Day 7 : SIMD 化

はじめに

ここまで読んだ人、お疲れ様です。ここから読んでいる人、それでも問題ありません。これまで、主に並列化についてだらだら書いてきたが、最後はシングルコアでの最適化技術である SIMD 化について説明してみたいと思う。

SIMD とは

スパコンプログラミングに興味があるような人なら、「SIMD」という言葉を聞いたことがあるだろう。SIMD とは、「single instruction multiple data」の略で、「一回の命令で複数のデータを同時に扱う」という意味である。先に、並列化は大きく分けて「データ並列」「共有メモリ並列」「分散メモリ並列」の三種類になると書いたが、SIMD はデータ並列 (Data parallelism) に属す。現在、一般的に数値計算に使われる CPU にはほとんど SIMD 命令が実装されている。後述するが、SIMD とは 1 サイクルに複数の演算を同時に行う技術であり、CPU の「理論ピーク性能」は、SIMD の能力を使い切った場合の性能を指す。したがって、 **まったく SIMD 化できなければ、ピーク性能が数分の 1 になることと等価である。**ここでは、なぜ SIMD が必要になるか、そして SIMD とは何かについて見てみよう。

計算機というのは、要するにメモリからデータと命令を取ってきて、演算器に投げ、結果をメモリに書き戻す機械である。 CPU の動作単位は「サイクル」で表される。演算器に計算を投げてから、結果が返ってくるまでに数サイクルかかるが、現代の CPU ではパイプライン処理という手法によって事実上 1 サイクルに 1 個演算ができる。 1 サイクル 1 演算できるので、あとは「1 秒あたりのサイクル数=動作周波数」を増やせば増やすほど性能が向上することになる。

というわけで CPU ベンダーで動作周波数を向上させる熾烈な競争が行われたのだが、2000 年代に入って動作周波数は上がらなくなった。これはリーク電流による発熱が主な原因なのだが、ここでは深く立ち入らない。 1 サイクルに 1 演算できる状態で、動作周波数をもう上げられないのだから、性能を向上させるためには「1 サイクルに複数演算」をさせなければならない。この「1 サイクルに複数演算」の実現にはいくつかの方法が考えられた。

まず、単純に演算器の数を増やすという方法が考えられる。 1 サイクルで命令を複数取ってきて、その中に独立に実行できるものがあれば、複数の演算器に同時に投げることで 1 サイクルあたりの演算数を増やそう、という方法論である。これを**スーパースカラ**と呼ぶ。独立に実行できる命令がないと性能が上がらないため、よくアウトオブオーダー実行と組み合わされる。要するに命令をたくさん取ってきて命令キューにためておき、スケジューラがそこを見て独立に実行できる命令を選んでは演算器に投げる、ということをする。

この方法には「命令セットの変更を伴わない」という大きなメリットがある。ハードウェアが勝手に並列実行できる命令を探して同時に実行してくれるので、プログラマは何もしなくて良い。そういう意味において、 **スーパースカラとはハードウェアにがんばらせる方法論**である。デメリット、というか問題点は、実行ユニットの数が増えると、依存関係チェックの手間が指数関数的に増えることである。一般的には整数演算が 4 つ程度、浮動小数点演算が 2 つくらいまでが限界だと思われる。

さて、スーパースカラの問題点は、命令の依存関係チェックが複雑であることだった。そこさえ解決できるなら、演算器をたくさん設置すればするほど性能が向上できると期待できる。そこで、事前に並列に実行できる命令を並べておいて、それをそのままノーチェックで演算器に流せばいいじゃないか、という方法論が考えられた。整数演算器や浮動小数点演算器、メモリのロードストアといった実行ユニットを並べておき、それらに供給する命令を予め全部ならべたものを「一つの命令」とする。並列実行できる実行ユニットの数だけ命令を「パック」したものを一つの命令にとするため、命令が極めて長くなる。そのため、こ

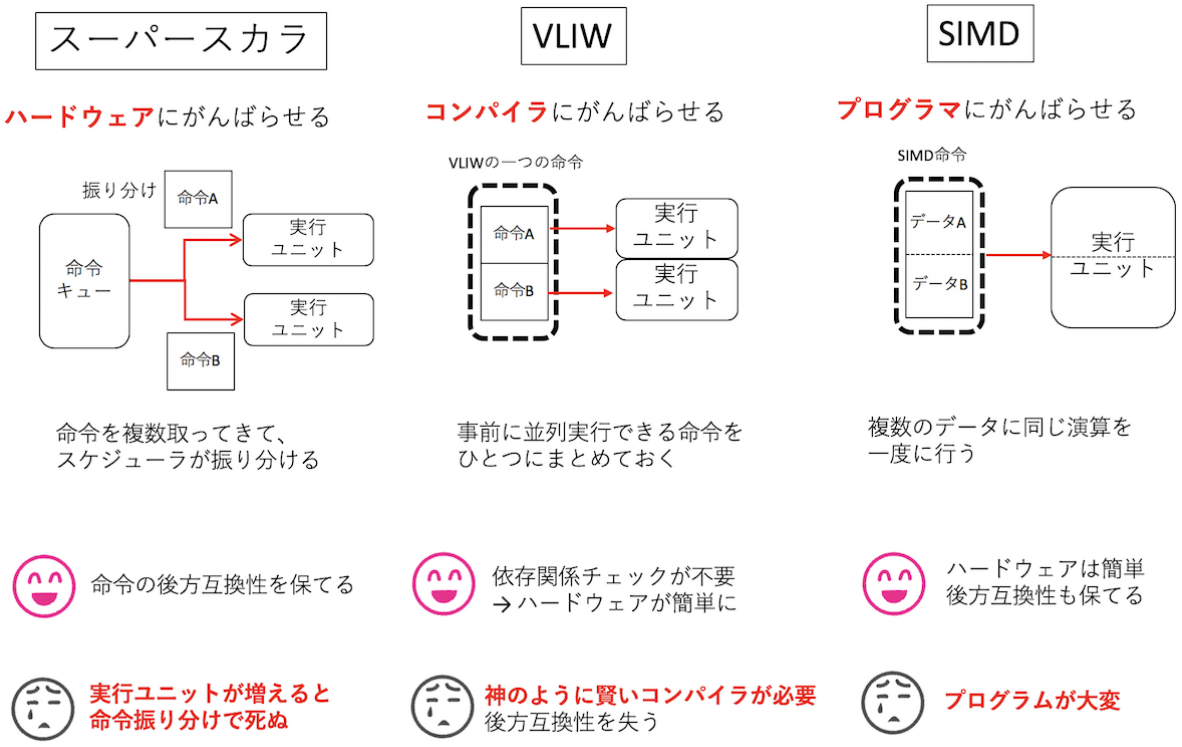

図 1: fig/simd.png

の方式は「Very Long Instruction Word (超長い命令語)」、略して VLIW と呼ばれる。実際にはコンパイラがソースコードを見て並列に実行できる命令を抽出し、なるべく並列に実行できるように並べて一つの命令を作る。そういう意味において、**VLIW とはコンパイラにがんばらせる方法論**である。

この方式で性能を稼ぐためには、VLIW の「命令」に有効な命令が並んでなければならない。しかし、命令に依存関係が多いと同時に使える実行ユニットが少なくなり、「遊ぶ」実行ユニットに対応する箇所には「NOP(no operation = 何もしない)」が並ぶことになる。 VLIW は Intel と HP が共同開発した IA-64 というアーキテクチャに採用され、それを実装した Itanium2 は、ハイエンドサーバやスパコン向けにそれなりに採用された。個人的には Itanium2 は (レジスタもいっぱいあって) 好きな石なのだが、この方式が輝くためには「神のように賢いコンパイラ」が必須となる。一般に「神のように賢いコンパイラ」を要求する方法はだいたい失敗する運命にある。また、命令セットが実行ユニットの仕様と強く結びついており、後方互換性に欠けるのも痛い。 VLIW は組み込み用途では人気があるものの、いまのところハイエンドな HPC 向けとしてはほぼ滅びたと言ってよいと思う。

さて、「ハードウェアにがんばらせる」方法には限界があり、「コンパイラにがんばらせる」方法には無理があった。残る方式は**プログラマにがんばらせる方法論** だけである。それが SIMD である。

