第9回
高速化チューニングとその関連技術2

渡辺宙志

東京大学物性研究所

Outline

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

Jun. 15, 2017@計算科学技術特論A

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

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

高速化とは何か？

アルゴリズムを変えず、実装方法によって実行時間を短くする方法
→ アルゴリズムレベルの最適化は終了していることを想定
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から高速化

実装レベルの高速化とは何か？

「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

高速化、その前に

やみくもに高速化をはじめるのはダメ、ゼッタイ
まず、どこが遅いか、どういう原因で遅いかを確認してから
→プロファイラによる確認

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
(分子動力学計算では力の計算)

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

まず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

プロファイリング、その前に (1/2)

そもそも僕のコード、早いの遅いの？

ざっくりで良いので、どのくらいの
性能が出るべきか考えましょう

計算機の「メモリ転送性能」と、計算の「要求メモリ転送性能」を比較する

計算機の性能

B/F (Byte-per-Flops)値
- バンド幅を演算性能で割った値
- 高いほどメモリ転送性能が良い

計算の性質

演算強度 (Flops/Byte)
- 1バイト転送あたり何回計算するか
  - 高いと計算能力依存
  - 低いとメモリ転送能力依存

ほとんどの場合メモリ転送能力がボトルネック

プロファイリング、その前に (2/2)

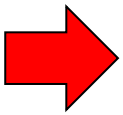
簡単な例:

```
for(int i=0;i<N;i++){
  a[i] = a[i] + b[i];
}
```

配列a, bは倍精度浮動小数(double)

1反復あたり
- 読み込み2個 (16バイト)
- 書き込み1個 (8バイト)
- 演算1回

なので演算強度 1/24 [Flops/Byte]



これが理論ピーク演算性能を出すためには、B/Fが24必要

現在の計算機はB/F～0.5程度なので、この計算はメモリバンド幅がボトルネック

- メモリバンド幅ボトルネックなのに、計算方法をいじっても無駄
- メモリバンド幅ボトルネックではないにも関わらず性能が出ていなければ次のステップ(プロファイリング)に進む

キャッシュとかレイテンシとかを考えると話がややこしくなる
真面目にやりたい人は「ルーフラインモデル」などで検索

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

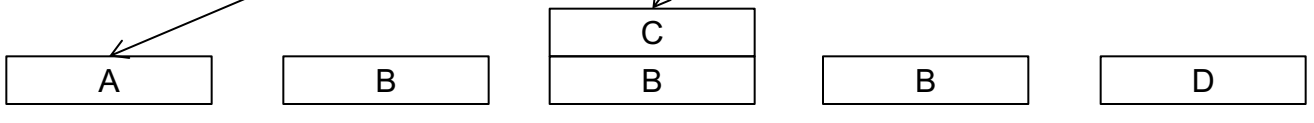
どんな命令が発効され、どこでCPUが待っているかを調べる
範囲単位で調査 (プログラム全体、ユーザ指定)
高速化の指針を探るのに利用

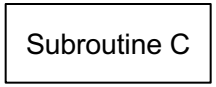
サンプリング

実行中、一定間隔で「いまどこを実行中か」を調べ、実行時間は
その出現回数に比例すると仮定する

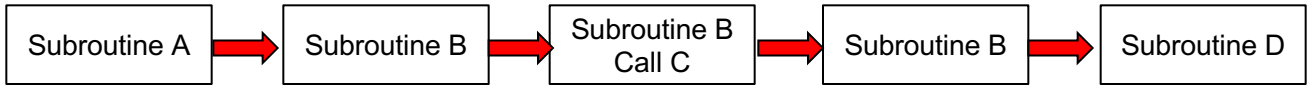
定期的(例えば1ミリ秒に一度)に
いま何が実行中かをカウント

呼び出しスタック





実行中のルーチン



gprofの使い方

gprofとは

広く使われるプロファイラ。perfが使えるならperfを使った方が良い。
(Macでは使えない。iprofilerとinstrumentsで代用できるが面倒)

使い方

```
$ gcc -pg test.cc
$ ./a.out
$ ls
a.out gmon.out test.cc
$ gprof a.out gmon.out
```

出力

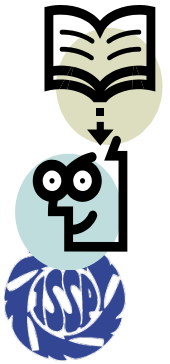
とりあえずEach % timeだけ見ればいいです

Flat profile:

Each sample counts as 0.01 seconds.

サンプリングレートも少しだけ気にすること

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.57	0.93	0.93	1	925.26	925.26	matmat()
0.00	0.93	0.00	1	0.00	0.00	global constructors keyed to A
0.00	0.93	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)
0.00	0.93	0.00	1	0.00	0.00	init()
0.00	0.93	0.00	1	0.00	0.00	matvec()
0.00	0.93	0.00	1	0.00	0.00	vecvec()



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

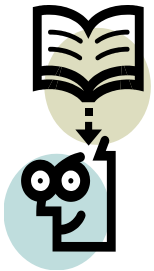
結果の解釈 (サンプリング)

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)

Hardware Counter

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

取得可能な主なイベント:
- 実行命令 (MIPS)
- 浮動小数点演算 (FLOPS)　←こういうのに気を取られがちだが
- サイクルあたりの実行命令数 (IPC)
- 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

プロファイラの使い方

システム依存
Linux では perfを使うのが便利
京では、カウントするイベントの種類を指定
カウンタにより取れるイベントが決まっている
→同じカウンタに割り当てられたイベントが知りたい場合、複数回実行する必要がある

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

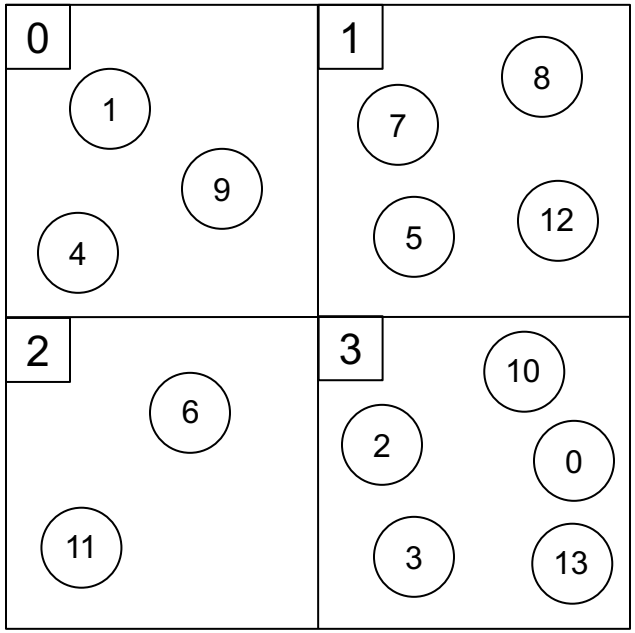
メモリ最適化1 セル情報の一次元実装 (1/2)

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

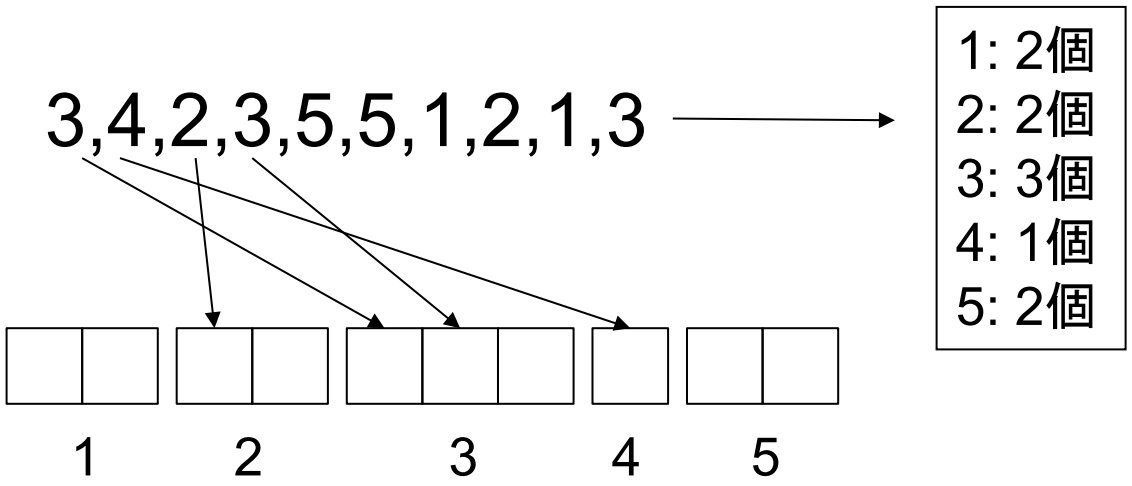
分布数えソート (Counting Sort)

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)

