy = q[i][Y];
   const double qiz = q[i][Z];
   double pix = 0.0;
   double piy = 0.0;
   double piz = 0.0;
   for (int k = 0; k < np; k++) {
     const int j = sorted_list[kp + k];
     double dx = q[j][X] - qix;
     double dy = q[j][Y] - qiy;
     double dz = q[j][Z] - qiz;
     double r2 = (dx * dx + dy * dy + dz * dz);
     if (r2 > CL2) continue;
     double r6 = r2 * r2 * r2;
     double df = ((24.0 * r6 - 48.0) / (r6 * r6 * r2)) * dt;
     pix += df * dx;
     piy += df * dy;
     piz += df * dz;
     p[j][X] -= df * dx;
     p[j][Y] -= df * dy;
     p[j][Z] -= df * dz;
   }
   p[i][X] += pix;
   p[i][Y] += piy;
   p[i][Z] += piz;
 }
```

for i 粒子
  for j 粒子
    i,j粒子の距離を計算
    カットオフ以上ならcontinue
    力を計算
    運動量の書き戻し
  end for
end for

条件分岐削除 (4/5)

修正後のコード

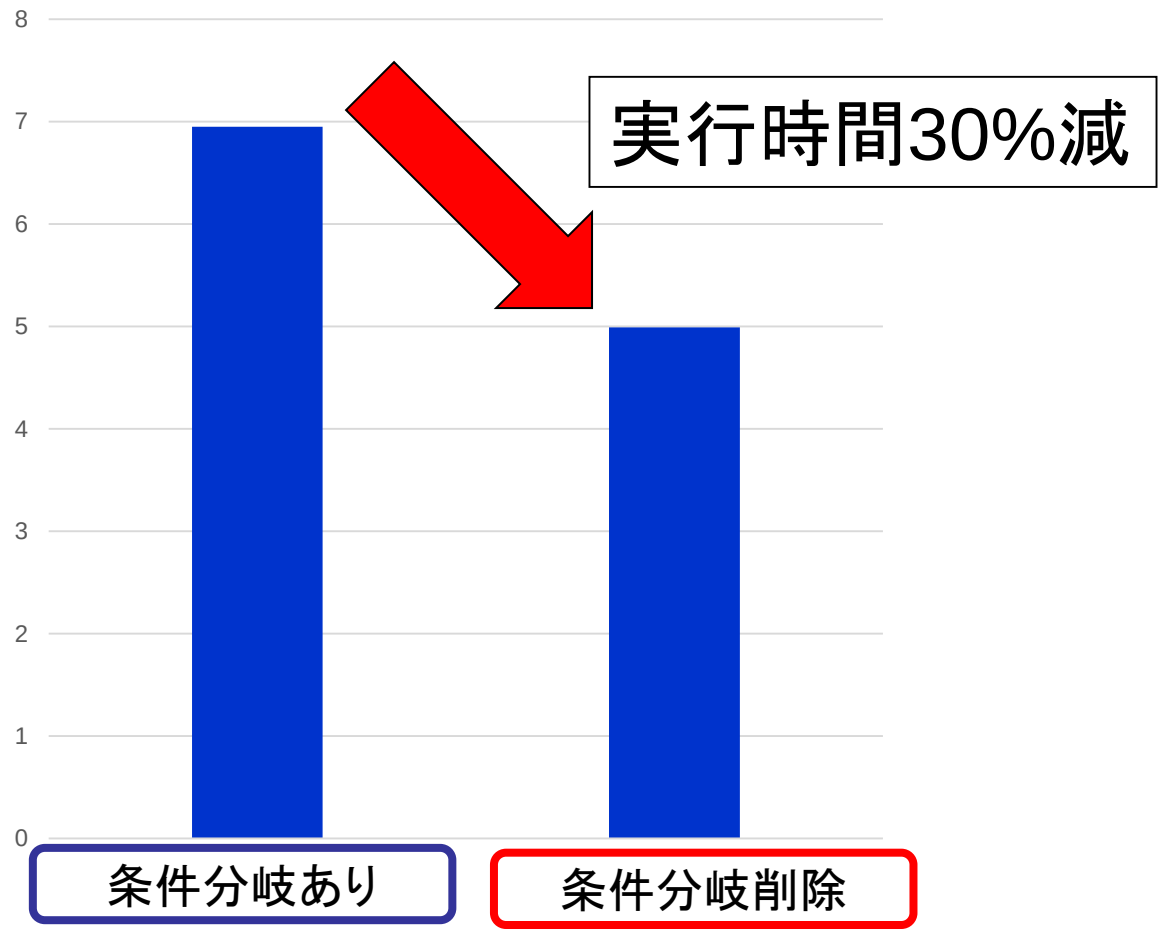
```
const int pn = particle_number;
 for (int i = 0; i < pn; i++) {
   const int kp = pointer[i];
   const int np = number_of_partners[i];
   const double qix = q[i][X];
   const double qiy = q[i][Y];
   const double qiz = q[i][Z];
   double pix = 0.0;
   double piy = 0.0;
   double piz = 0.0;
   for (int k = 0; k < np; k++) {
     const int j = sorted_list[kp + k];
     double dx = q[j][X] - qix;
     double dy = q[j][Y] - qiy;
     double dz = q[j][Z] - qiz;