異なる命令をまとめてパックするのは難しい。というわけで、命令ではなく、データをパックすることを考える。たとえば「C=A+B」という足し算を考える。これは「A と B というデータを取ってきて、加算し、C としてメモリに書き戻す」という動作をする。ここで、「C1 = A1+B1」「C2 = A2+B2」という独立な演算があったとしよう。予め「A1:A2」「B1:B2」とデータを一つのレジスタにパックしておき、それぞれの和を取ると「C1:C2」という、2 つの演算結果がパックしたレジスタが得られる。このように複数のデータをパックするレジスタを SIMD レジスタと呼ぶ。例えば AVX2 なら 256 ビットの SIMD レジスタがあるため、64 ビットの倍精度実数を 4 つパックできる。そしてパックされた 4 つのデータ間に独立な演算を実行することができる。ここで、ハードウェアはパックされたデータの演算には依存関係が無いことを

仮定する。つまり依存関係の責任はプログラマ側にある。ハードウェアから見れば、レジスタや演算器の「ビット幅」が増えただけのように見えるため、ハードウェアはさほど複雑にならない。しかし、性能向上のためには、SIMD レジスタを活用したプログラミングを行わなければならない。 SIMD レジスタを活用して性能を向上させることを俗に「SIMD 化 (SIMD-vectorization)」などと呼ぶ。原理的にはコンパイラによって SIMD 化することは可能であり、実際に、最近のコンパイラの SIMD 最適化能力の向上には目を見張るものがある。しかし、効果的な SIMD 化のためにはデータ構造の変更を伴うことが多く、コンパイラにはそういったグローバルな変更を伴う最適化が困難であることから、基本的には「SIMD 化はプログラマが手で行う必要がある」のが現状である。

SIMD レジスタを触ってみる

SIMD 化とは、CPU に実装されている SIMD レジスタをうまく使うコードを書いて実行速度を加速させることである。そのためには、まず SIMD レジスタを使う必要がある。 SIMD レジスタとは、要するに複数のデータを一度に保持できる変数である。これを読んでいる人の大多数は AVX2 に対応した x86 系 CPU のパソコンを持っているだろう。まずは AVX2 の 256bit レジスタ、YMM レジスタを使ってみよう。以下、変数としては倍精度実数型を使う。倍精度実数は 64 ビットなので、256 ビットレジスタは倍精度実数を 4 つ保持できる。

SIMD を明示的に扱うには、まず `x86intrin.h` を include する。すると、SIMD レジスタに対応する `_m256d` という型が使えるようになる。これは 256bit の YMM レジスタを、倍精度実数 4 つだと思って使う型である。この型の変数に値を入れるには、例えば `_mm256_set_pd` という組み込み関数を使う。

```
__m256d v1 =  _mm256_set_pd(3.0, 2.0, 1.0, 0.0);
```

この関数は、 右から下位に値を放り込んでいく。上記の例なら、一番下位の 64 ビットに倍精度実数の 0、次の 64 ビットに 1 … と値が入る。さて、SIMD レジスタとは、変数を複数同時に保持できるものである。今回のケースでは、`_m256d` という型は、ほぼ `double [4]` だと思ってかまわない。実際、これらはそのままキャスト可能である。 SIMD 化を行う時、こんなデバッグ用の関数を作っておくと便利である。

```
void print256d(__m256d x) {
  printf("%f %f %f %f\n", x[3], x[2], x[1], x[0]);
}
```

`_m256d x` が、そのまま `double x[4]` として使えているのがわかると思う。この時、`x[0]` が一番下位となる。先程の代入と合わせるとこんな感じになる。

print.cpp

```
#include <cstdio>
#include <x86intrin.h>

void print256d(__m256d x) {
  printf("%f %f %f %f\n", x[3], x[2], x[1], x[0]);
}

int main(void) {
  __m256d v1 =  _mm256_set_pd(3.0, 2.0, 1.0, 0.0);
  print256d(v1);
}
```

これを g++でコンパイルするには、AVX2 を使うよ、と教えてあげる必要がある。

```
$ g++ -mavx2 print.cpp
$ ./a.out
3.000000 2.000000 1.000000 0.000000
```

ちゃんと SIMD レジスタに値が代入され、それが表示されたことがわかる。

さて、SIMD レジスタの強みは、SIMD レジスタ同士で四則演算をすると、4 つ同時に計算が実行できることであった。それを確認してみよう。

単に_m256d 同士の足し算をやる関数を書いてみる。_m256d の型は、そのまま四則演算ができる。 double[4] の型をラップしたクラスを作り、オペレータのオーバーロードをしたようなイメージである。

```
#include <cstdio>
#include <x86intrin.h>

__m256d add(__m256d v1, __m256d v2) {
  return v1 + v2;
}
```

アセンブリを見てみる。少し最適化したほうがアセンブリが読みやすい。

```
g++ -mavx2 -O2 -S add.cpp
```

アセンブリはこうなる。

```
__Z3addDv4_dS_:
  vaddpd  %ymm1, %ymm0, %ymm0
  ret
```

vaddpd は SIMD の足し算を行う命令であり、ちゃんと YMM レジスタの足し算が呼ばれていることがわかる。

実際に 4 要素同時に足し算できることを確認しよう。

add.cpp

```
#include <cstdio>
#include <x86intrin.h>

void print256d(__m256d x) {
  printf("%f %f %f %f\n", x[3], x[2], x[1], x[0]);
}

int main(void) {
  __m256d v1 =  _mm256_set_pd(3.0, 2.0, 1.0, 0.0);
  __m256d v2 =  _mm256_set_pd(7.0, 6.0, 5.0, 4.0);
  __m256d v3 = v1 + v2;
  print256d(v3);
}
```

```
$ g++ -mavx2 add.cpp
$ ./a.out
10.000000 8.000000 6.000000 4.000000
```

(0,1,2,3) というベクトルと、(4,5,6,7) というベクトルの和をとり、(4,6,8,10) というベクトルが得られた。このように、ベクトル同士の演算に見えるので、SIMD 化のことをベクトル化と呼んだりする。ただし、線形代数で出てくるベクトルの積とは違い、SIMD の積は単に要素ごとの積になることに注意。実際、さっきの和を積にするとこうなる。

mul.cpp

```
#include <cstdio>
#include <x86intrin.h>

void print256d(__m256d x) {
  printf("%f %f %f %f\n", x[3], x[2], x[1], x[0]);
}

int main(void) {
  __m256d v1 =  _mm256_set_pd(3.0, 2.0, 1.0, 0.0);
  __m256d v2 =  _mm256_set_pd(7.0, 6.0, 5.0, 4.0);
  __m256d v3 = v1 * v2; // 積にした
  print256d(v3);
}
```

```
$ g++ -mavx2 mul.cpp
$ ./a.out
21.000000 12.000000 5.000000 0.000000
```

それぞれ、0*0、1*5、2*6、3*7 が計算されていることがわかる。

あと SIMD 化で大事なのは、SIMD レジスタへのデータの読み書きである。先程はデバッグのために _mm256_set_pd を使ったが、これは極めて遅い。どんな動作をするか見てみよう。

setpd.cpp

```
#include <x86intrin.h>

__m256d setpd(double a, double b, double c, double d) {
  return _mm256_set_pd(d, c, b, a);
}
```

このアセンブリを見てみる。

```
g++ -mavx2 -O2 -S setpd.cpp
```

```
__Z5setpddddd:
  vunpcklpd %xmm3, %xmm2, %xmm2
  vunpcklpd %xmm1, %xmm0, %xmm0
  vinsertf128 $0x1, %xmm2, %ymm0, %ymm0
  ret
```

これは、

1. a と b を xmm2 レジスタにパック
2. c と d を xmm0 レジスタにパック
3. xmm2 レジスタの値を ymm0 レジスタの上位に挿入

ということをしている。ここで、xmm0 レジスタと ymm0 レジスタの下位 128 ビットは共有していることに注意。つまり、xmm0 レジスタに読み書きすると、ymm0 レジスタの下位 128 ビットも影響を受ける。上記の例はそれを利用して、最終的に欲しい情報、つまり 4 要素をパックしたレジスタを作っている。

とりあえず 4 つの要素を YMM レジスタに載せることができれば、あとは 4 要素同時に計算ができるようになるのだが、4 要素をパックする際に_mm256_set_pd を使うとメモリアクセスが多くなって性能が出ない。そのため、メモリから連続するデータをごそっとレジスタにとってきたり、書き戻したりする命令がある。例えば、_mm256_load_pd は、指定されたポインタから連続する 4 つの倍精度実数をとってきて YMM レジスタに入れてくれる。ただし、そのポインタの指すアドレスは 32 バイトアラインされていなければならない。