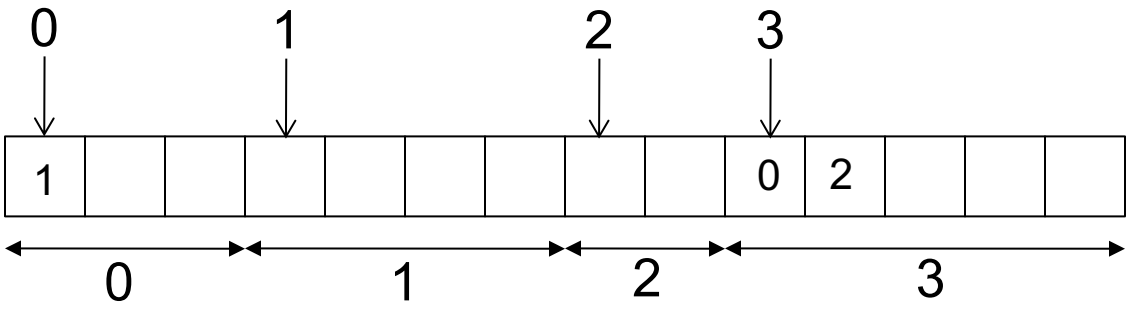


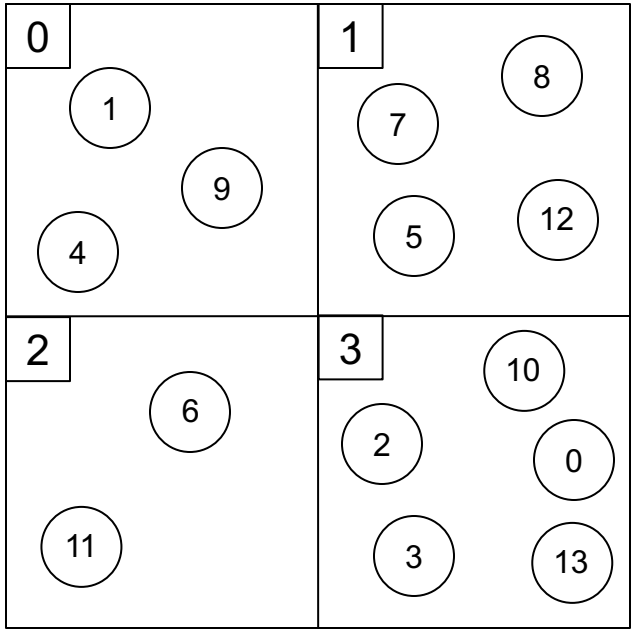
最適化のみならず、知っていると便利なソートアルゴリズムの一種

メモリ最適化1 セル情報の一次元実装 (2/2)

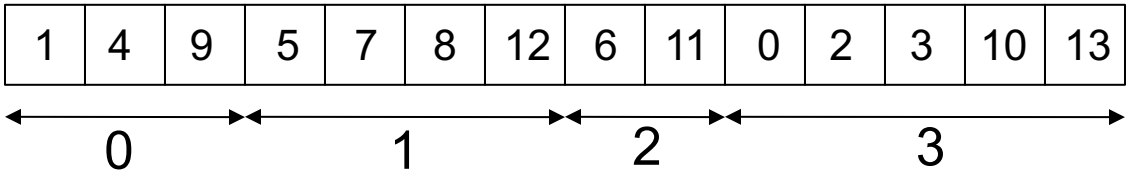
一次元実装

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)



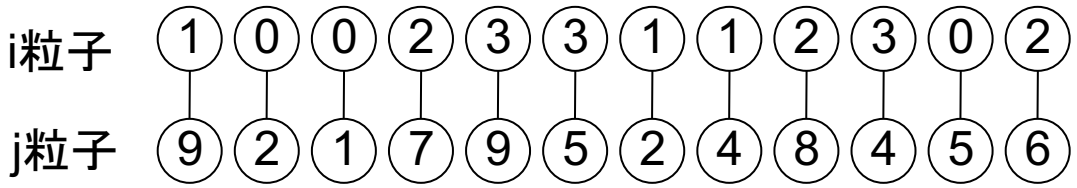
メモリを密に使っている (キャッシュ効率の向上)

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

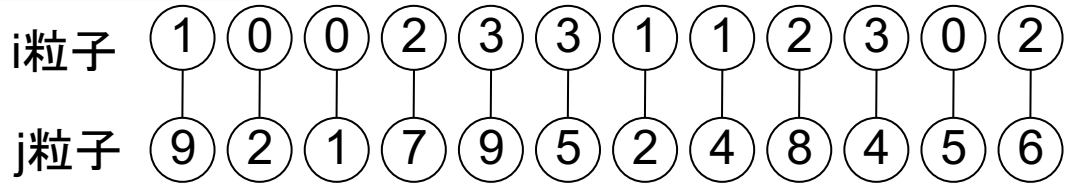
このまま計算すると

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)

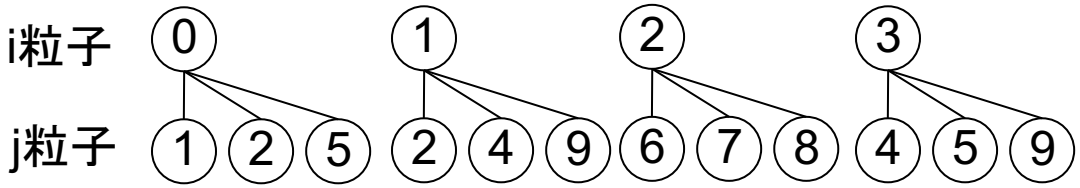
メモリ最適化2 相互作用ペアソート (2/2)

相互作用相手でソート

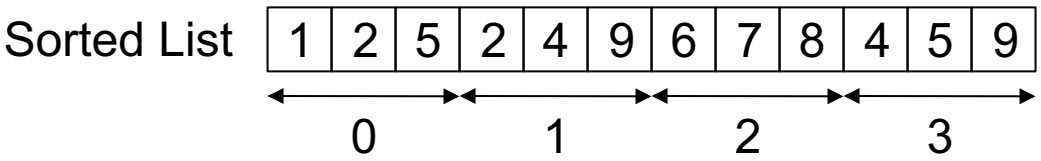
相互作用ペア



i粒子でソート



配列表現



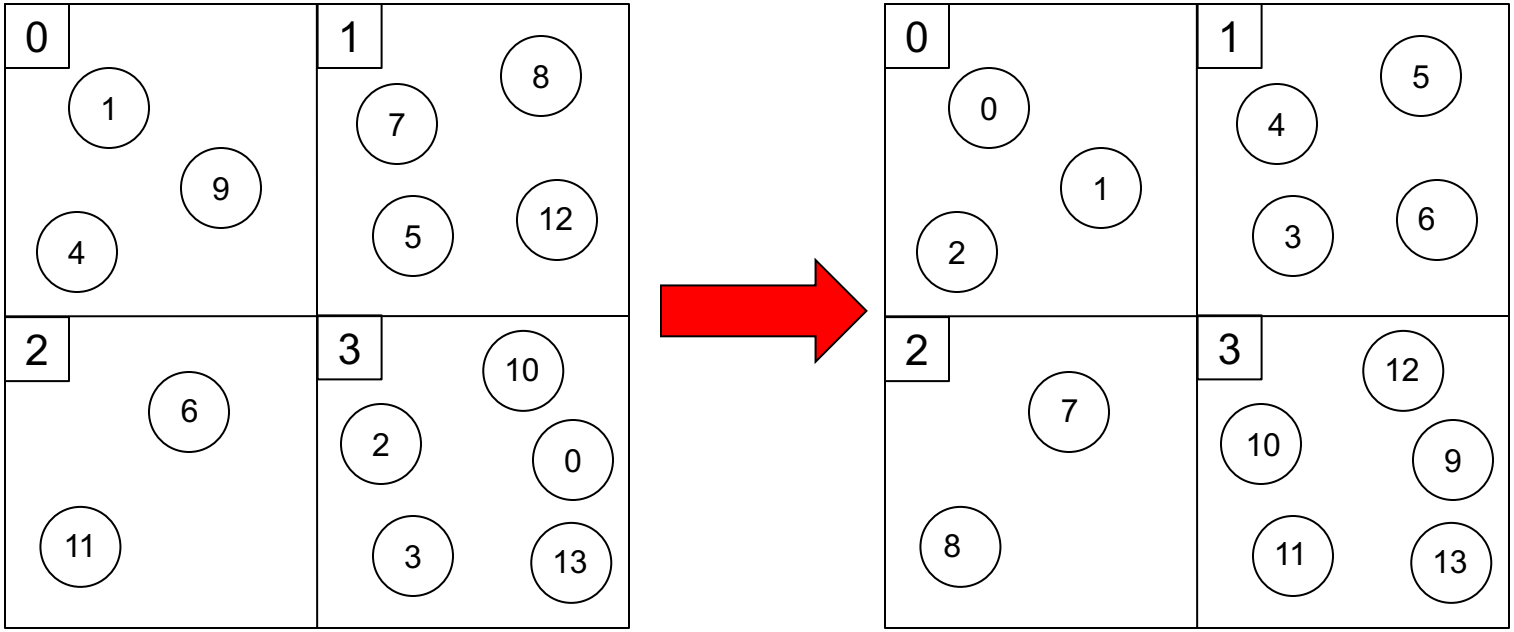
i粒子をキーとした分布数えソート

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

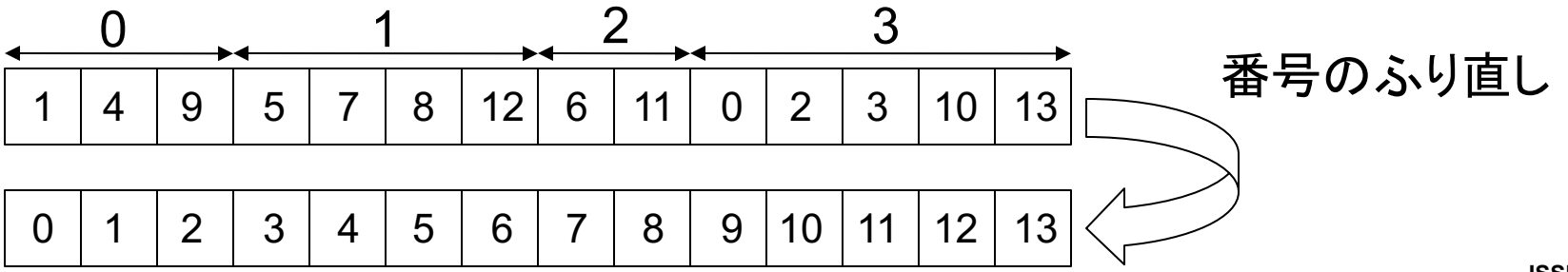
メモリ最適化3 空間ソート (1/2)

