SIMD 拡張命令を用いた N 体計算加速ライブラリ Phantom-GRAPE

似鳥　啓吾

2012 年 9 月 13 日

概要

重力多体問題の数値計算では、粒子間相互作用の計算が計算量の殆どを占めるため、この箇所を集中的にチューニングすることが効果的である。一方、最近のマイクロプロセッサでは SIMD 拡張命令を備えることでピーク性能を向上させるようになっている。ひとつの問題は、コンパイラベースの最適化では、C や FORTRAN で書かれた相互作用計算コードから SIMD 命令を十分に活用したピークに近い性能を得るようなコードを生成することは困難であるということである。しかし徹底的に高速化したい箇所は、高級言語で２０行程度のものであり、この箇所をインラインアセンブラなり組み込み関数なりで書いてしまえばそれで済む話であり、このような経緯で開発されたのが「Phantom-GRAPE」である。

キーワード：N 体計算；SIMD 拡張命令；SSE；AVX

1　重力相互作用の数値計算

いわゆる重力多体系の数値計算で殆どの計算時間を占めるのは、数式ではたったの一行のものである：

$$\ddot{\boldsymbol{x}}_i = \sum_{j \neq i}^{N} G m_j \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{(|\boldsymbol{x}_j - \boldsymbol{x}_i|^+ \varepsilon^2)^{3/2}}, \tag{1}$$

ここで $\boldsymbol{x}_i$、m_i がそれぞれ粒子 i の座標と質量、G が重力定数（数値計算では 1 になるように単位系を取ることが多い）、ε が発散を防ぐために設定するソフトニングパラメータであり、長さの次元を持つ。計算によってはこの時間微分も計算したり、ソフトニング長を粒子ペアに応じて可変にしたり、あるいは何かしらのカットオフ関数を掛けたりはするが、この式を時間積分することが重力多体問題の数値計算の全てであるといってしまってよい。

この式の左辺を N 個の粒子について評価するため、$O(N^2)$ の計算といわれるわけだが、$O(N \log N)$ の Barnes Hut ツリー法や時間方向に最適化された「独立時間刻み法」といった「賢い」方法であっても、肝心な相互作用計算は「N_i 個の粒子が N_J この粒子から受ける重力を足し合わせる」といった計算操作に落とせるため、式の形や二重ループの構造は殆ど変わらない。実際に C++ のループとして書いてみたものが、次のようなものである（後の精度の議論のために、`float` と `double` を混合させてある）。

リスト 1　C++ による重力計算の二重ループ

```
1 void evaluate_gravity(
2         const int ni,
3         const int nj,
```

```
4           const double posi[][3],
5           const double posj[][3],
6           const float mj[],
7           const float eps2,
8           double acci[][3])
9   {
10      for(int i=0; i<ni; i++){
11          const double xi = posi[i][0];
12          const double yi = posi[i][1];
13          const double zi = posi[i][2];
14          double ax = 0.0;
15          double ay = 0.0;
16          double az = 0.0;
17          for(int j=0; j<nj; j++){
18              const float dx = float(posj[j][0] - xi);
19              const float dy = float(posj[j][1] - yi);
20              const float dz = float(posj[j][2] - zi);
21              const float r2 = eps2 + dx*dx + dy*dy + dz*dz;
22              const float ri2 = 1.0f / r2;
23              const float mri1 = m[j] * sqrtf(ri2);
24              const float mri3 = mri1 * ri2;
25              ax += double(mri3 * dx);
26              ay += double(mri3 * dy);
27              az += double(mri3 * dz);
28          }
29          acc[i][0] = ax;
30          acc[i][1] = ay;
31          acc[i][2] = az;
32      }
33  }
```

このぐらいに書いておけば、コンパイラが「無駄なコード」を吐くことはまずない。コンパイラとオプションによっては、部分的に SIMD 命令を使ってくれることもあるだろうが、得られる性能は手動で（インラインアセンブラか組み込み関数で）SIMD 化したものに比べればかなり見劣りするものとなる。