     double r2 = (dx * dx + dy * dy + dz * dz);
     //if (r2 > CL2) continue;
     double r6 = r2 * r2 * r2;
     double df = ((24.0 * r6 - 48.0) / (r6 * r6 * r2)) * dt;
     if (r2 > CL2) df = 0.0;
     pix += df * dx;
     piy += df * dy;
     piz += df * dz;
     p[j][X] -= df * dx;
     p[j][Y] -= df * dy;
     p[j][Z] -= df * dz;
   }
   p[i][X] += pix;
   p[i][Y] += piy;
   p[i][Z] += piz;
 }
```

for i 粒子
  for j 粒子
    i,j粒子の距離を計算
    力を計算
    カットオフ以上なら力をゼロクリア
    運動量の書き戻し
  end for
end for

条件分岐削除 (5/5)

結果

119164粒子、密度1.0、カットオフ3.0



Xeon E5-2680 v3 @ 2.50GHzでの結果

条件分岐削除のまとめ

- 条件分岐によるジャンプ (breakやcontinue等)を削除する

  条件により実行したりしなかったりするコードブロックがないようにする

- 効果があるかどうかは環境依存

  SPARCやPOWERなどで効果が大きかった
  最近のx86でも早くなることがある
  早くならないこともある (KNLでは効果がなかった)

プログラムを数行書き換えるだけで数倍早くなることもあるので試す価値はある

SIMDとは

SIMDとは

Single Instruction Multiple Data
(シムディー、シムド)

複数のデータに、一種類の演算を
同時に実行する

1		3		3
5	X	2	=	10
3		12		36
2		9		18

計算機の性能

性能 ＝ 動作周波数 × 1サイクルあたりの命令実行数 × 1命令あたりの演算数

スーパースカラ

SIMD

- 動作周波数: リーク電流の影響で2000年頃から上がっていない
- 1サイクルあたりの命令実行数: 依存関係チェックの問題で、もう頭打ち
- 1命令あたりの演算数: あと増やせそうなのここだけ

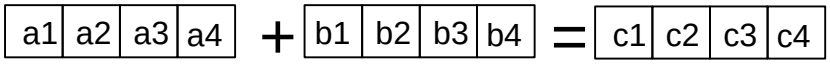
現代アーキテクチャにおいてSIMD対応は必須

SIMDでできること

演算

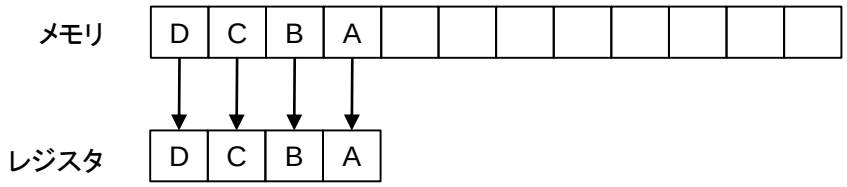
演算はSIMDの幅だけ同時に、かつ独立に実行できる (ベクトル演算)

$$\vec{a} + \vec{b} = \vec{c}$$

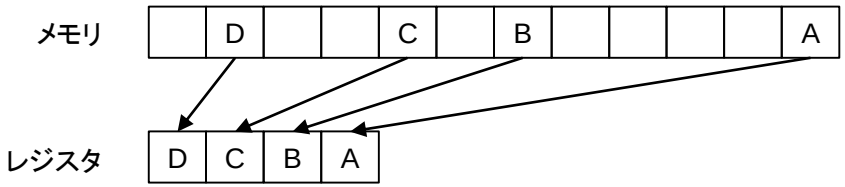