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

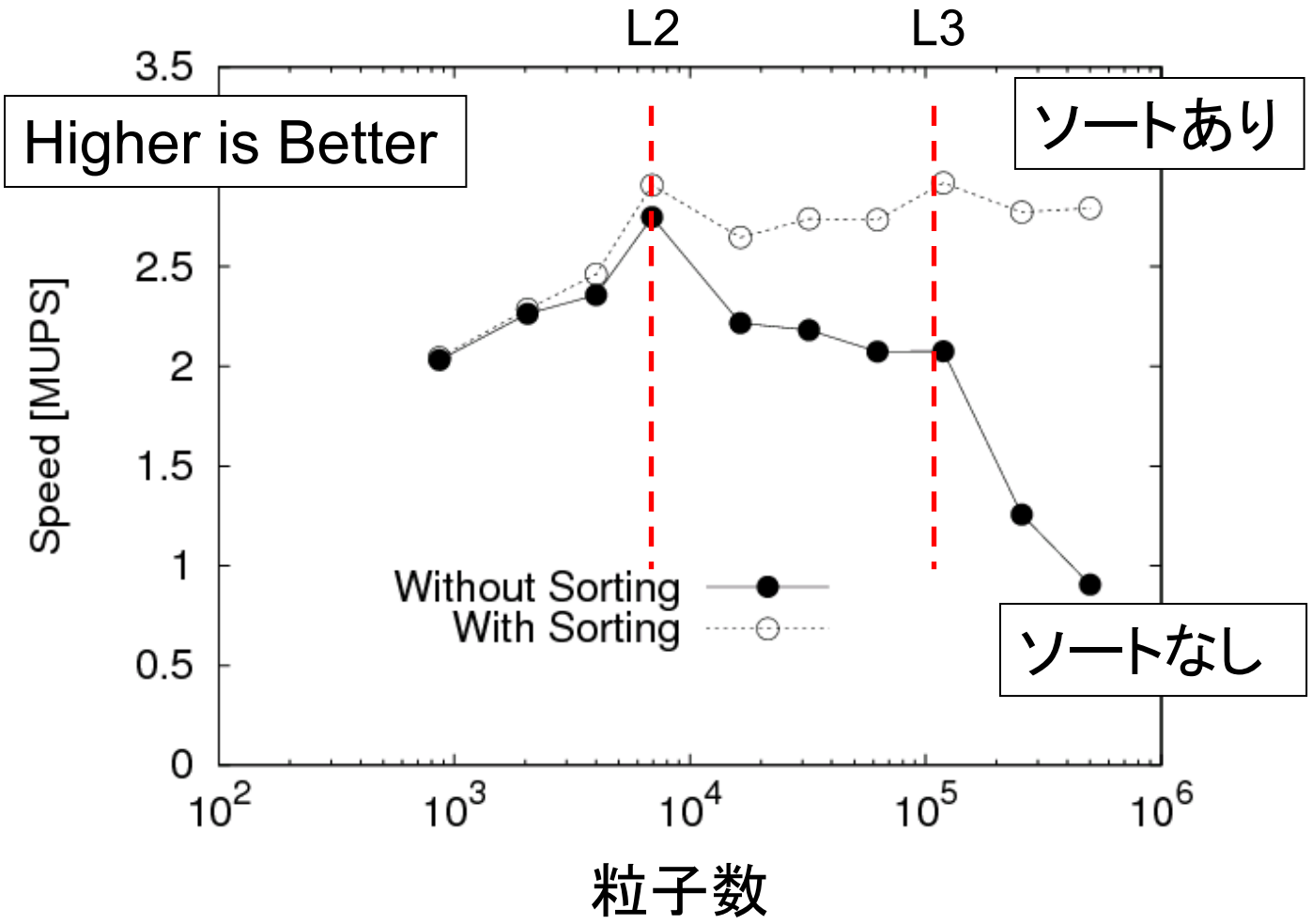


ソートのやりかたはセル情報の一次元実装と同じ



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

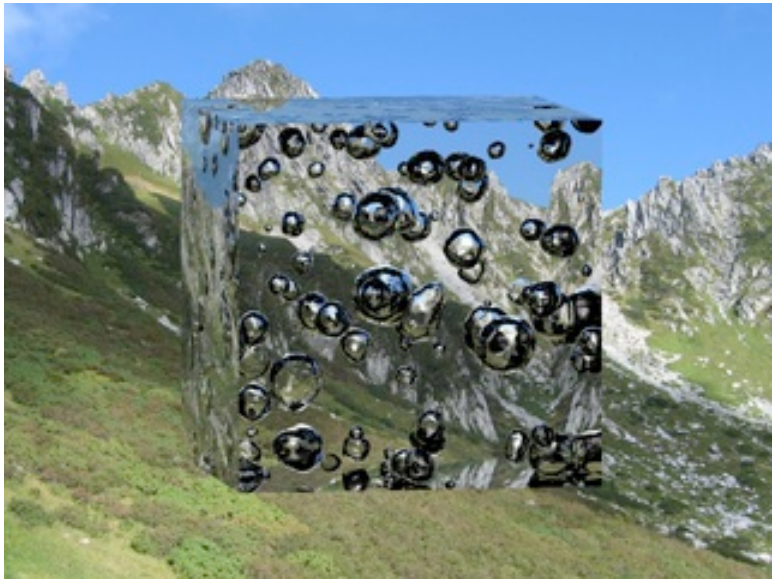
必要なデータがほぼキャッシュに載っており、CPUの
計算待ちがほとんどになって初めてCPUチューニングへ

閑話休題

研究成果のアウトリーチについて

研究成果のアウトリーチ (1/5)

「京」4096ノード、10億粒子規模の計算を行い、気泡生成のダイナミクスを解明。
以下の論文として出版
H. Watanabe, et al. J. Chem. Phys. **141** 234703 (2014)

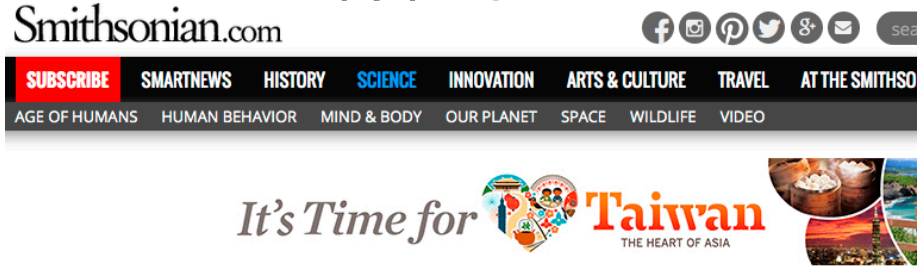


論文誌「J. Chem. Phys.」の出版元であるAIPよりプレスリリース
リリース直前にディレクターから「タイトル盛ったよ」

「シャンパンの泡が世界のエネルギー危機を救う？」
How the Physics of Champagne and Soda Bubbles May Help Address the World's Future Energy Needs

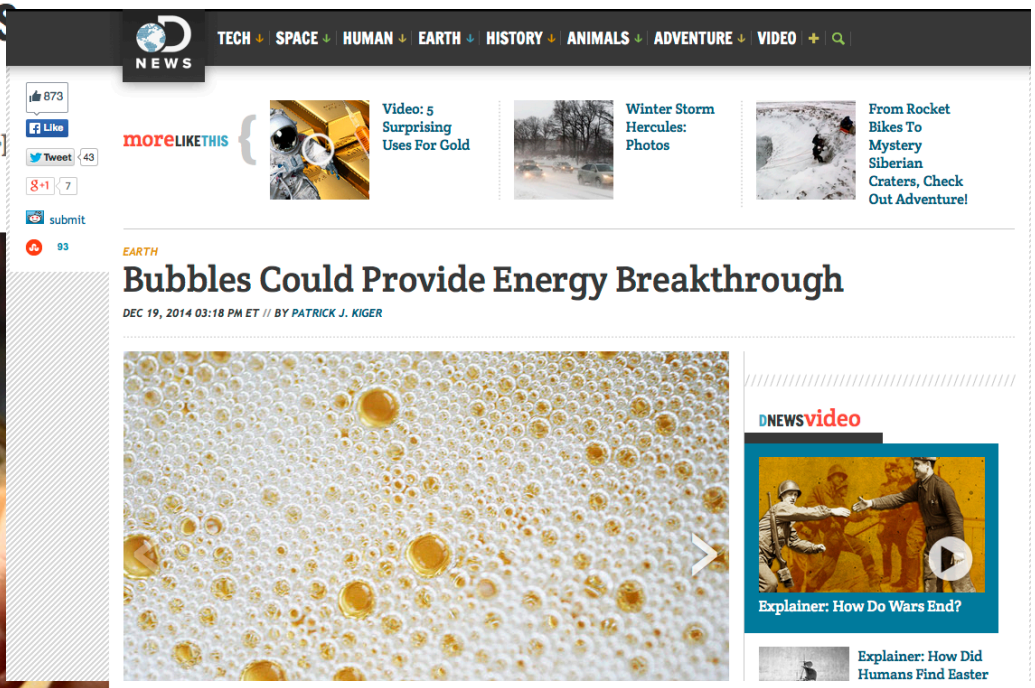
研究成果のアウトリーチ (2/5)

スミソニアン博物館のニュースサイト (メールによる取材)



シャンパンの泡がエネルギーの未来を拓く

ディスカバリーチャネルのニュースサイト



泡がエネルギー問題を解決する？

研究成果のアウトリーチ (3/5)

シャンパン、スパークリングワインの通販サイト

GLASS of BUBBLY

Champagne ▾ Prosecco Cava Cremant English Sparkling Other Sparkling Wines ▾

Could Champagne bubbles help address the world's energy needs?