コンパイラベースの SIMD 化が難しいのは、そもそもは高級言語での「型」と CPU が扱う「語」が、スカラー算のときは一致していたのが、CPU の側だけ SIMD 命令に行こうとしてももはや一致しないからである。このとき言語の側の「型」の方も拡張して SIMD 命令側に対応するものを用意したのがコンパイラ提供の「組み込み関数 (intrinsics)」であり、これを用いて並列度を明示的にに指定してプログラムすることで、実行効率の高いコードを作ることができる（ただしアーキテクチャ間の移植性は失われる）。この辺は後でもう少し議論することとして、これから x86 プロセッサにの SIMD 拡張の歴史を簡単に振り返ってみる。

2 x86 プロセッサの SIMD 拡張の歴史

もともと x86 プロセッサには x87 と呼ばれる浮動小数点命令がオプションで用意されており、これは 80-bit のスタック型のレジスタを 8 本備えたものでる。オプションということで、80386 の時代までは FPU をコプロセッサとして別のチップで買ってきて、ソケットに増設するといったスタイルだった。80486 以降は下位機

種で無効化されたのを除いては FPU は x 内蔵されるようになり、Pentium からは全てのモデルで標準搭載となっている。

この 80-bit のレジスタを流用（利用する際は排他利用）するかたちで、64-bit のフラットなレジスタを８本用いて 8/16/32-bit の整数演算の SIMD 実行をサポートしたのが MMX（公式ではないが MultiMedia eXtension とも）であり、後期の Pentium プロセッサから搭載された。レジスタ名には `mm0-mm7` が用いられている。

MMX では整数演算どまり（ビットマップ画像や PCM 音声にはこれで十分で、浮動小数点を用いるのはもったいなかった）だったが、x86 に浮動小数点の SIMD 拡張命令を最初に用いたのは AMD の 3DNow!であり、これは MM レジスタ上で２語の単精度浮動小数点の演算を同時に行うことができた。また、高速な逆数と逆数平方根の近似命令を備えており、近似の精度は 15-bit であったが、Newton 法によりフル単精度の 24-bit まで精度を改善する命令もあった（加減乗算だけでも可能だが、命令数やレジスタ消費数を削減できた)。3DNoW!は AMD の k6-2 から搭載され、１クロックに２回の単精度加減算と単精度乗算をそれぞれ実行できたため、クロックあたりのピークでは４演算、333 MHz の k6-2 では単精度ピーク 1.33 GFLOPS と、当時としては非常に高い性能を誇った。この時点で何かしらの N 体計算への応用が試みられていたとしても不思議ではないのだが、文献として残されているものは存在しないようである。

この少し後にでてきたのが Intel の SSE (Streaming Simd Extensions) であり、MMX や 3DNoW!との一番の違いは、新設された８本の XMM レジスタ (eXtended MultiMedia?) を備えていたことである。このため x87 の FPU レジスタとの排他利用ではなくなり、x87 命令と拡張命令が混合するケースでも問題なく使えるようになった。XMM レジスタは 128-bit 幅を持ち、Pentium III 以降でサポートされている SSE では単精度４語の、Pentium 4 以降でサポートされている SSE2 では倍精度２語や、MMX の幅をそのまま２倍にしたかたちでの整数演算を SIMD 実行できる。単精度の SSE では、逆数と逆数平方根の近似命令が 12-bit の半精度で提供されている。Newton 法を一度用いると、24-bit の単精度が得られるが、この反復用の命令は用意されていない。この後も細かな拡張が繰り返され、SSE4.2 までが番号が振られている。性能はというと、Pentium III では２クロックかけて加算と乗算を１命令ずつ実行ということで単精度が４演算でクロックあたりのピークは k6-2 と変わらず、Pentium 4 でも同様に単精度が４演算、倍精度が２演算となっている。SSE2 の命令が１クロックのスループットで実行できる、すなわち単精度８演算、倍精度４演算となったのは Intel では Core2 以降、AMD では Phenom ないし Barcelona 世代の Opteron 以降となる。

x86_64 になってから SSEx 命令の取り扱いが若干変わったことも触れておきたい。ひとつが XMM レジスタの本数が８本かあ１６本へと倍増したことで、このためある程度複雑な計算をしても LOAD/STORE 命令が多発することを避けられるようになった。もうひとつが SSE/SSE2 が浮動小数点命令の標準とされ x87 がレガシーとされたことで、このため x86_64 環境で普通に `float` や `double` を用いたコードをコンパイルすると SSE/SSE2 のスカラー命令（XMM レジスタの最下位の１語を用いる）が使用される。