データのロード/ストア

一度にもってくると早い

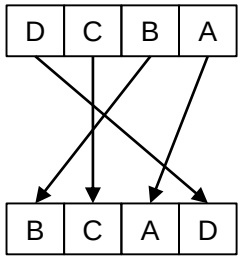


バラバラに持ってくると遅い

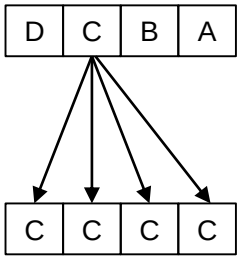


シャッフル、マスク処理

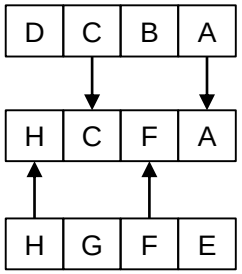
並び替え



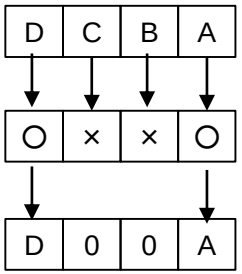
値のコピー



混合



マスク処理



SIMDの使い方

SIMDは使うCPUの命令セットに強く依存する
→ SIMDは原則としてアセンブラで書く
(実際には、アセンブラに対応したC言語の関数を使う)

実際のコード例

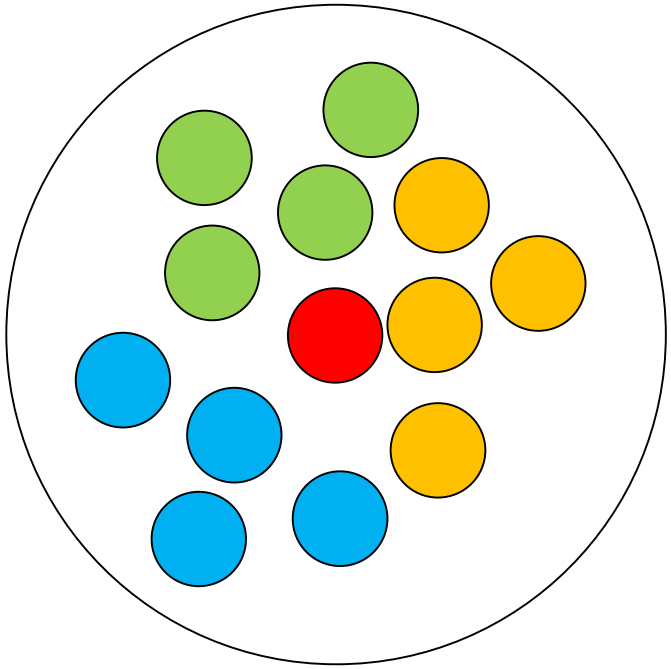
```
v4df vqj_a = _mm256_load_pd((double*)(q + j_a));
v4df vdq_a = (vqj_a - vqi);
v4df vd1_a = vdq_a * vdq_a;
v4df vd2_a = _mm256_permute4x64_pd(vd1_a, 201);
v4df vd3_a = _mm256_permute4x64_pd(vd1_a, 210);
v4df vr2_a = vd1_a + vd2_a + vd3_a;
```