利用例はこんな感じになる。

load.cpp

```
#include <cstdio>
#include <x86intrin.h>

void print256d(__m256d x) {
  printf("%f %f %f %f\n", x[3], x[2], x[1], x[0]);
}

__attribute__((aligned(32))) double a[] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0};

int main(void) {
  __m256d v1 = _mm256_load_pd(a);
  __m256d v2 = _mm256_load_pd(a + 4);
  __m256d v3 = v1 + v2;
  print256d(v3);
}
```

ここで__attribute__((aligned(32))) が、「後に続くデータを 32 バイトアライメントにしてください」という指示である。メモリアライメントについてはここでは深入りしないが、「利用メモリの端がちょうど良いとこから始まっていること」とおぼえておけば良い。具体的には、配列の先頭アドレスが 32 で割り切れる値になる。メモリアライメントが合わないと遅くなったり、実行時に SIGSEGV で落ちたりする。

コンパイル、実行はこんな感じ。

```
$ g++ -mavx2 load.cpp
$ ./a.out
10.000000 8.000000 6.000000 4.000000
```

_mm256_load_pd が何をやっているか (どんなアセンブリに対応するか) も見てみよう。こんなコードのアセンブリを見てみる。

loadasm.cpp

```
#include <x86intrin.h>
__m256d load(double *a, int index) {
  return _mm256_load_pd(a + index);
}
```

```
g++ -O2 -mavx2 loadasm.cpp
```

例によって、少し最適化をかけておく。

```
__Z4loadPdi:
  movslq  %esi, %rsi
  vmovapd (%rdi,%rsi,8), %ymm0
  ret
```

ymm0 レジスタに、vmovapd 一発でデータがロードされていることがわかる。

ここでは読み出しを見たが、書き戻し (_mm256_store_pd) も同様である。

さて、これで SIMD 化の基礎は終わりである。ようするにデータを複数個とってきて、SIMD レジスタに乗せて、レジスタでなんか計算して、それを書き戻せばよろしい。簡単でしょう？ただし、バラバラのデータを pack/unpack すると遅いので、なるべく連続したデータをロードしたりストアしたりするように心がける必要がある。実際に使おうとすると、「あれ？if 文があるんだけど、どうすればいいんだろう？」とか「メモリの配置とレジスタに載せたい配置が微妙に違う」とかいろいろ出てくると思うが、それに対応して SIMD のための補助命令 (シャッフルとか) が死ぬほどある。まぁ、それは必要に応じて覚えておけばいいと思う。以下、簡単な例で SIMD 化の実際を見てみよう。

余談：アセンブリ言語？アセンブラ言語？

アセンブリの話になると、必ずといっていいほど「アセンブリ言語 (assembly language) が正しく、アセンブラ言語 (assembler language) は誤り」と言い出す人がいる。もちろんほとんどの場合において、 **アセンブリ言語** (assembly language)」を**アセンブル** (assemble) して**機械語** (machine language) に変換するのが**アセンブラ** (assembler) である、という認識で良い。しかし、「アセンブラ言語 (assembler language)」という用法が無いわけではない。もっとも有名なのが IBM である。 IBM は昔から「アセンブラ言語 (assembler language)」という呼び方をしてきた。 IBM が現在サポートしているアセンブリ言語は IBM High Level Assembler(HLASM) であり、メインフレームである System z 上で動作する。 HLASM のマニュアル内でも「アセンブラ言語 (assembler language)」という呼び方をしている。 しかし、System z の祖先である System 360 上で動作するアセンブリ言語は「IBM Basic Assembly Language (BAL)」と呼ばれており、「Assembly language」と「Assembler language」の使い分けは微妙である。しかし、IBM 自身は BAL のことを Basic Assembler Language と表現しているので、もしかしたら BAL のことを IBM は Basic Assembler Language、サードパーティは Basic Assembly Language と呼んでいるのかもしれない。

IBM 以外でも、ARM が規定したアセンブリ言語である UAL は「Unified Assembler Language」の略である。直訳すると「統合**アセンブラ**言語」とでもなるだろうか。しかし、ARM 自身はアセンブリ言語のことを「assembly language」と呼んでおり、やはりその使い分けは微妙である。これは筆者の経験になるが、日本でも昔は「アセンブリ言語」のことを指して「アセンブラ」と呼ぶのがかなり一般的であった記憶がある。 C 言語などを使わずに書くことを「フルアセンブラで書く」といった具合である。いまでもそのような使い方をしている人を見かける。どうでもいいが、当時 x86 のアセンブラを書いていた人は、機械語のことを「マシン語」と呼ぶのが一般的であった気もする。

繰り返しになるが、現在は「アセンブリ言語 (assembly language)」の方が一般的な用語であると思われるので、「アセンブリ言語をアセンブラがアセンブルして機械語にする」と表現することになんの問題もない。しかし、誰かが「アセンブラを書く」もしくは「アセンブラ言語」と言ったときに、脊髄反射で「アセンブリ言語が正しい」とマウントを取る前に、上記のような事情を思い出していただけたらと思う。

余談の余談となるが、アセンブリで書かれたものを手で機械語に翻訳する作業を「ハンドアセンブル」と呼ぶ。昔のアセンブリはほぼ機械語と一対一対応しており、「便利なマクロ付き機械語」といった趣であっ

たため、ハンドアセンブルはさほど難しい作業ではなかった。しかし、現在の機械語、特に x86 の機械語はかなり複雑になっており、アセンブリから機械語に翻訳するのはかなり大変になっている。そのあたりは例えば x86_64 機械語入門なんかを参照してほしい。

簡単な SIMD 化の例

では、実際に SIMD 化をやってみよう。こんなコードを考える。一次元の配列の単純な和のループである。

func.cpp

```
const int N = 10000;
double a[N], b[N], c[N];

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

これを普通にコンパイルすると、こんなアセンブリになる。

```
g++ -O1 -S func.cpp
```

```
  xorl  %eax, %eax
  leaq  _a(%rip), %rcx
  leaq  _b(%rip), %rdx
  leaq  _c(%rip), %rsi
  movsd (%rax,%rcx), %xmm0
  addsd (%rax,%rdx), %xmm0
  movsd %xmm0, (%rax,%rsi)
  addq  $8, %rax
  cmpq  $80000, %rax
  jne LBB0_1
```

配列 a,b,c のアドレスを%rcx, %rdx, %rsi に取得し、movsd で a[i] のデータを%xmm0 に持ってきて、addsd で a[i]+b[i] を計算して%xmm0 に保存し、それを c[i] の指すアドレスに movsd で書き戻す、ということをやっている。これはスカラーコードなのだが、xmm は 128 ビット SIMD レジスタである。 x86 は歴史的経緯からスカラーコードでも浮動小数点演算に SIMD レジスタである xmm を使う (後述の余談参照)。

さて、このループを AVX2 を使って SIMD 化することを考える。 SIMD 化の基本は、ループをアンロールして独立な演算を複数作り、それを SIMD レジスタで同時に演算することである。 AVX2 の SIMD レジスタは ymm で表記される。 ymm レジスタは 256 ビットで、倍精度実数 (64 ビット) が 4 つ保持できるので、

- ループを 4 倍展開する
- 配列 a から 4 つデータを持ってきて ymm レジスタに乗せる
- 配列 b から 4 つデータを持ってきて ymm レジスタに乗せる
- 二つのレジスタを足す
- 結果のレジスタを配列 c のしかるべき場所に保存する

ということをすれば SIMD 化完了である。コードを見たほうが早いと思う。

func_simd.cpp

```cpp
#include <x86intrin.h>
void func_simd() {
  for (int i = 0; i < N; i += 4) {
    __m256d va = _mm256_load_pd(&(a[i]));
    __m256d vb = _mm256_load_pd(&(b[i]));
    __m256d vc = va + vb;
    _mm256_store_pd(&(c[i]), vc);
  }
}
```

先程みたように、4 つ連続したデータを持ってくる命令が_mm256_load_pd であり、SIMD レジスタの内容をメモリに保存する命令が_mm256_load_pd である。これをコンパイルしてアセンブリを見てみよう。