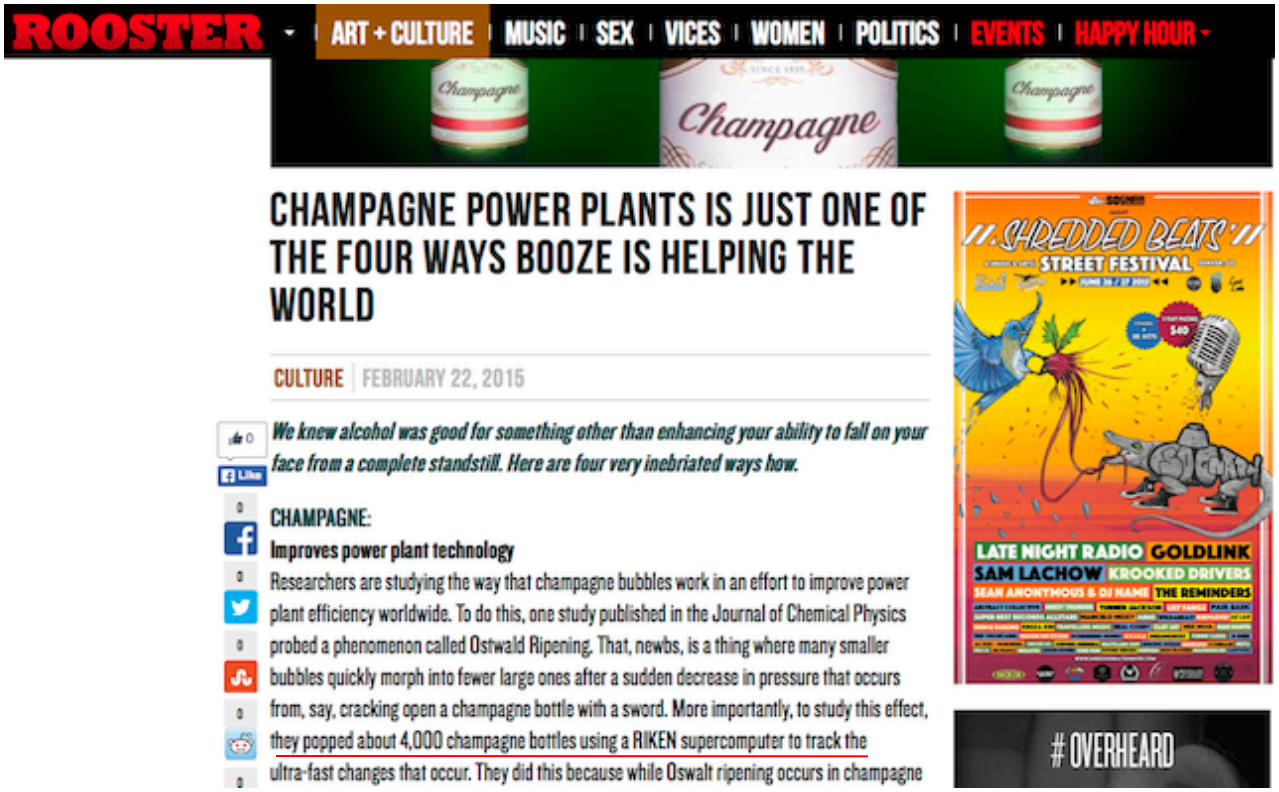
We knew it all along, Champagne is always the answer!

Researchers in Tokyo may have discovered how Champagne bubbles could address the world's future energy needs.

Using Japan's most powerful computer, they have explored the physcics of Champagne bubbles to help find better alternatives for efficient energy.

いつだってシャンパンが答えだってことはわかってたことさ！

研究成果のアウトリーチ (4/5)



ROOSTER | ART + CULTURE | MUSIC | SEX | VICES | WOMEN | POLITICS | EVENTS | HAPPY HOUR

Champagne

CHAMPAGNE POWER PLANTS IS JUST ONE OF THE FOUR WAYS BOOZE IS HELPING THE WORLD

CULTURE | FEBRUARY 22, 2015

We knew alcohol was good for something other than enhancing your ability to fall on your face from a complete standstill. Here are four very inebriated ways how.

CHAMPAGNE:

Improves power plant technology

Researchers are studying the way that champagne bubbles work in an effort to improve power plant efficiency worldwide. To do this, one study published in the Journal of Chemical Physics probed a phenomenon called Ostwald Ripening. That, newbs, is a thing where many smaller bubbles quickly morph into fewer large ones after a sudden decrease in pressure that occurs from, say, cracking open a champagne bottle with a sword. More importantly, to study this effect, they popped about 4,000 champagne bottles using a RIKEN supercomputer to track the ultra-fast changes that occur. They did this because while Oswalt ripening occurs in champagne

「・・・研究者達は、この研究のために理研のスパコンを使って4000本ものシャンパンボトルを空けたそうだ。
次にやるときは俺たちも呼んでくれよ、な？」

研究のアウトリーチってなんだろう？

研究成果のアウトリーチ (5/5)

日本でも広報につとめたが・・・

ISSP Website

RIKEN AICS Website

理研チャンネル

視聴回数 873 回

ほとんど反響無し・・・

研究成果のアウトリーチのまとめ

タイトル重要!!!

どこまで盛るの？
どこまで正確に表現するの？

わかりやすさと正確さのバランスをどうとるか・・・

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)

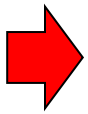
条件分岐削除とは

- 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
(4) i = i + 1
(5) iがNより小さければL1へジャンプ