ロード/ストア/シャッフル系は関数呼び出し
四則演算や代入は普通に書ける

ナイーブなSIMD化 (1/2)

ループアンロール

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



ナイーブな実装

4つの座標データをレジスタにロード

$\hat{q}_x^i$ | q_x^{i+3} | q_x^{i+2} | q_x^{i+1} | q_x^i |

$\hat{q}_x^j$ | q_x^{j+3} | q_x^{j+2} | q_x^{j+1} | q_x^j |

※ y, zも同様

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

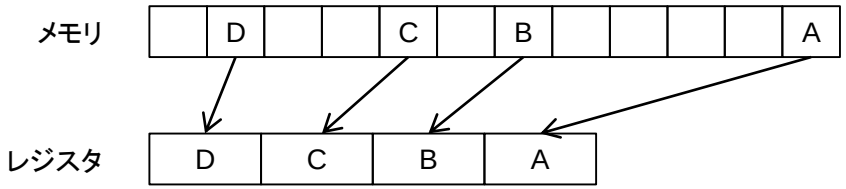
$$\hat{r^2} = \hat{dx}^2 + \hat{dy}^2 + \hat{dz}^2$$

距離が4つパックされた

ナイーブなSIMD化 (2/2)

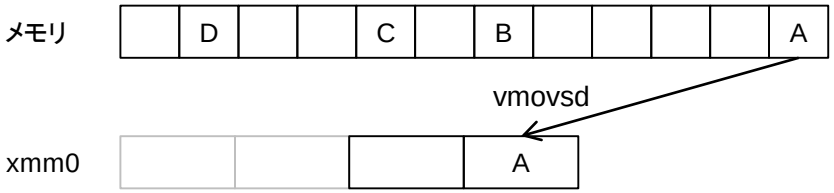
問題点

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ロード

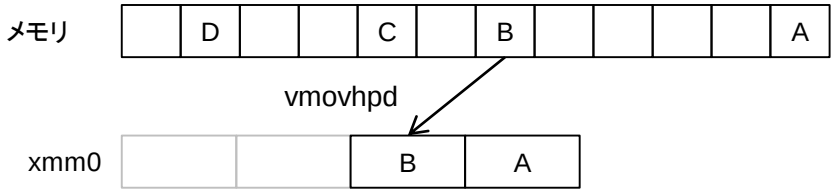


実際に起きること

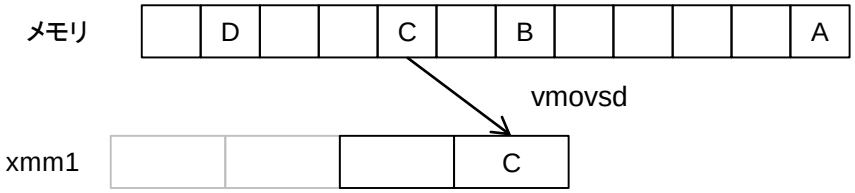