```
g++ -O1 -mavx2 -S func_simd.cpp
```

```
  xorl  %eax, %eax
  leaq  _a(%rip), %rcx
  leaq  _b(%rip), %rdx
  leaq  _c(%rip), %rsi
  xorl  %edi, %edi
LBB0_1:
  vmovupd (%rax,%rcx), %ymm0          # (a[i],a[i+1],a[i+2],a[i+3]) -> ymm0
  vaddpd  (%rax,%rdx), %ymm0, %ymm0   # ymm0 + (b[i],b[i+1],b[i+2],b[i+3]) -> ymm 0
  vmovupd %ymm0, (%rax,%rsi)          # ymm0 -> (c[i],c[i+1],c[i+2],c[i+3])
  addq  $4, %rdi    # i += 4
  addq  $32, %rax
  cmpq  $10000, %rdi
  jb  LBB0_1
```

ほとんどそのままなので、アセンブリ詳しくない人でも理解は難しくないと思う。配列のアドレスを%rcx, %rdx, %rsi に取得するところまでは同じ。元のコードでは movsd で xmm レジスタにデータをコピーしていたのが、vmovupd で ymm レジスタにデータをコピーしているのがわかる。どの組み込み関数がどんな SIMD 命令に対応しているかは Intel Intrinsics Guide が便利である。

念の為、このコードが正しく計算できてるかチェックしよう。適当に乱数を生成して配列 a[N] と b[N] に保存し、ついでに答えも ans[N] に保存しておく。

```cpp
int main() {
  std::mt19937 mt;
  std::uniform_real_distribution<double> ud(0.0, 1.0);
  for (int i = 0; i < N; i++) {
    a[i] = ud(mt);
    b[i] = ud(mt);
    ans[i] = a[i] + b[i];
  }
  check(func, "scalar");
  check(func_simd, "vector");
}
```

倍精度実数同士が等しいかチェックするのはいろいろと微妙なので、バイト単位で比較しよう。ここでは

配列 c[N] と ans[N] を unsigned char にキャストして比較している。

```
void check(void(*pfunc)(), const char *type) {
  pfunc();
  unsigned char *x = (unsigned char *)c;
  unsigned char *y = (unsigned char *)ans;
  bool valid = true;
  for (int i = 0; i < 8 * N; i++) {
    if (x[i] != y[i]) {
      valid = false;
      break;
    }
  }
  if (valid) {
    printf("%s is OK\n", type);
  } else {
    printf("%s is NG\n", type);
  }
}
```

全部まとめたコードはこちら。

simdcheck.cpp

実際に実行してテストしてみよう。

```
$ g++ -mavx2 -O3 simdcheck.cpp
$ ./a.out
scalar is OK
vector is OK
```

正しく計算できているようだ。

さて、これくらいのコードならコンパイラも SIMD 化してくれる。で、問題はコンパイラが SIMD 化したかどうかをどうやって判断するかである。一つの方法は、コンパイラの吐く最適化レポートを見ることだ。インテルコンパイラなら-qopt-report で最適化レポートを見ることができる。

```
$ icpc -march=core-avx2 -O2 -c -qopt-report -qopt-report-file=report.txt func.cpp
$ cat report.txt
Intel(R) Advisor can now assist with vectorization and show optimization
  report messages with your source code.
(snip)
LOOP BEGIN at func.cpp(5,3)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

実際にはもっとごちゃごちゃ出てくるのだが、とりあえず最後に「LOOP WAS VECTORIZED」とあり、SIMD 化できたことがわかる。しかし、どう SIMD 化したのかはさっぱりわからない。-qopt-report=5 として最適レポートのレベルを上げてみよう。

```
$ icpc -march=core-avx2 -O2 -c -qopt-report=5 -qopt-report-file=report5.txt func.cpp
$ cat report5.txt
```

```
Intel(R) Advisor can now assist with vectorization and show optimization
  report messages with your source code.
(snip)
LOOP BEGIN at func.cpp(5,3)
   remark #15388: vectorization support: reference c has aligned access   [ func.cpp(6,5) ]
   remark #15388: vectorization support: reference a has aligned access   [ func.cpp(6,5) ]
   remark #15388: vectorization support: reference b has aligned access   [ func.cpp(6,5) ]
   remark #15305: vectorization support: vector length 4
   remark #15399: vectorization support: unroll factor set to 4
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 2
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 6
   remark #15477: vector loop cost: 1.250
   remark #15478: estimated potential speedup: 4.800
   remark #15488: --- end vector loop cost summary ---
   remark #25015: Estimate of max trip count of loop=625
LOOP END
```

このレポートから以下のようなことがわかる。

- ymm レジスタを使った SIMD 化であり (vector length 4)
- ループを 4 倍展開しており (unroll factor set to 4)
- スカラーループのコストが 6 と予想され (scalar loop cost: 6)
- ベクトルループのコストが 1.250 と予想され (vector loop cost: 1.250)
- ベクトル化による速度向上率は 4.8 倍であると見積もられた (estimated potential speedup: 4.800)

でも、こういう時にはアセンブリ見ちゃった方が早い。

```
icpc -march=core-avx2 -O2 -S func.cpp
```

コンパイラが吐いたアセンブリを、少し手で並び替えたものがこちら。

```
        xorl      %eax, %eax
..B1.2:
        lea       (,%rax,8), %rdx
        vmovupd   a(,%rax,8), %ymm0
        vmovupd   32+a(,%rax,8), %ymm2
        vmovupd   64+a(,%rax,8), %ymm4
        vmovupd   96+a(,%rax,8), %ymm6
        vaddpd    b(,%rax,8), %ymm0, %ymm1
        vaddpd    32+b(,%rax,8), %ymm2, %ymm3
        vaddpd    64+b(,%rax,8), %ymm4, %ymm5
        vaddpd    96+b(,%rax,8), %ymm6, %ymm7
        vmovupd   %ymm1, c(%rdx)
        vmovupd   %ymm3, 32+c(%rdx)
        vmovupd   %ymm5, 64+c(%rdx)
        vmovupd   %ymm7, 96+c(%rdx)
        addq      $16, %rax
```

```
cmpq      $10000, %rax
jb        ..B1.2
```

先程、手でループを 4 倍展開して SIMD 化したが、さらにそれを 4 倍展開していることがわかる。

これ、断言しても良いが、こういうコンパイラの最適化について調べる時、コンパイラの最適化レポートとにらめっこするよりアセンブリ読んじゃった方が絶対に早い。「アセンブリを読む」というと身構える人が多いのだが、どうせ SIMD 化で使われる命令ってそんなにないし、最低でも「どんな種類のレジスタが使われているか」を見るだけでも「うまく SIMD 化できてるか」がわかる。ループアンロールとかそういうアルゴリズムがわからなくても、アセンブリ見て xmm だらけだったら SIMD 化は (あまり) されてないし、ymm レジスタが使われていれば AVX/AVX2 を使った SIMD 化で、zmm レジスタが使われていれば AVX-512 を使った SIMD 化で、`vmovupd` が出てればメモリからスムーズにデータがコピーされてそうだし、`vaddpd` とか出てればちゃんと SIMD で足し算しているなとか、そのくらいわかれば実用的にはわりと十分だったりする。そうやっていつもアセンブリを読んでいると、そのうち「アセンブリ食わず嫌い」が治って、コンパイラが何を考えたかだんだんわかるようになる……かもしれない。

余談：x86 における浮動小数点演算の扱い

数値計算においては、浮動小数点演算が必須である。この浮動小数点数は、単精度なら 32 ビット、倍精度なら 64 ビットで表現される。その表現方法は IEEE 754 という仕様で決まっている。 CPU は演算をレジスタで行うが、整数演算を行う汎用レジスタと、浮動小数点演算を行う浮動小数点レジスタは別々に用意されているのが一般的である。そして、倍精度実数演算をサポートする CPU は 64 ビットの浮動小数点レジスタを持っていることが多いのだが、x86 は例外で 64 ビットの浮動小数点レジスタが無い。ではどうするかというと、普通の浮動小数点演算に 128 ビットの XMM という SIMD レジスタを使う。これには x86 における浮動小数点演算のサポートの歴史的な事情がある。

もともと、x86 はハードウェアレベルで浮動小数点演算ができなかった。したがって、浮動小数点演算をやろうとすると、汎用レジスタでソフトウェア的に浮動小数点演算をエミュレートしてやる必要があり、極めて遅かった。そこで、浮動小数点演算をサポートするために x87 というコプロセッサが用意された。コプロセッサとは、CPU の近くに外付けして機能拡張する装置で、コプロセッサに対応する命令が来ると、CPU からコプロセッサに処理が依頼される。ユーザからは CPU の命令セットが拡張されたように見える。この x87 コプロセッサは 80 ビットマシンであり、単精度 (32 ビット) でも倍精度 (64 ビット) でも、一度内部で 80 ビットに変換されてから計算され、演算結果をまた単精度なり倍精度なりに変換する。なお、`long double` 型は、80 ビットそのままで計算され、変換を伴わない。 x86 系の石で `long double` が 80 ビットなのはこういう事情による。

x87 コプロセッサは、x86 の進化に合わせて一緒に進化していった。 Intel 8086 には Intel 8087 が、80186 には 80187 が、80286 には 80287 が、といった具合である。なので x86 と同様に x87 と呼ばれる。 x87 はしばらく外付けのコプロセッサだったが、80486 から x87 が CPU 内に内蔵されるようになった。