ループ判定ジャンプ以外のジャンプが消えた

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

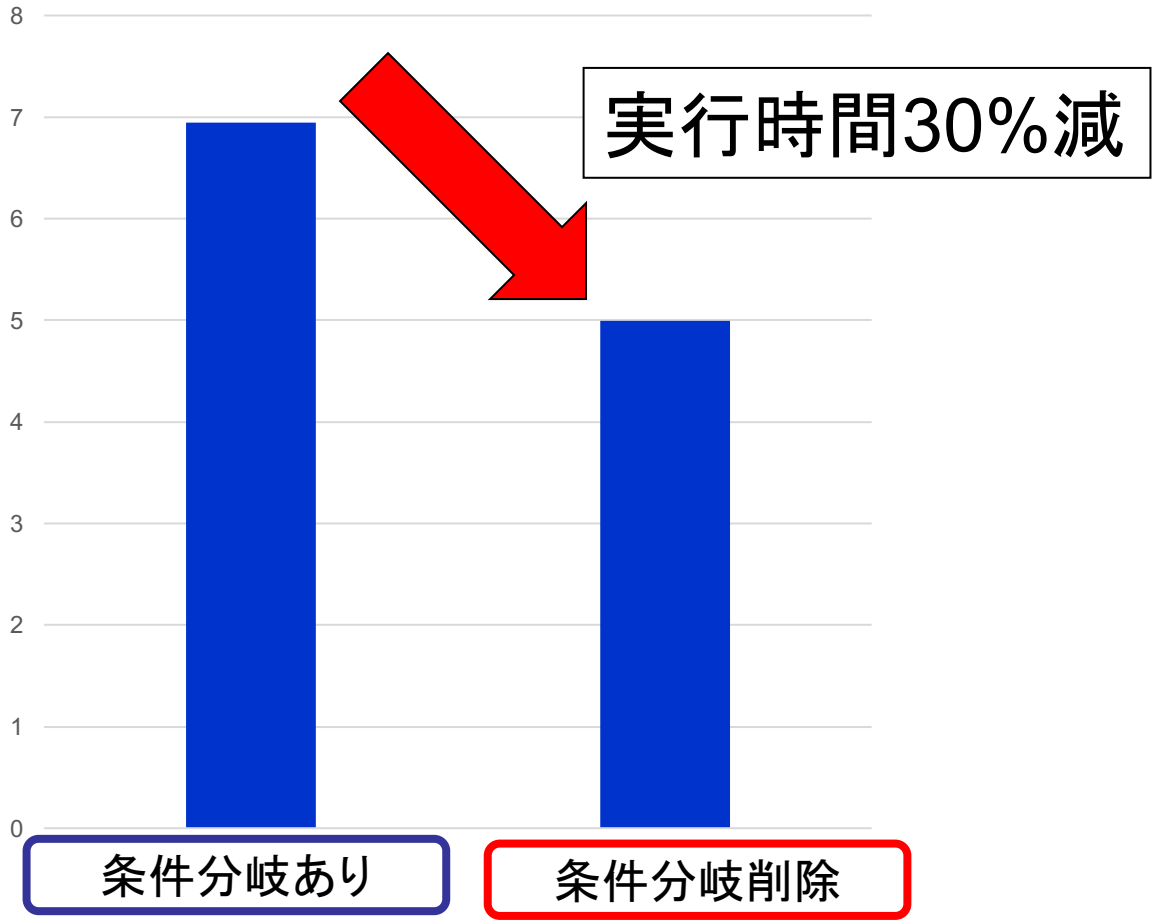
```
const int pn = particle_number;
 for (int i = 0; i < pn; i++) {
   const int kp = pointer[i];
   const int np = number_of_partners[i];
   const double qix = q[i][X];
   const double qiy = q[i][Y];
   const double qiz = q[i][Z];
   double pix = 0.0;
   double piy = 0.0;
   double piz = 0.0;
   for (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

スーパースカラ　　SIMD

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

- 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

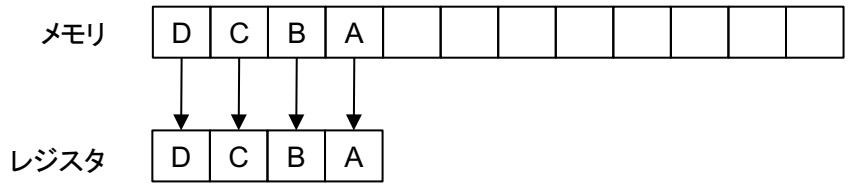
演算はSIMDの幅だけ同時に、かつ独立に実行できる (ベクトル演算)

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

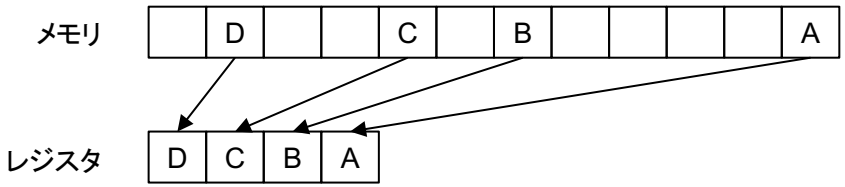
a1	a2	a3	a4	+	b1	b2	b3	b4	=	c1	c2	c3	c4

データのロード/ストア

一度にもってくると早い

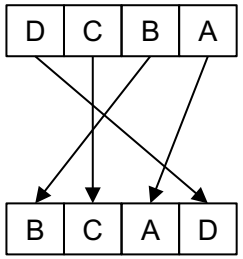


バラバラに持ってくると遅い

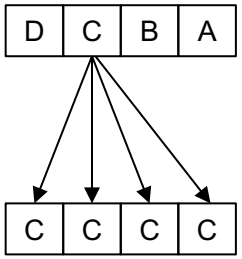


シャッフル、マスク処理

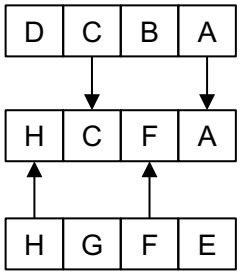
並び替え



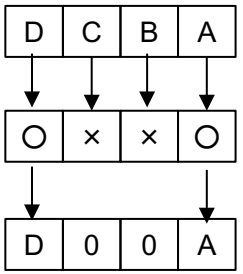
値のコピー



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

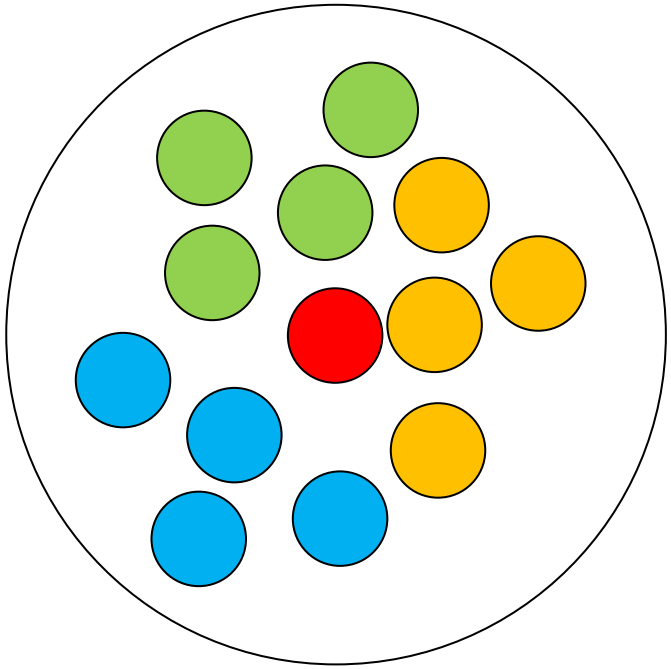
```
v4df vqj_a = _mm256_load_pd((double*)(q + j_a));
v4df vdq_a = (vqj_a - vqi);
v4df vd1_a = vdq_a * vdq_a;
v4df vd2_a = _mm256_permute4x64_pd(vd1_a, 201);
v4df vd3_a = _mm256_permute4x64_pd(vd1_a, 210);
v4df vr2_a = vd1_a + vd2_a + vd3_a;
```

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

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

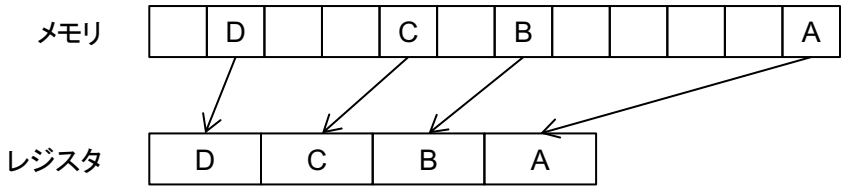
$$\hat{r^2} = \hat{dx}^2 + \hat{dy}^2 + \hat{dz}^2$$

距離が4つパックされた

ナイーブなSIMD化 (2/2)

問題点

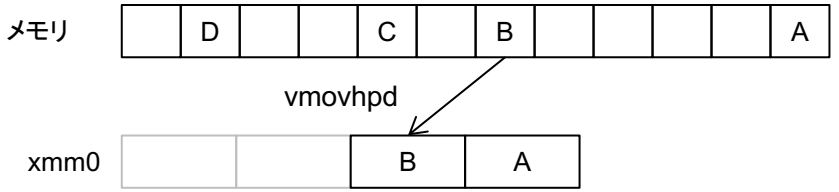
相互作用粒子のインデックスは連続ではない
→ 4つのデータをバラバラにロード



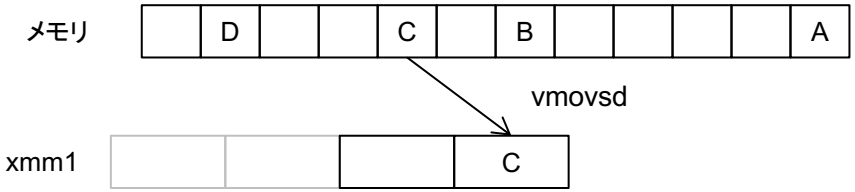
実際に起きること

(1) Aをxmm0下位にロード

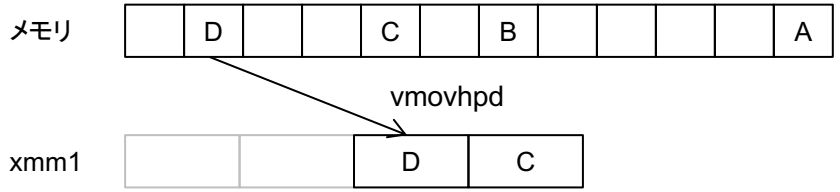
(2) Bをxmm0上位にロード



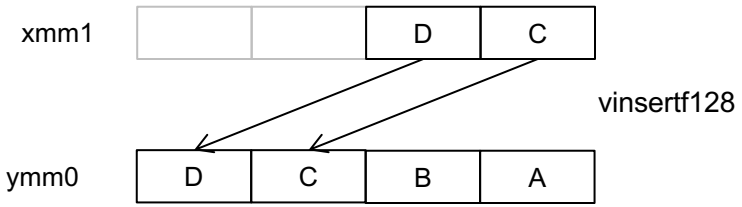
(3) Cをxmm1下位にロード



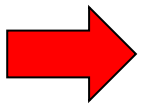
(4) Dをxmm1上位にロード



(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

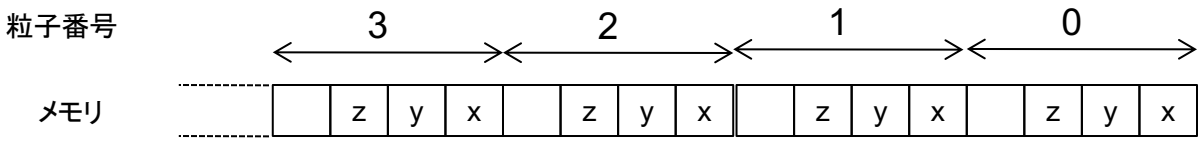


無茶苦茶遅い

AVX2命令を用いたSIMD化 (1/4)

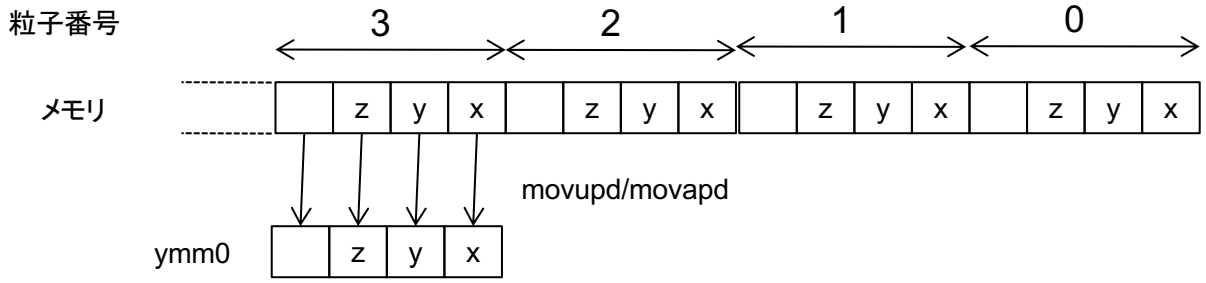
やったこと

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

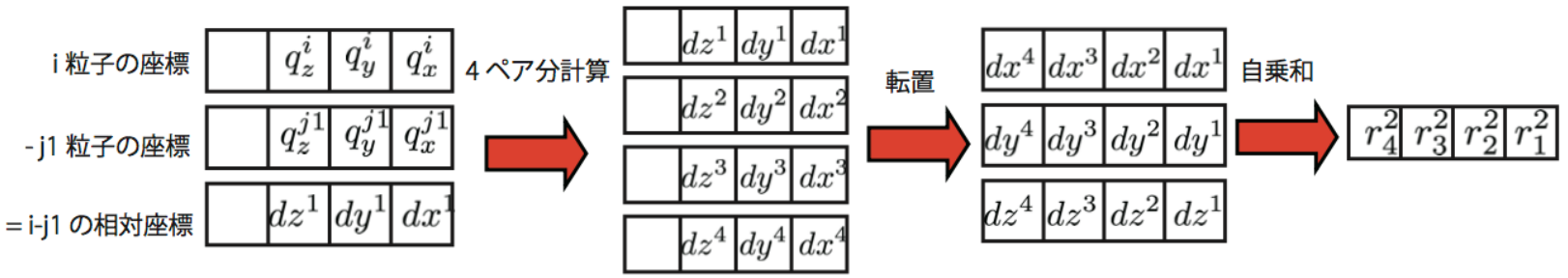


AVX2命令を用いたSIMD化 (2/4)

レジスタへのロードの後、データの転置が必要になる

$$df = \frac{(48 - 24 \times r^6) \times dt}{r^{14}}$$

←Lennard-Jones 粒子の力として計算したい量
（相対距離の自乗の関数）