(1) Aをxmm0下位にロード



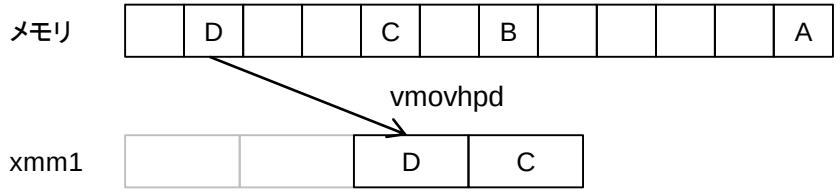
(2) Bをxmm0上位にロード



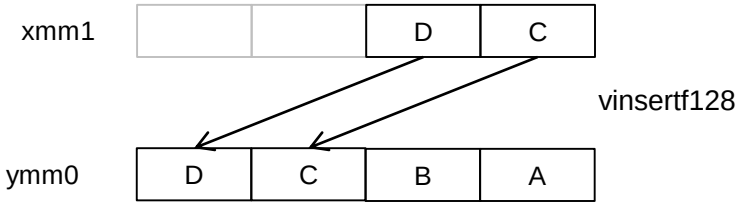
(3) Cをxmm1下位にロード



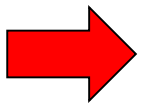
(4) Dをxmm1上位にロード



(5) xmm1全体をymm0上位128bitにコピー



※ これをx,y,z座標それぞれでやる
※ データの書き戻しも同様



無茶苦茶遅い

AVX2命令を用いたSIMD化 (1/4)

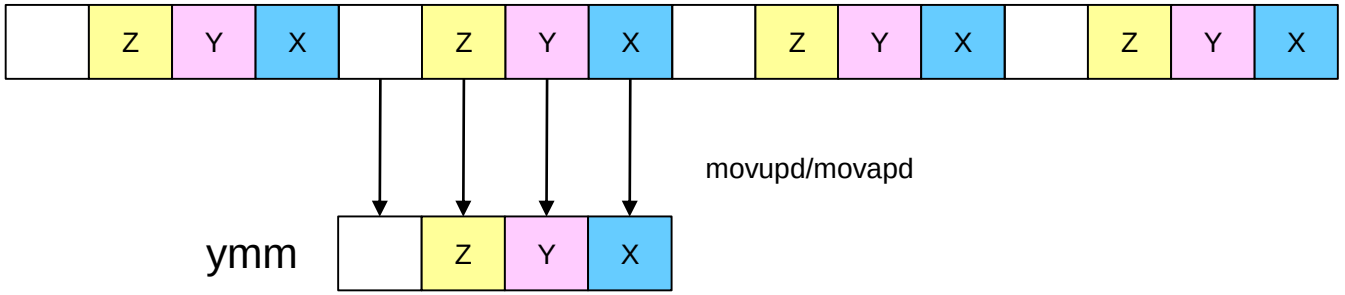
パディング付きAoS

※Array of Structure

データを粒子数*4成分の二次元配列で宣言
double q[N][4], p[N][4];

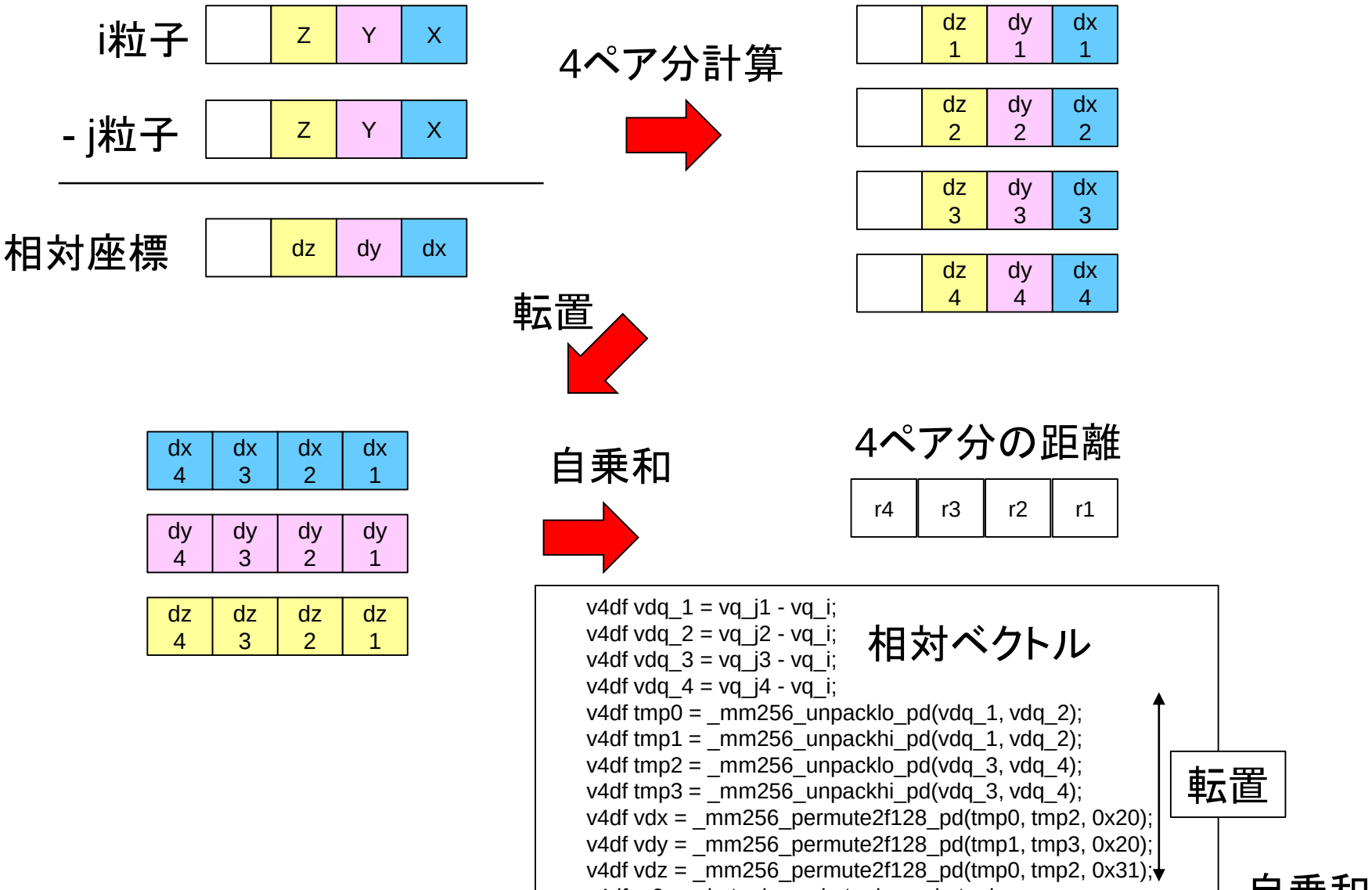
うれしいこと

(x,y,z)の3成分のデータを256bitレジスタに一発ロード
ただし、1成分(64bit) は無駄になる



AVX2命令を用いたSIMD化 (2/4)

データの転置



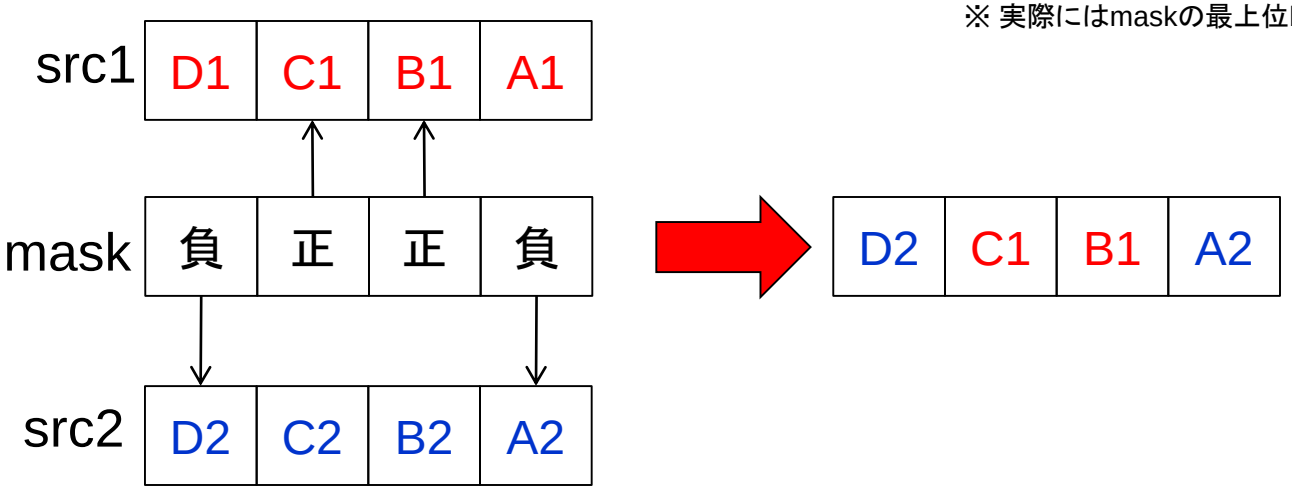
AVX2命令を用いたSIMD化 (3/4)

マスク処理