さて、浮動小数点演算にはもう一本の歴史がある。 SIMD である。 CPU の処理能力が上がると、音声や動画のエンコード、デコードや、3D 処理などの処理能力がもっとほしい、という要望が出てきた。そうした声に答える形で AMD が 3DNow!という SIMD 命令拡張を発表した。これは Intel による MMX を拡張し、64 ビットの MMX レジスタを使って単精度実数の演算を二つ同時にできるようにしたものだ。その後、Intel も SSE という SIMD 命令拡張を作り、ここで 128 ビットである XMM レジスタが導入された。当初、XMM レジスタは単精度実数の演算しかできなかったが、SSE2 で倍精度実数を扱えるようになる。 AMD は倍精度実数の計算にデフォルトで XMM レジスタを使うようになり、その後 Intel もそうなったと思われる (そのあたりの前後関係はよく知らない)。

そんなわけで、x86 には浮動小数点レジスタとして、x87 で導入された 80 ビットのレジスタと、SSE で導入された 128 ビットの XMM レジスタがあるが、「普通の」64 ビットの浮動小数点レジスタはついに導入されなかった。現在の x86 では、SIMD ではない普段遣いの倍精度実数演算でも、SIMD レジスタである XMM レジスタの下位 64 ビットを使って演算する。

その後、SIMD 命令拡張は AVX、AVX2、AVX-512 と発展を続け、SIMD 幅も 256 ビット、512 ビットと増えていった。それに伴って SIMD レジスタも YMM、ZMM と拡張されていく。 YMM の下位 128 ビットが XMM、ZMM の下位 256 ビットが YMM になるのは、汎用レジスタ AX、EAX、RAX に包含関係があるのと同様である。

もう少し実戦的な SIMD 化

上記で触れた SIMD 化は非常に単純なもので、コンパイラが自動でできるものだった。しかし、一般には SIMD 化はレジスタ内をシャッフルしたり、データ構造を考えたりといろいろ複雑である。ここでは「もっとガチな SIMD 化をしたい！」というコアな SIMDer のために、もう少しだけ実践的な例を挙げてみよう。

磁場中の荷電粒子の運動を考える。磁場ベクトルを $\vec{B}$、速度ベクトルを $\vec{v}$、位置ベクトルを $\vec{r}$ とする。簡単のため、素電荷も質量も光速も 1 とする単位系を取ろう。運動方程式はこんな感じになる。

$$\dot{\vec{v}} = \vec{v} \times \vec{B}$$

$$\dot{\vec{r}} = \vec{v}$$

ここで、速度の時間微分を要素をあらわに書くとこうなる。

$$\dot{v_x} = v_y B_z - v_z B_y$$

$$\dot{v_y} = v_z B_x - v_x B_z$$

$$\dot{v_z} = v_x B_y - v_y B_x$$

磁場中の荷電粒子の運動は、磁場と平行な向きには等速直線運動、垂直な向きには円運動をするため、結果として螺旋を描いてすすむ。こんな感じ。

さて、適当な方向に向いた磁場中に、ランダムな向きに初速を持った荷電粒子たちをばらまいた系を計算してみよう。なお、粒子同士の相互作用も無視する。三次元シミュレーションなので、三次元ベクトルを構造体で表現する。

```
struct vec {
  double x, y, z;
};
```

粒子数を `N` とし、位置ベクトル、速度ベクトルを構造体の配列にとる。

```
const int N = 100000;
vec r[N], v[N];
```

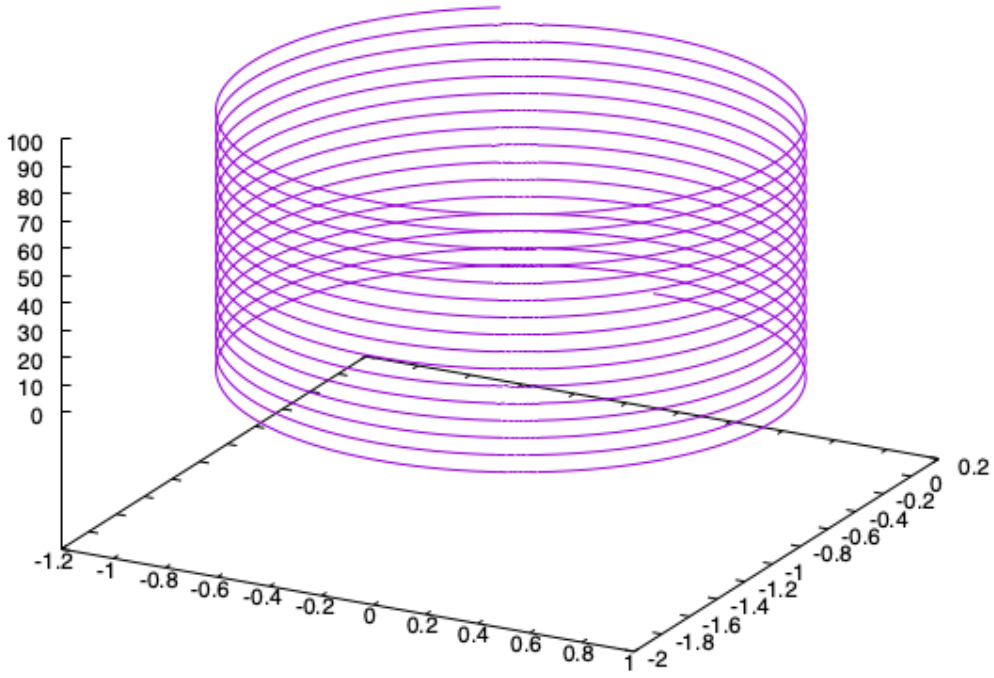

図 2: magnetic/one.png

この時、一次のオイラー法で時間発展を書くとこんな感じになる。

```
void calc_euler() {
  for (int i = 0; i < N; i++) {
    double px = v[i].y * BZ - v[i].z * BY;
    double py = v[i].z * BX - v[i].x * BZ;
    double pz = v[i].x * BY - v[i].y * BX;
    v[i].x += px * dt;
    v[i].y += py * dt;
    v[i].z += pz * dt;
    r[i].x = r[i].x + v[i].x * dt;
    r[i].y = r[i].y + v[i].y * dt;
    r[i].z = r[i].z + v[i].z * dt;
  }
}
```

しかし、よく知られているように一次のオイラー法は非常に精度が悪い。あなたが物理を学んだのなら、「磁場は荷電粒子に仕事をしない」ということを知っているはずである。つまり、全エネルギーは保存しなければならない。粒子間相互作用がないため、エネルギーは運動エネルギーだけである。

```
double energy(void) {
  double e = 0.0;
  for (int i = 0; i < N; i++) {
    e += v[i].x * v[i].x;
    e += v[i].y * v[i].y;
    e += v[i].z * v[i].z;
  }
  return e * 0.5 / static_cast<double>(N);
}
```

ちなみに、運動エネルギーなどは「一粒子あたり」の量にしておく (つまり平均エネルギーとする) と、粒子数の増減にエネルギーがよらなくなって便利である。時間発展はこんな感じにかけるだろう。

```
  init();
  double t = 0.0;
  for (int i = 0; i < 10000; i++) {
    calc_euler();
    t += dt;
    if ((i % 1000) == 0) {
      std::cout << t << " " << energy() << std::endl;
    }
  }
```