```
v4df vdq_1 = vq_j1 - vq_i;
v4df vdq_2 = vq_j2 - vq_i;
v4df vdq_3 = vq_j3 - vq_i;
v4df vdq_4 = vq_j4 - vq_i;
v4df tmp0 = _mm256_unpacklo_pd(vdq_1, vdq_2);
v4df tmp1 = _mm256_unpackhi_pd(vdq_1, vdq_2);
v4df tmp2 = _mm256_unpacklo_pd(vdq_3, vdq_4);
v4df tmp3 = _mm256_unpackhi_pd(vdq_3, vdq_4);

v4df vdx = _mm256_permute2f128_pd(tmp0, tmp2, 0x20);
v4df vdy = _mm256_permute2f128_pd(tmp1, tmp3, 0x20);
v4df vdz = _mm256_permute2f128_pd(tmp0, tmp2, 0x31);
v4df vr2 = vdx * vdx + vdy * vdy + vdz * vdz;
```

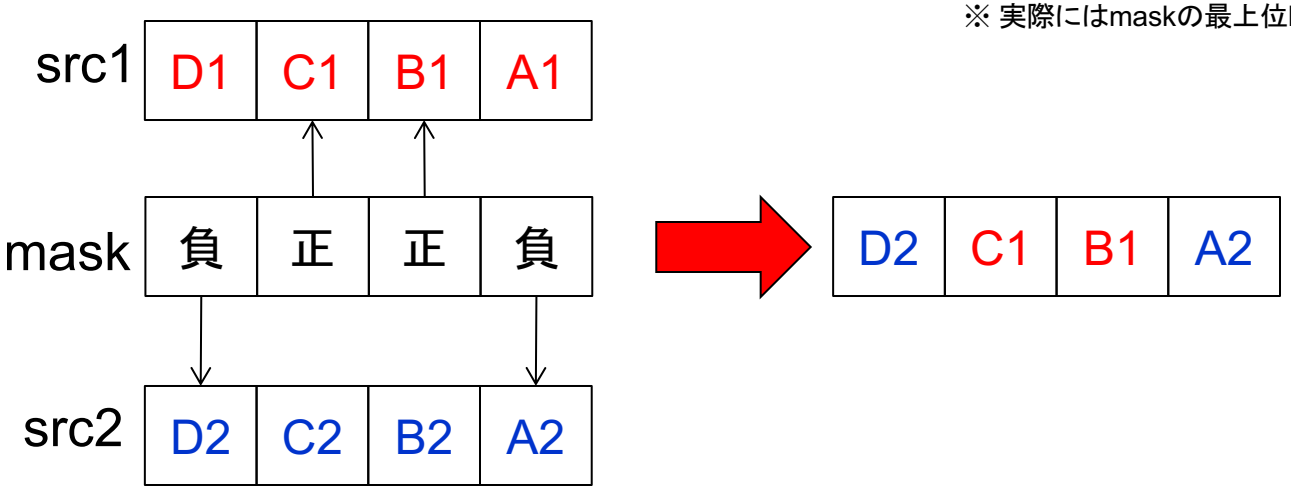
AVX2命令を用いたSIMD化 (3/4)

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

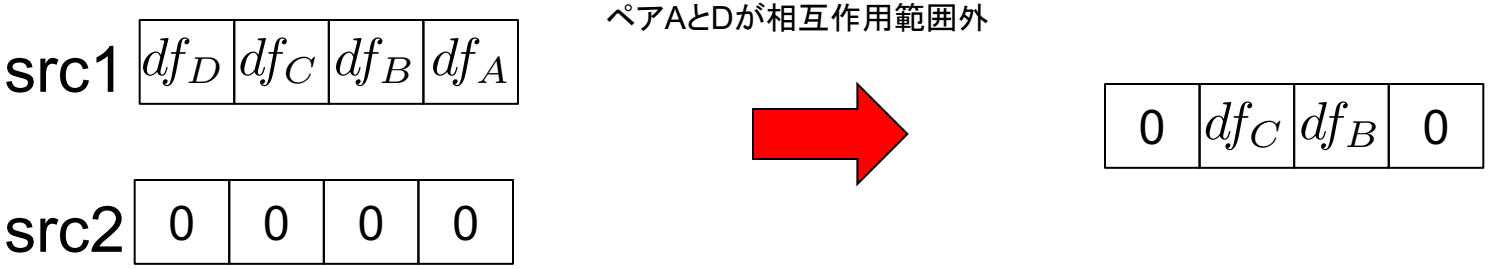
vblendvpd: マスクの要素の正負により要素を選ぶ

※ 実際にはmaskの最上位bitが0か1かで判定している



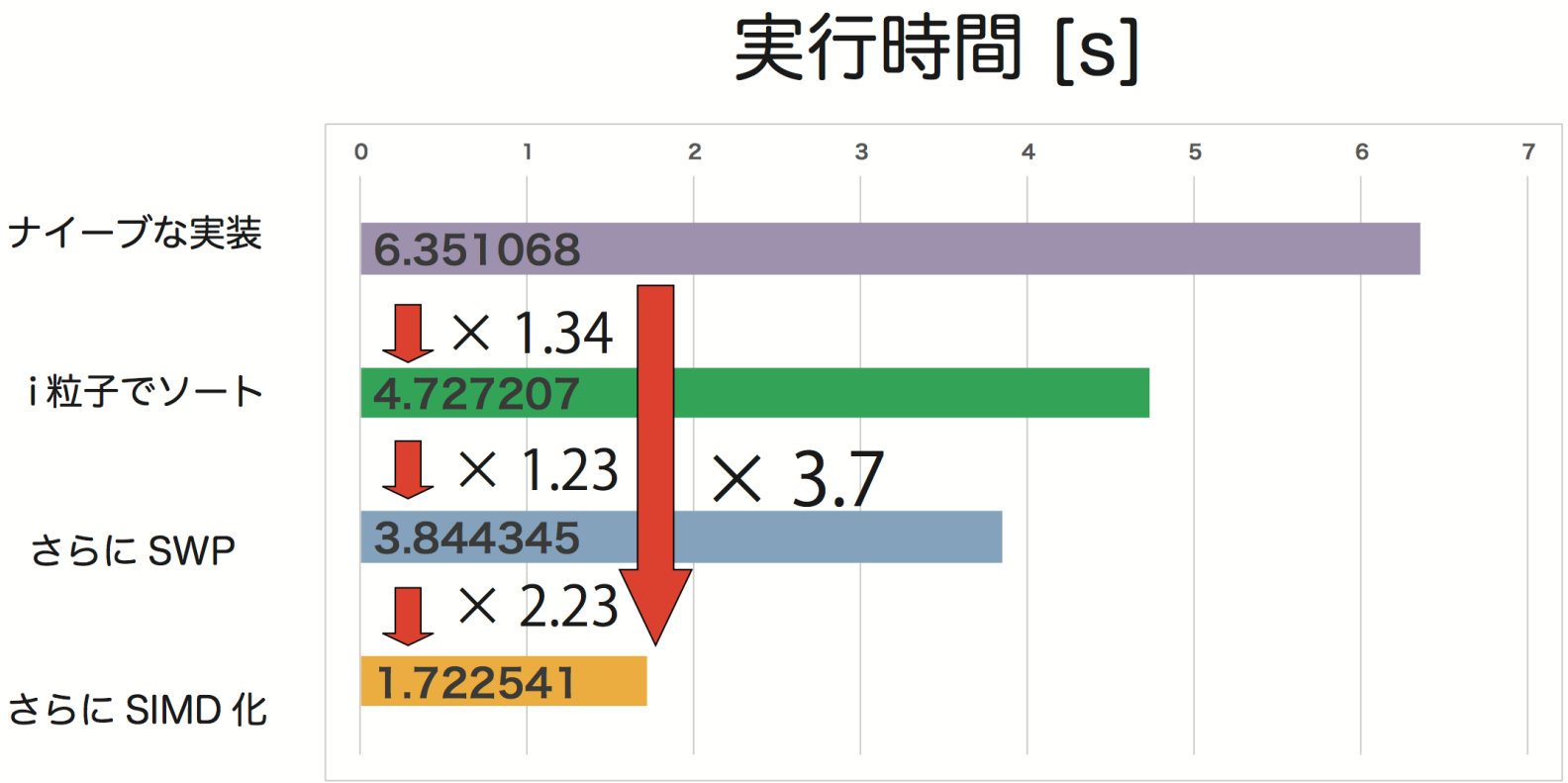
相互作用距離とカットオフ距離でマスクを作成し、力をゼロクリア

ペアAとDが相互作用範囲外



AVX2命令を用いたSIMD化 (4/4)

・12万粒子、密度1.0、カットオフ3.0
・力計算(力積の書き戻しまで)を100回行うのにかかった時間



https://github.com/kaityo256/lj_simdstep

SIMD以外でできるだけ最適化したところから、さらに2倍以上高速化

※ SWP, ソフトウェアパイプライニング。ここでは詳細は省略。