Bookkeeping法により、相互作用範囲外のペアもいる→マスク処理

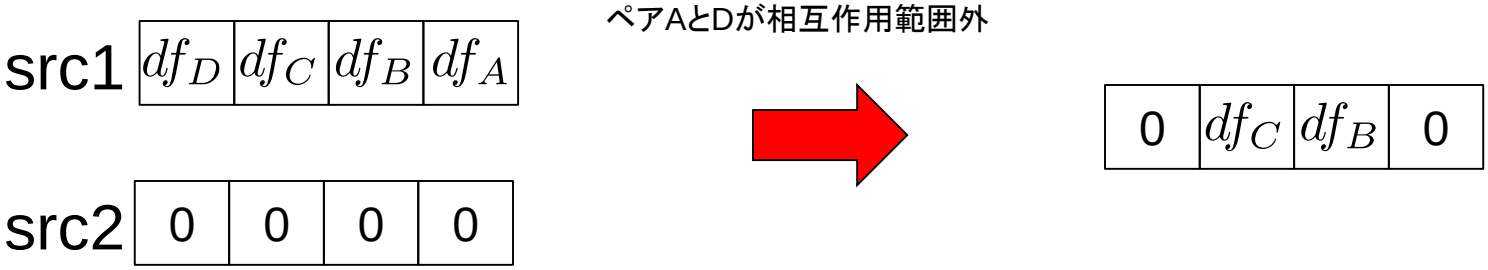
vblendvpd: マスクの要素の正負により要素を選ぶ

※ 実際にはmaskの最上位bitが0か1かで判定している



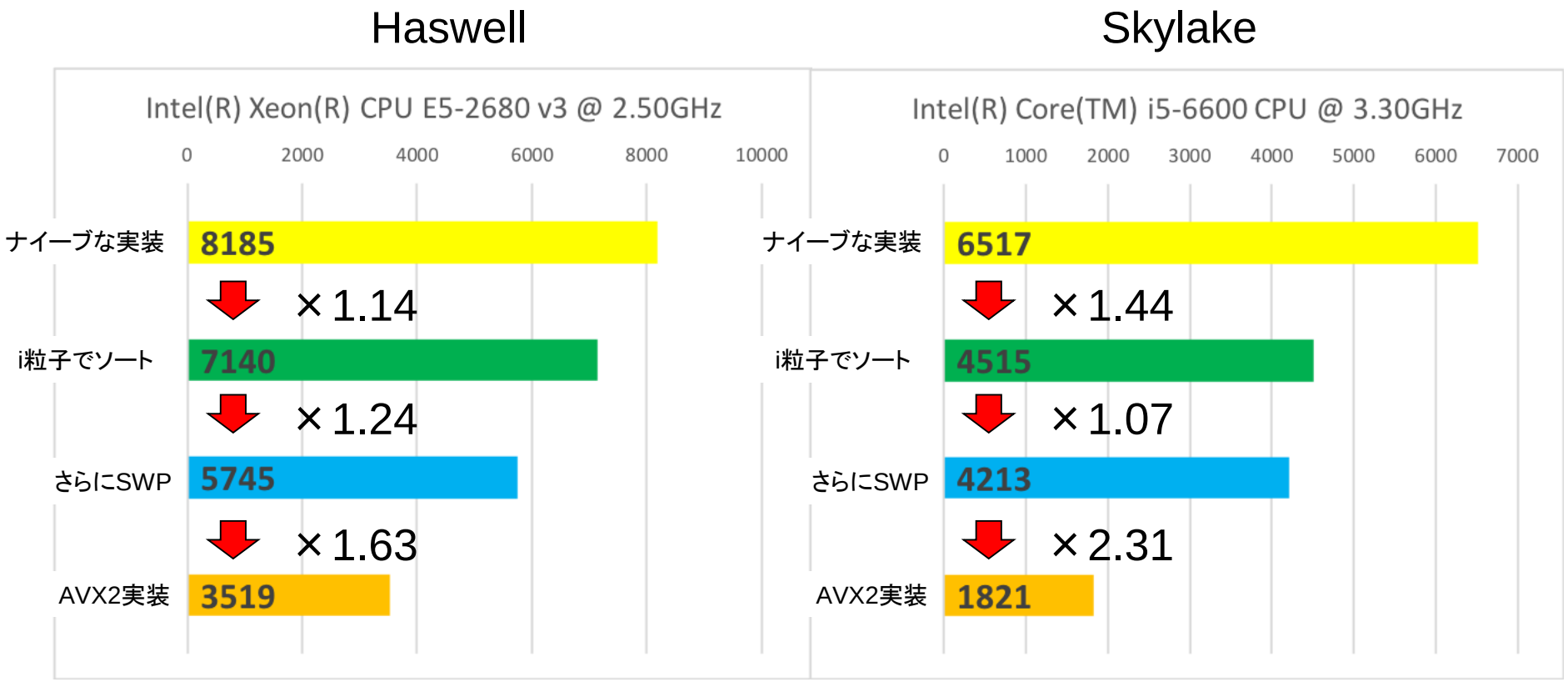
相互作用距離とカットオフ距離でマスクを作成し、力をゼロクリア

ペアAとDが相互作用範囲外



AVX2命令を用いたSIMD化 (4/4)

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化無しの最速実装に比べてHWで1.6倍、SLで2.3倍高速化
- AVX2を用いた加速度合いはSkylakeの方が高い

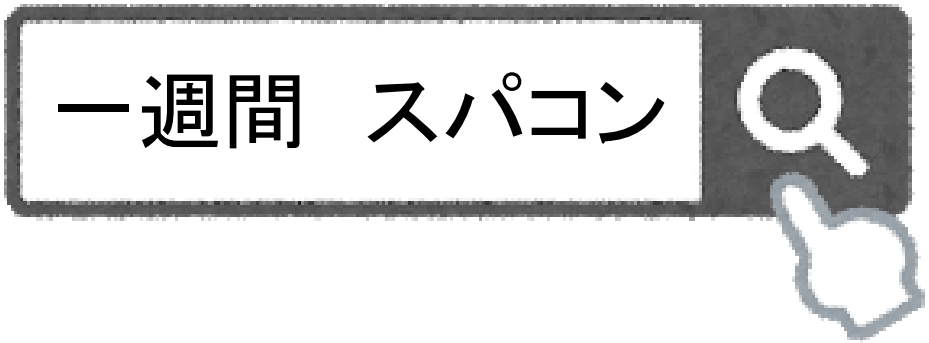
SIMD化のまとめ

- 明示的なSIMD化により、数倍以上高速化する可能性がある
- データ構造を工夫することで、コンパイラが自動でSIMD化することがある
- 効果的なSIMD化はデータ構造の変更を伴う

スパコンに興味を持った方は…

「一週間でなれる！スパコンプログラマ」

https://kaityo256.github.io/sevendayshpc/



全体のまとめ

高速化、並列化は好きな人がやればよい

なぜなら科学とは好きなことをするものだから