平均エネルギーの時間発展はこうなる。

1st-Euler とあるのが 1 次のオイラー法である。保存するべきエネルギーがどんどん増えてしまっていることがわかる。

さて、精度を上げる数値積分法はいくらでもあるが、ここでは簡単に二次の Runge-Kutta(RK) 法を採用しよう。

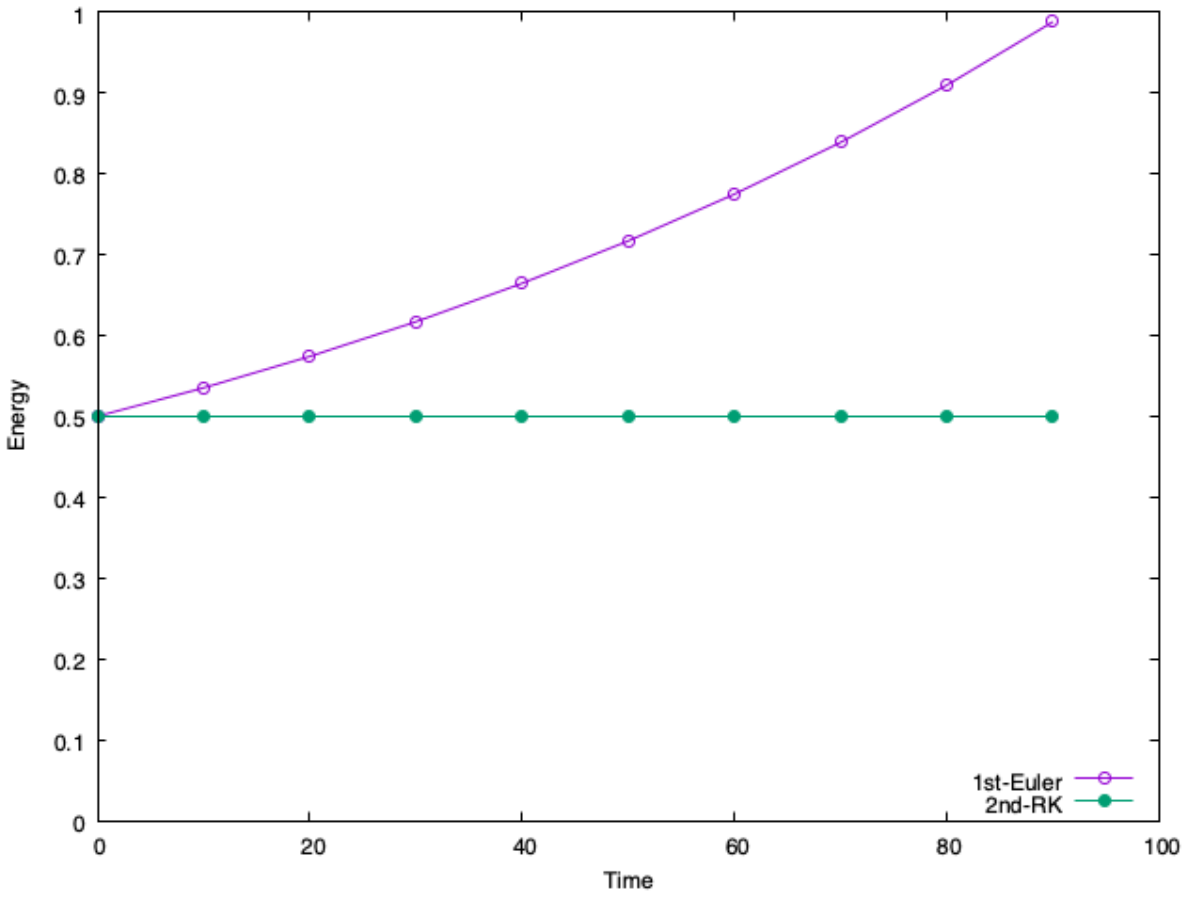

図 3: magnetic/energy.png

```
void calc_rk2() {
  for (int i = 0; i < N; i++) {
    double px = v[i].y * BZ - v[i].z * BY;
    double py = v[i].z * BX - v[i].x * BZ;
    double pz = v[i].x * BY - v[i].y * BX;
    double vcx = v[i].x + px * dt * 0.5;
    double vcy = v[i].y + py * dt * 0.5;
    double vcz = v[i].z + pz * dt * 0.5;
    double px2 = vcy * BZ - vcz * BY;
    double py2 = vcz * BX - vcx * BZ;
    double pz2 = vcx * BY - vcy * BX;
    v[i].x += px2 * dt;
    v[i].y += py2 * dt;
    v[i].z += pz2 * dt;
    r[i].x += v[i].x * dt;
    r[i].y += v[i].y * dt;
    r[i].z += v[i].z * dt;
  }
}
```

二次の RK は、まず 1 次のオイラー法で時間刻みの半分だけ系を仮想的に時間発展させ、その場所において再度時間微分を計算し、その微分係数をもとに現在時刻から時間発展させる方法である。一般に Runge-Kutta というと 4 次の方法を指すが、ここでは手抜きして 2 次にする。また、座標の更新はどうせ同じなので 1 次のオイラーのままにする。

こうして計算したエネルギーが先程の図の「2nd-RK」と記されたデータ点である。エネルギーがきっちり保存していることがわかるだろう。この関数 `calc_rk2` を SIMD 化して見ることにしよう。

先に、SIMD 化では、連続したデータを取ってくるのが重要であると書いた。いまは YMM レジスタを使うので、4 つの要素を取ってきたい。しかし、三次元シミュレーションなので、各粒子は 3 要素の速度と 3 要素の位置を持っている。まずこれをなんとかしよう。

具体的には、3 要素のベクトルを 4 要素にしてしまう。

```
struct vec {
  double x, y, z, w; // w を増やした
};
```

こうすると、一命令で粒子の速度がごそっとレジスタに乗る。

上図の例では、ポインタ`&v[i].x` が、i 番目の粒子の速度ベクトルの先頭位置を示すため、

```
// vv <- (w,z,y,x)
__m256d vv = _mm256_load_pd((double *)(&(v[i].x)));
```

とすると、レジスタに 3 つのデータを一度に持ってくることができる (一要素は無駄になる)。

次に、レジスタに載せたデータから微分を計算したいのだが、そのままではベクトル積の形にならない。具体的にやりたいのはこんな計算であった。

```
    double px = v[i].y * BZ - v[i].z * BY;
    double py = v[i].z * BX - v[i].x * BZ;
```

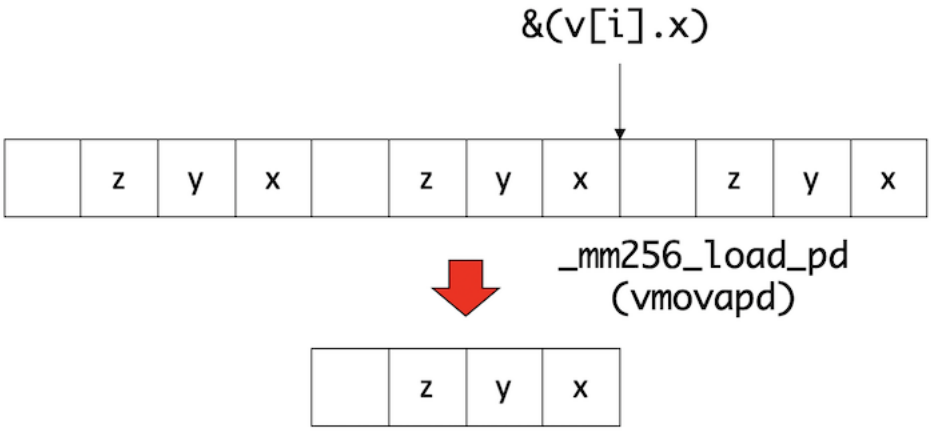

図 4: fig/load_pd.png

```
    double pz = v[i].x * BY - v[i].y * BX;
```

この計算をデータをレジスタに載せたままで実行するためには、レジスタ内で (x,y,z,w) と並んでいるデータを (y,z,x,w) という順番に並び替えなければならない。このようなレジスタ内の要素の並び替えをするためにシャッフル命令が用意されている。シャッフルのやり方については、例えば AVX の倍精度実数シャッフル系命令チートシートを参照してほしいが、シャッフル後の要素の並び方を四進数で表現してやる。たとえば (0,1,2,3) とあるレジスタを (1,2,0,3) の順番にしたければ、

```
const int im_yzx = 64 * 3 + 16 * 0 + 4 * 2 + 1 * 1;
```

という数字を引数に与えてやれば良い。上記の数を四進数表記すると 3021 になることに注意。これを

```
// (w,x,z,y) <- (w,z,y,x)
 __m256d vv_yzx = _mm256_permute4x64_pd(vv, im_yzx);
```

注：このあたり、左右どちらを下位に取るかいつも混乱する。レジスタは右に下位ビットを取るのが慣例であるため、その意味では (w,z,y,x) という順番で並んでいる。しかし、普通の数学の意味でのベクトルは最初の要素を左に書くのが慣例なので (x,y,z,w) と書きたくなる。そのあたり混乱しがちだが、適当に補って読んでほしい。