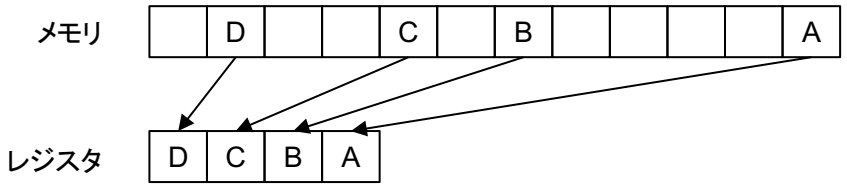
AVX-512命令を用いたSIMD化 (1/7)

AVX-512とは

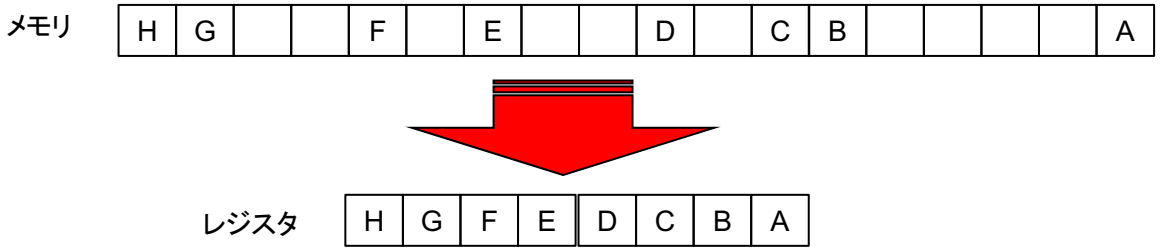
- 512bit幅のSIMD
- 様々な命令の追加 (gather/scatter)
- XeonPhi (Knights Landing, KNL)に実装されている
- 様々な命令セットがある (AVX-512F, AVX-512ER, AVX-512VL ...)

gather/scatter

AVX2まではバラバラに持ってくると遅かった



AVX-512では、メモリから8要素同時にバラバラにロードできる (gather)

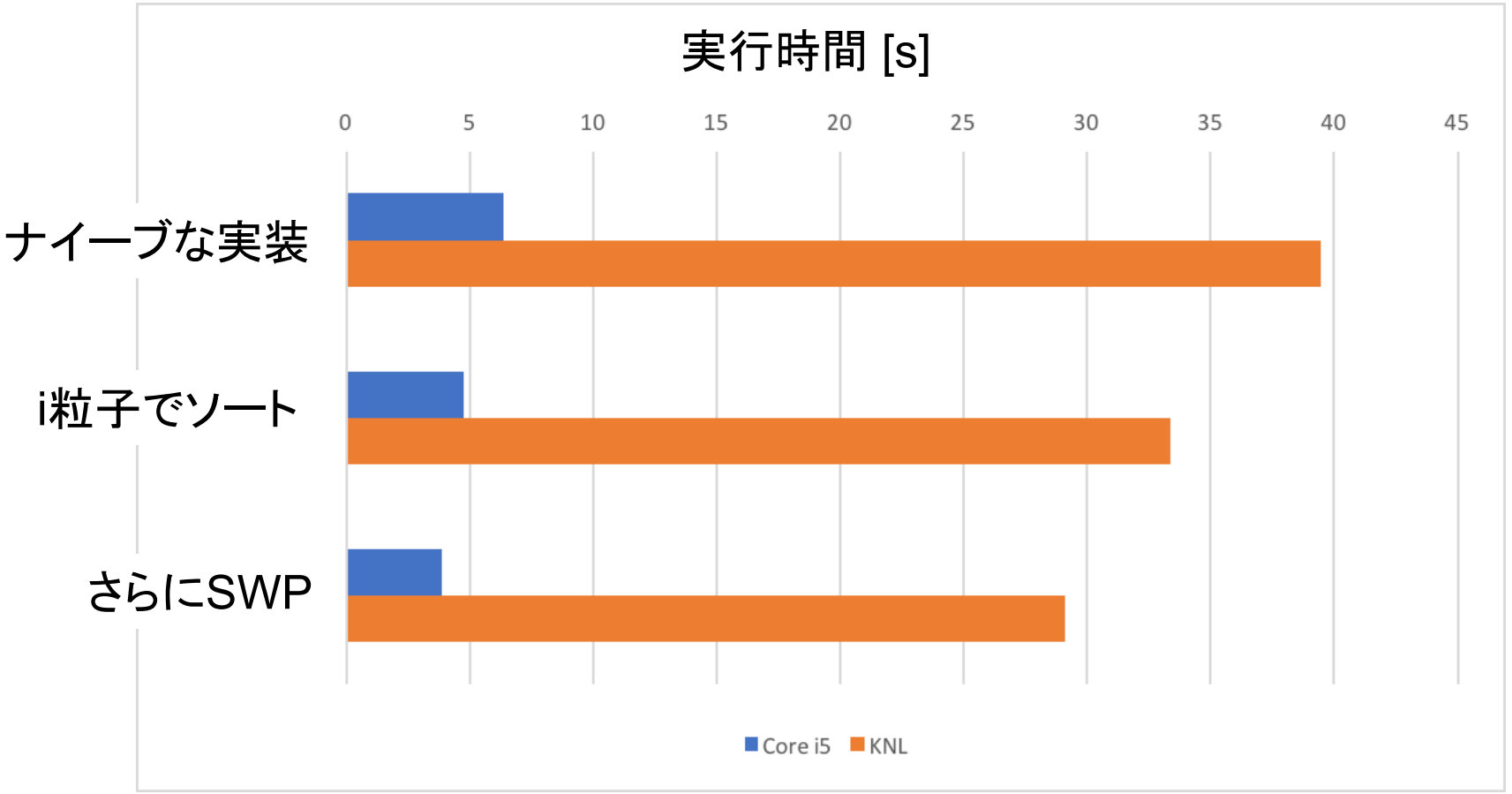


バラバラにストアできるscatterもある

AVX-512命令を用いたSIMD化 (2/7)

SIMD化前

KNL向け実装は物性研の中川さんのコードを参考にしました
https://github.com/kohnakagawa/lj_knl



・KNLのシングルコアの実行速度はCore i5 (Skylake)の6〜7倍遅い

AVX-512命令を用いたSIMD化 (3/7)

データ構造の変更

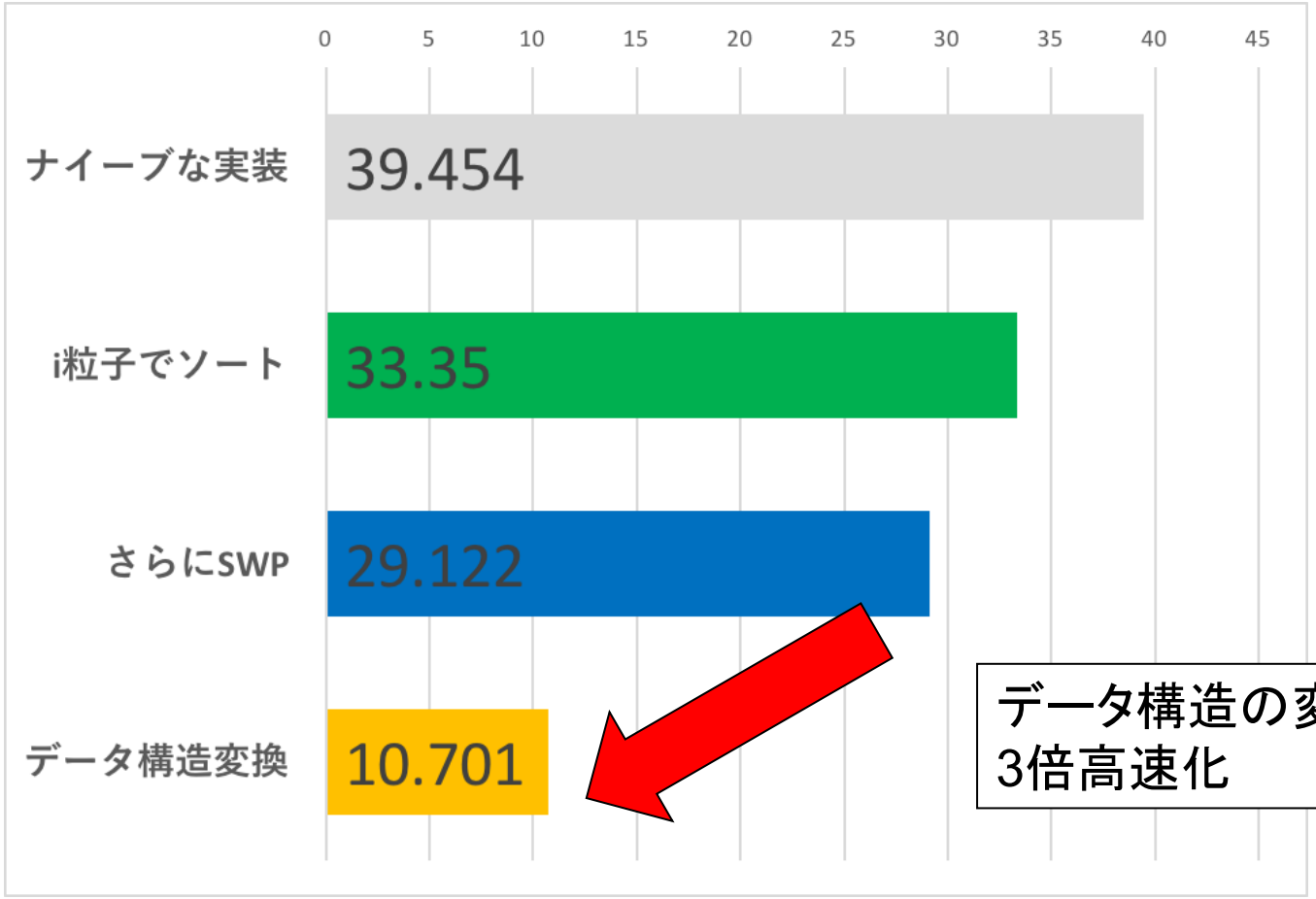
Array of Structure (AoS)

double q[N][4], p[N][4];

ひっくり返す

Structure of Array (SoA)

double q[3][N], p[3][N];

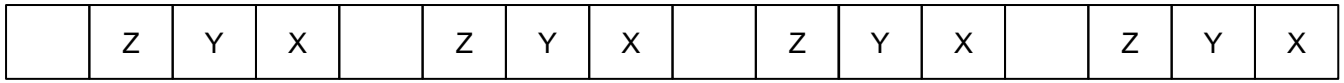


データ構造の変更により
3倍高速化

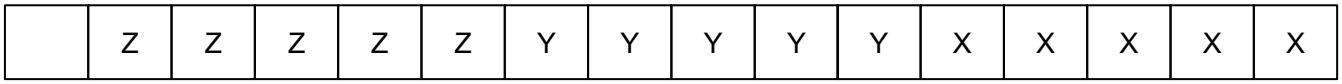
AVX-512命令を用いたSIMD化 (4/7)

何が起きたか？

AoSデータ構造　　データが粒子ごとに固まっている



SoAデータ構造　　データが座標ごとに固まっている



コンパイラがこのパターンは「gather/scatterが使える」と判断
gather/scatter命令を発行した