SSEx の更なる拡張として、Sandy Bridge 世代の Core i3/i5/i7 プロセッサからは AVX (Advanced Vector eXtensions) が搭載された。これは XMM レジスタの幅を倍の 256-bit に拡張した YMM レジスタ上で単精度８語ないし倍精度４語を扱うというもので、安易にはみえるがピーク性能は倍になる。また、RISC 風の非破壊形３オペランド形式になったため、レジスタ数の圧迫が若干軽減され、また出力されたアセンブリコードも読みやすくなった。

さらには、近々発売予定の Intel MIC (Many Integrated Core) ないし Xeon Phi では、512-bit の ZMM レジスタを、本数も倍増された 32 本搭載する。

また、理論ピーク性能を倍増させる方法として、RISC 系の CPU では比較的昔から使われてきた FMA

(Fused Multiply-Add) も x86 CPU に取り入れられつつある。これは $A \times B + C$ のような演算をひとつの命令で行ってしまうもので、これで 2 演算と数えられる他、精度上の利点もある。ただし、行列乗算や多項式の計算では FMA の恩恵を受けられるが、単独の加減算や乗算が必要な局面では 1 命令あたりの演算数が半分になってしまうという点で注意が必要である。FMA は AMD の Bulldozer アーキテクチャに搭載され販売されている他、将来 Intel の Haswell アーキテクチャには 256-bit FMA がコアあたりふたつ搭載される（倍精度でクロックあたり 1 6 演算）ことが近日発表された。

さて、ここまでピーク性能を倍々に増やしてきた（増やしていく）話ばかりを書いてきたが、多くの読者の興味はアプリケーションの実効性能にあるだろう。主記憶帯域が律速となる計算では、SIMD 拡張命令の恩恵は常に小さく、コンパイラが出力してくれればそれを使う、程度でよい。行列乗算や N 体計算では、主記憶帯域の影響は小さく（あるいは実装の工夫で小さくできる）SIMD 命令が利用できるかどうかで性能に大きな差が出る。特に N 体計算では、現行の AVX で倍精度で計算を行うとすると、SIMD 化さていないと性能で 4 倍損をする、と考えておけばよいだろう。他にも、「キャッシュに計算の自由度がひととおり入り切るけれども、非常に多くのステップ数を要する計算」というカテゴリでも、ホットスポットをなんらかの手段で SIMD 化することは必須であるといってもいいだろう。

3 SIMD 化プログラミングの考え方と書き方

前の方でも、高級言語（ここでは C/C++、FORTRAN のこと）からコンパイラによって SIMD 命令を利用するのには多くの制限があると書いたが、ここでは、そもそも CPU とはどのように動作し、それに対して高級言語とはどのような考えで設計されているのか、という観点で困難さの由来をみてゆきたい。

そもそも CPU は「語」という単位でデータを取り扱う（普遍的な定義である自信はないが、この文書内では以下の定義で話を進める）。語の長さは 32-bit だったり 64-bit だったりするし、しばしば互換性や文字列処理のためにもっと短い語もサポートする。整数の語としては 8/16/32/64-bit が扱えて、浮動小数点の語としては 32/64-bit が扱えるものが今日では一般的である。レジスタの幅がその CPU の扱う語長であり、またこの語長を単位にメモリとレジスタの間でデータを行き来させる。このとき原則として、メモリのアドレスは語長の倍数に整列されている。異なる長さの語の間での演算は直接はサポートされていないが、変換命令は存在する。例として、x86_64 の一番目の汎用レジスタは 64-bit 長の rax であるが、その下位 32-bit は eax という別名を、さらにその下位 16-bit は ax という別名を、さらにその下位 8-bit と上位 8-bit はそれぞれ al と ah という別名を持っている。