これによりレジスタの中身を適宜並び替えて、足したりかけたり引いたりすればよろしい。磁場については、使うベクトルが (z,x,y,0) と (z,x,y,0) のパターンしかないので、最初に宣言しておこう。

```
  __m256d vb_zxy = _mm256_set_pd(0.0, BY, BX, BZ);
  __m256d vb_yzx = _mm256_set_pd(0.0, BX, BZ, BY);
```

先程の図の中央に縦に並んだベクトルの計算は、実際のコードでは

```
__m256d vp = vv_yzx * vb_zxy - vv_zxy * vb_yzx;
```

と一行で書ける (SIMD がベクトル演算と呼ばれる所以である)。

これを使って中点の微分係数を求めるのはさほど難しくなく、中点の微分係数がもとまったら、それを使って速度を更新するのも難しくないだろう。

さて、速度ベクトルが、`__m256d vv` に保存されたとする。これはレジスタであるため、メモリに書き戻し

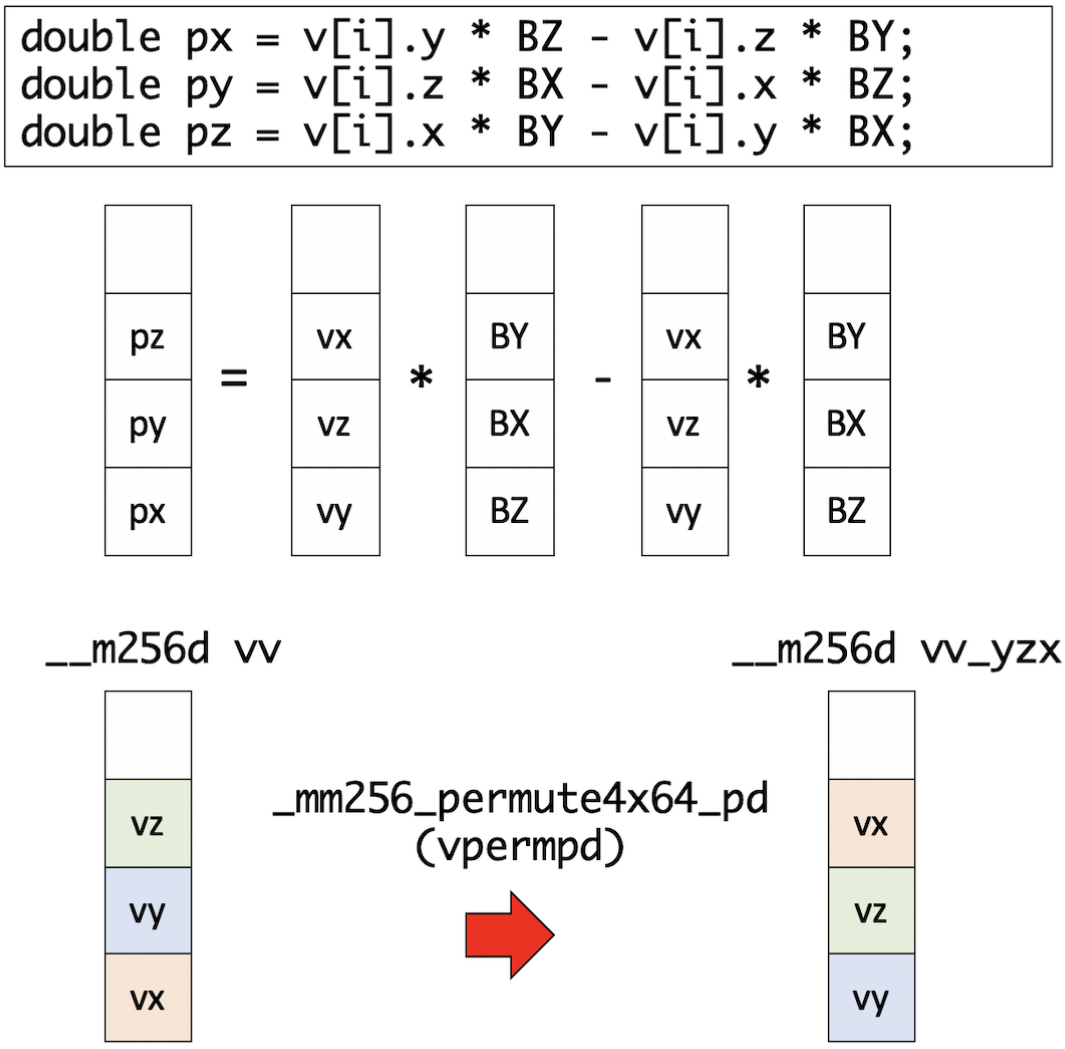

図 5: fig/permute.png

てやる必要がある。これはロードと同様にポインタ&v[i].x が指す場所にストアしてやればよろしい。

```
// (w,z,y,z) -> v[i]
_mm256_store_pd((double *)(&(v[i].x)), vv);
```

位置の更新も同様に、更新された速度を使って

```
    __m256d vr = _mm256_load_pd((double *)(&(r[i].x)));
    vr += vv * vdt;
    _mm256_store_pd((double *)(&(r[i].x)), vr);
```

とすれば良い。以上をまとめると、時間発展ルーチンの SIMD 版はこんな感じにかける。

```
void calc_rk2_simd() {
  __m256d vb_zxy = _mm256_set_pd(0.0, BY, BX, BZ);
  __m256d vb_yzx = _mm256_set_pd(0.0, BX, BZ, BY);
  __m256d vdt = _mm256_set_pd(0.0, dt, dt, dt);
  __m256d vdt_h = _mm256_set_pd(0.0, dt * 0.5, dt * 0.5, dt * 0.5);
  const int im_yzx = 64 * 3 + 16 * 0 + 4 * 2 + 1 * 1;
  const int im_zxy = 64 * 3 + 16 * 1 + 4 * 0 + 1 * 2;
  for (int i = 0; i < N; i++) {
    __m256d vv = _mm256_load_pd((double *)(&(v[i].x)));
    __m256d vr = _mm256_load_pd((double *)(&(r[i].x)));
    __m256d vv_yzx = _mm256_permute4x64_pd(vv, im_yzx);
    __m256d vv_zxy = _mm256_permute4x64_pd(vv, im_zxy);
    __m256d vp = vv_yzx * vb_zxy - vv_zxy * vb_yzx;
    __m256d vc = vv + vp * vdt_h;
    __m256d vp_yzx = _mm256_permute4x64_pd(vc, im_yzx);
    __m256d vp_zxy = _mm256_permute4x64_pd(vc, im_zxy);
    __m256d vp2 = vp_yzx * vb_zxy - vp_zxy * vb_yzx;
    vv += vp2 * vdt;
    vr += vv * vdt;
    _mm256_store_pd((double *)(&(v[i].x)), vv);
    _mm256_store_pd((double *)(&(r[i].x)), vr);
  }
}
```

19 行が 23 行になっただけなので、さほどややこしいことはないと思う。増えた 4 行も、磁場ベクトルを作るところとシャッフルのインデックスの準備だけである。いきなりこのルーチンを作るとバグが入りやすいが、シリアルコードに埋め込む形で SIMD 化をすすめ、各ステップごとにシリアルルーチンと SIMD レジスタの内容が一致することを確認しながらやればさほど難しいことはない。

さて、これは SIMD 化はしたが、演算順序その他は全く変えていないため、丸め誤差も含めて完全にシリアルコードと一致するはずである。それを確認するため、時間発展後に位置ベクトルのダンプを取ろう。

```
void dump() {
  for (int i = 0; i < N; i++) {
    std::cout << r[i].x << " ";
    std::cout << r[i].y << " ";
    std::cout << r[i].z << std::endl;
```

```
  }
}
```

時間発展後に座標をダンプして、その結果を比較しよう。シリアル版を mag.cpp、SIMD 版を mag_simd.cpp としておき、以下のようにコンパイル、実行、結果の比較をする。

```
$ g++ -std=c++11 -O3 -mavx2 -mfma mag.cpp -o a.out
$ g++ -std=c++11 -O3 -mavx2 -mfma mag_simd.cpp -o b.out
$ time ./a.out > a.txt
./a.out > a.txt  4.58s user 0.27s system 99% cpu 4.876 total

$ time ./b.out > b.txt
./b.out > b.txt  2.54s user 0.29s system 99% cpu 2.849 total

$ diff a.txt b.txt # 結果が一致
```