```
for (int k = 0; k < np; k++) {
     const auto j = sorted_list[kp + k];
     const auto dx = q[X][j] - qix;
     const auto dy = q[Y][j] - qiy;
     const auto dz = q[Z][j] - qiz;
     //calculate force
  }
```

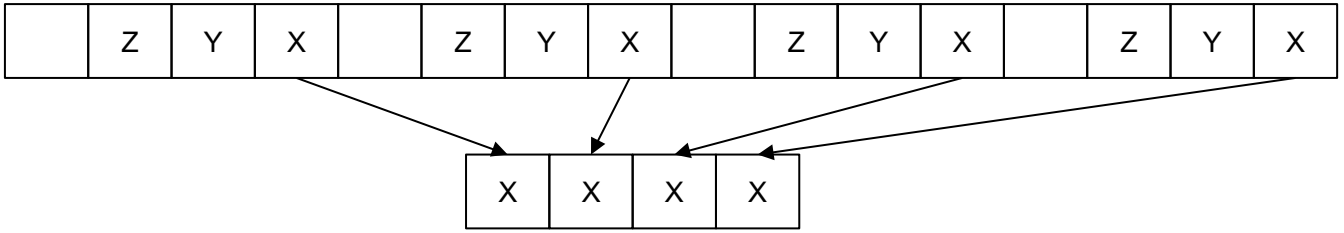
8要素がSIMDレジスタにロードされ、8ペア同時計算

AVX-512命令を用いたSIMD化 (5/7)

AoSでは？

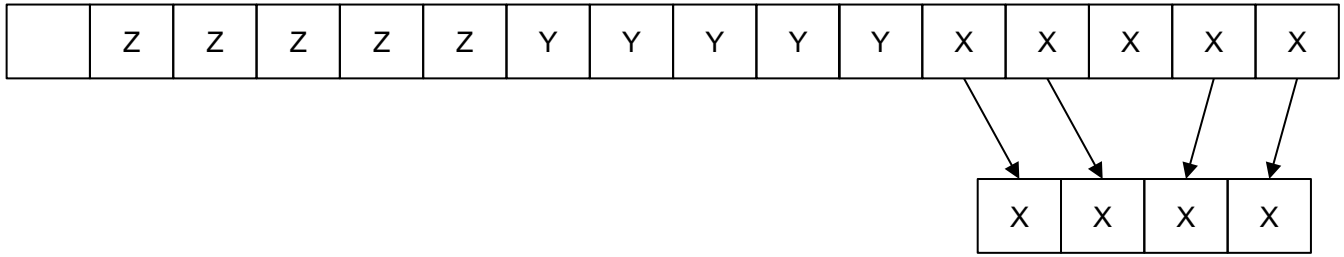
AoSでもgatherは使えるが、コンパイラは自動で判断できなかった
手で書けばgather/scatterは使える

AoS



こちらの方がコンパイラにはわかりやすかった

SoA



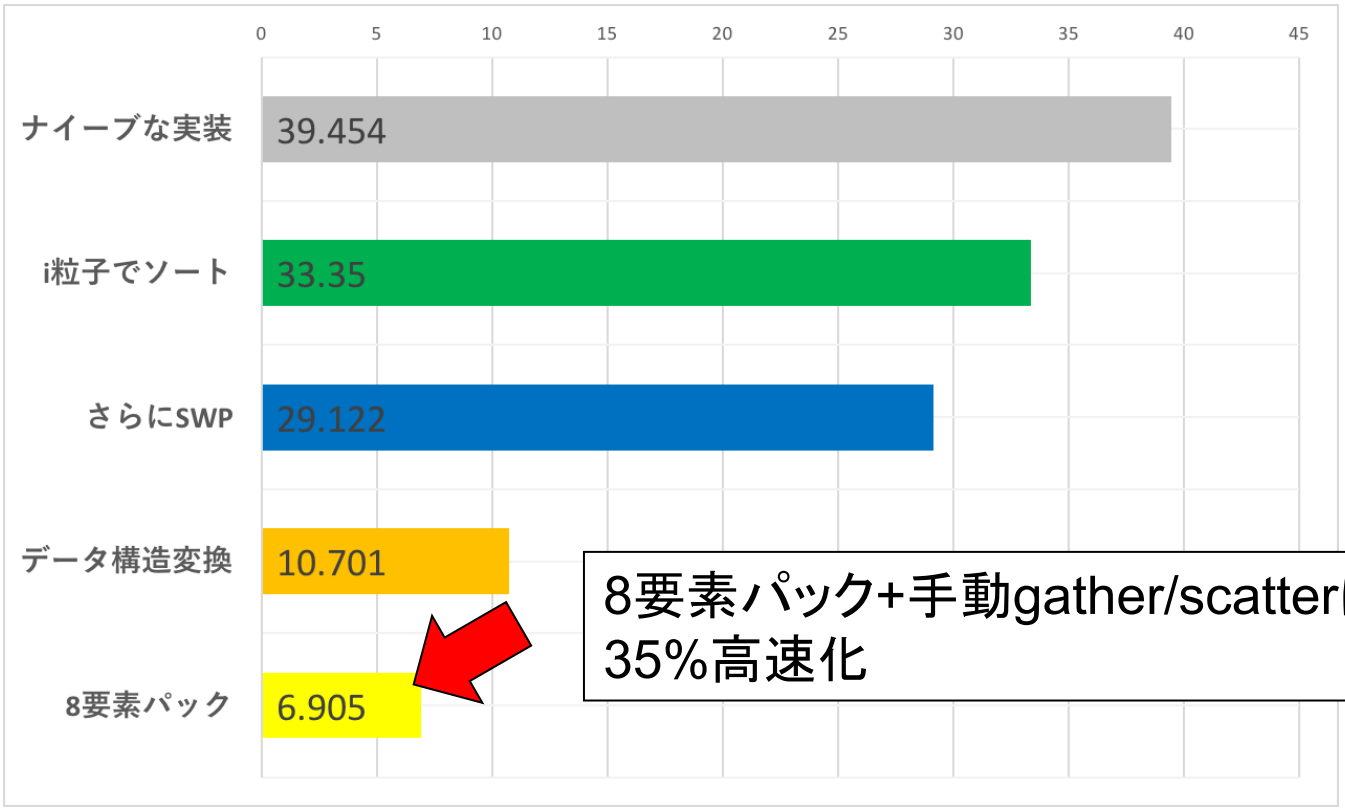
AVX-512命令を用いたSIMD化 (6/7)

SoAとAoS

gather/scatterを手で書けば、SoAとAoSの速度はどちらも大差ないが
ここでもう少しデータ構造を工夫してみる

Array of Structure (AoS)
double q[N][4], p[N][4];

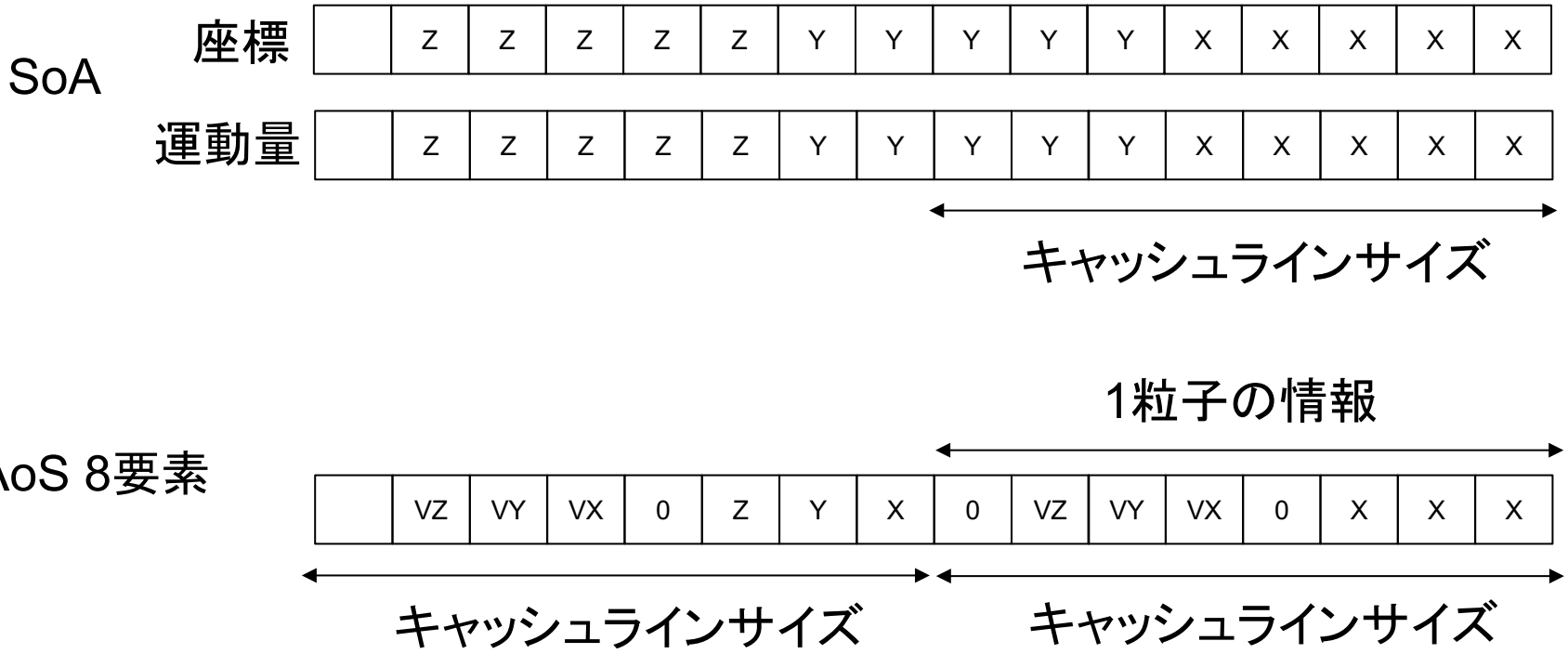
運動量と座標をまとめてしまう
double z[N][8];



8要素パック+手動gather/scatterにより
35%高速化

AVX-512命令を用いたSIMD化 (7/7)

何が起きたか？



- 一つの粒子が、ちょうど全部ひとつのキャッシュラインに乗る
- 座標にアクセスしたら、運動量の情報もキャッシュにもってくる
  (運動量の書き戻しの際、あとで必要になる情報)
- キャッシュミスの低下→性能向上

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

AVX2の場合

データをAoSの4要素パックにして手動SIMD化することで2.2倍高速化

AVX512の場合

AoSからSoAにしたらコンパイラが自動SIMD化して3倍高速化
AoS 8要素パックにして手動SIMD化したらさらに35%高速化

どこまでやるかは貴方次第・・・

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