実行時間が 4.58s → 2.54s と高速化され、かつ実行結果が一致していることがわかる。

さて、この結果を見て「おお！まぁまぁSIMD 化の効果あるじゃん！」と思うのは早計である。先程のデータ構造は、構造体の配列の形になっていた。このような構造を **Array of Structure (AoS)** と呼ぶ。逆に、同じデータを、配列の構造体で表現することもできる。

```
double rx[N], ry[N], rz[N];
double vx[N], vy[N], vz[N];
```

ここでは配列を構造体にまとめていないが、このようなデータ構造を **Structure of Array (SoA)** と呼ぶ。 AoS と SoA、どちらが良いかは場合によるのだが、SIMD 化においては SoA にしたほうが性能がでる場合が多い。

先程の AoS のシリアルコードと全く同じ内容を、データ構造を SoA にして書いたものを mag_soa.cpp とする。例えば、時間発展ルーチンはこんな感じにかける。

```
void calc_rk2() {
  for (int i = 0; i < N; i++) {
    double px = vy[i] * BZ - vz[i] * BY;
    double py = vz[i] * BX - vx[i] * BZ;
    double pz = vx[i] * BY - vy[i] * BX;
    double vcx = vx[i] + px * dt * 0.5;
    double vcy = vy[i] + py * dt * 0.5;
    double vcz = vz[i] + pz * dt * 0.5;
    double px2 = vcy * BZ - vcz * BY;
    double py2 = vcz * BX - vcx * BZ;
    double pz2 = vcx * BY - vcy * BX;
    vx[i] += px2 * dt;
    vy[i] += py2 * dt;
    vz[i] += pz2 * dt;
    rx[i] = rx[i] + vx[i] * dt;
    ry[i] = ry[i] + vy[i] * dt;
    rz[i] = rz[i] + vz[i] * dt;
  }
}
```

v[i].x が vx[i] とかになっているだけで、そのまんまなのがわかるかと思う。これも実行して、結果を比較しよう。

```
$ g++ -std=c++11 -O3 -mavx2 -mfma mag_soa.cpp -o c.out
$ time ./c.out > c.txt
./c.out > c.txt  1.20s user 0.28s system 98% cpu 1.493 total

$ diff a.txt c.txt # 結果が一致
```

手で SIMD 化した場合に比べて、2.54s → 1.20s と倍以上高速化されたことがわかる。もともとのシリアルコードが 4.58s だったので、4 倍近い。つまり、このコードはコンパイラによる自動 SIMD 化により 4 倍早くなったわけで、理想的な SIMD 化ができたことがわかる。

実際にコンパイラはこんなコードを吐いている。

```
L10:
  vmovapd (%rdi,%rax), %ymm0
  vmovapd (%r9,%rax), %ymm11
  vmovapd (%r8,%rax), %ymm10
  vmulpd  %ymm7, %ymm0, %ymm2
  vmulpd  %ymm11, %ymm5, %ymm1
  vfmsub231pd %ymm6, %ymm0, %ymm1
  vmulpd  %ymm4, %ymm1, %ymm1
  vmulpd  %ymm10, %ymm6, %ymm3
  vfmadd132pd %ymm8, %ymm10, %ymm1
  vfmsub231pd %ymm11, %ymm7, %ymm3
  vfmsub231pd %ymm10, %ymm5, %ymm2
  vmulpd  %ymm4, %ymm3, %ymm3
  vmulpd  %ymm4, %ymm2, %ymm2
  vfmadd132pd %ymm8, %ymm0, %ymm3
  vfmadd132pd %ymm8, %ymm11, %ymm2
  vmulpd  %ymm6, %ymm1, %ymm9
  vfmsub231pd %ymm7, %ymm2, %ymm9
  vmulpd  %ymm5, %ymm2, %ymm2
  vfmadd132pd %ymm4, %ymm0, %ymm9
  vfmsub231pd %ymm6, %ymm3, %ymm2
  vmovapd %ymm9, (%rdi,%rax)
  vfmadd132pd %ymm4, %ymm10, %ymm2
  vfmadd213pd (%rsi,%rax), %ymm4, %ymm9
  vmovapd %ymm2, (%r8,%rax)
  vmulpd  %ymm7, %ymm3, %ymm0
  vfmadd213pd (%rdx,%rax), %ymm4, %ymm2
  vfmsub231pd %ymm5, %ymm1, %ymm0
  vmovapd %ymm9, (%rsi,%rax)
  vfmadd132pd %ymm4, %ymm11, %ymm0
  vmovapd %ymm2, (%rdx,%rax)
  vmovapd %ymm0, (%r9,%rax)
  vfmadd213pd (%rcx,%rax), %ymm4, %ymm0
  vmovapd %ymm0, (%rcx,%rax)
```

```
  addq  $32, %rax
  cmpq  $800000, %rax
  jne L10
```

最内ループだけ抜き出したが、基本的に ymm レジスタだらけであり、理想的に SIMD 化されていることがわかる。また、シャッフル命令も全く出ていないことがわかる。コンパイラはループを素直に 4 倍展開し、各レジスタに (x1, x2, x3, x4) のような形でデータを保持して計算している。ループカウンタは%rax で、毎回 32 ずつ増えており、800000 になったら終了なので、このループは 25000 回転することがわかる。

ちなみに、先程手で SIMD 化したループのアセンブリはこうなっている。

```
L13:
  vmovapd (%rax), %ymm2
  addq  $32, %rax
  addq  $32, %rdx
  vpermpd $201, %ymm2, %ymm0
  vpermpd $210, %ymm2, %ymm1
  vmulpd  %ymm3, %ymm1, %ymm1
  vfmsub132pd %ymm4, %ymm1, %ymm0
  vfmadd132pd %ymm6, %ymm2, %ymm0
  vpermpd $201, %ymm0, %ymm1
  vpermpd $210, %ymm0, %ymm0
  vmulpd  %ymm3, %ymm0, %ymm0
  vfmsub231pd %ymm4, %ymm1, %ymm0
  vfmadd132pd %ymm5, %ymm2, %ymm0
  vmovapd %ymm0, -32(%rax)
  vmovapd %ymm0, %ymm1
  vfmadd213pd -32(%rdx), %ymm5, %ymm1
  vmovapd %ymm1, -32(%rdx)
  cmpq  %rcx, %rax
  jne L13
```

vpermpd がシャッフル命令である。ループボディがかなり小さいが、このループは 100000 回まわるため、25000 回しかまわらないコンパイラによる自動 SIMD 化ルーチンには勝てない。大雑把な話、ループボディの計算コストが半分だが、回転数が 4 倍なので 2 倍負けた、という感じである。

上記の例のように、「いま手元にあるコード」をがんばって「そのまま SIMD 化」して高速化しても、データ構造を変えるとコンパイラがあっさり自動 SIMD 化できて負けることがある。多くの場合「SIMD 化」はデータ構造のグローバルな変更を伴う。先のコードの AoS 版である md.cpp と、SoA 版である md_soa.cpp は、全く同じことをしているが**全書き換え**になっている。今回はコードが短いから良いが、10 万行とかあるコードだと「やっぱり SoA の方が早いから全書き換えで！」と気軽には言えないだろう。また、デバイスによってデータ構造の向き不向きもある。例えば「CPU では AoS の方が早いが、GPGPU では SoA の方が早い」なんてこともざらにある。こういう場合には、ホットスポットルーチンに入る前に AoS ←→ SoA の相互変換をしたりすることも検討するが、もちろんその分オーバーヘッドもあるので面倒くさいところである。

まぁ、以上のようにいろいろ面倒なことを書いたが、ちゃんと手を動かして上記を試してみた方には「SIMD 化は (原理的には) 簡単だ」ということには同意してもらえると思う。 MPI も SIMD 化も同じである。いろいろ考えることがあって面倒だが、やること自体は単純なので難しくはない。今回はシャッフル命令を取り上げたが、他にもマスク処理や gather/scatter、pack/unpack など、SIMD には実に様々な命令がある。

しかし、「そういう命令欲しいな」と思って調べたらたいがいある。あとは対応する組み込み関数を呼べばよい。要するに「やるだけ」である。ただし、MPI 化は「やれば並列計算ができ、かつプロセスあたりの計算量を増やせばいくらでも並列化効率を上げられる」ことが期待されるのに対して、SIMD 化は「やっても性能が向上するかはわからず、下手に手を出すよりコンパイラに任せた方が早い」なんてこともある。全く SIMD 化されていないコードに対して SIMD 化で得られるゲインは、256bit なら 4 倍、512 ビットでも 8 倍程度しかなく、現実にはその半分も出れば御の字であろう。 SIMD 化はやってて楽しい作業であるが、手間とコストが釣り合うかどうかは微妙だな、というのが筆者の実感である。