実のところ、これは C 言語の「型」というシステムにそっくりそのまま対応する。崇高な型理論からすればこれはもう石器時代の遺物なのかもしれないが、C 言語の型とは結局、そもマシンで扱うデータ形式そのものということである。なのでその機械で扱える型でプログラムを書けば、比較的簡単なコンパイラで効率よく、高級言語の記述と対応の良い、アセンブリコードを得ることができる。しかし C 言語で書くことでアセンブリ言語で書くよりははるかに多くの恩恵を受けることができる。ある程度の移植性は確保できるし、変数には名前でアクセスでき、ある程度複雑な式も一行で書くことができ、構造化された制御構文を用いることができる。また、コンパイラの局所的な最適化も期待でき、結果が使われない計算の削除、重複した計算の削除、レジスタ割り付け、命令発行順序の変更等がある（この文脈では自動で SIMD 命令にしてくれるといった「高度な」最適化は指していない）。

一方、CPU の側の SIMD 拡張命令はどのようなものかというと、これは語長を倍々にしているようなものであり、ゆえにコンパイルする側からするとこれに応じて新しい型を言語拡張として追加していくというのが

最も自然な対処のしかたである。ただし CPU の側では同じ 128-bit の語であっても、C 言語の側では「単精度 4 語の型」と「倍精度 2 語の型」は区別され、ダイレクトキャスト又は別途用意された読み替え用の組み込み関数で相互変換できたりする。標準の演算子ですべての命令をカバーすることはできないので、個々の命令に対応する組み込み関数を用意する（あるいは演算子はサポートされていいないこともある)。このように C 言語の標準に対してコンパイラ拡張として SIMD 用の幅の広いデータ型、追加の演算子定義と組み込み関数を追加したものを「SIMD intrinsics」ないし「SIMD 拡張組み込み関数一式」のようにいったりする。これを用いてコードを書くと、128-bit/256-bit の変数はレジスタや 16/32-byte 境界に整列されたメモリ上に置かれ、これらの拡張型へのポインタも境界に整列されたアドレスを持つものとして取り扱われる。

さて、ある程度低レベルで考えてプログラミングすることに慣れていれば、コンパイラの拡張型と組み込み関数で何かしらのコードを書き始めることの敷居はさほど高くない。しかし実際に解きたい問題にこれを適用するには、いくつかの困難がある。問題の持つ並列度をどのように SIMD 型に持ってくるのかを考えなければならないのはもちろん、その結果データ構造も SIMD 命令に合わせたものにする必要が出てくる。また、aligned access だけではベクトルの要素間で垂直方向（同じ添字）でしか相互作用ができないということになり、水平方向に要素をやりとりするには、ある程度のオーバーヘッドを覚悟で unaligned access の命令を明示的に使ったり、レジスタ間で shuffle 命令を用いることになるのだが、この箇所は技巧的になりやすい。

簡単な重力相互作用の計算を SSE（`float` 4 語）と GCC 拡張で書いてみた例を以下に示す。

リスト 2　GCC 拡張による SSE を用いた重力相互作用計算関数

```
1  void evaluate_gravity_SSE(
2          const int ni,
3          const int nj,
4          const float xi[],
5          const float yi[],
6          const float zi[],
7          const float xmj[][4], // {x, y, z, m}[nj]
8          const float eps2,
9          float axi[],
10         float ayi[],
11         float azi[],
12         float poti[])
13 {
14     typedef float v4sf __attribute__((vector_size(16)));
15
16     assert(ni % 4 == 0);
17     assert((unsigned long)xi % 16 == 0);
18     assert((unsigned long)yi % 16 == 0);
19     assert((unsigned long)zi % 16 == 0);
20     assert((unsigned long)xmj % 16 == 0);
21     assert((unsigned long)axi % 16 == 0);
22     assert((unsigned long)ayi % 16 == 0);
23     assert((unsigned long)azi % 16 == 0);
24     assert((unsigned long)poti % 16 == 0);
25
26     const v4sf veps2 = {eps2, eps2, eps2, eps2};
27
```

```
28      for(int i=0; i<ni; i+=4){
29          const v4sf x = *(const v4sf *)(xi + i);
30          const v4sf y = *(const v4sf *)(yi + i);
31          const v4sf z = *(const v4sf *)(zi + i);
32          v4sf ax = {0.0, 0.0, 0.0, 0.0};
33          v4sf ay = {0.0, 0.0, 0.0, 0.0};
34          v4sf az = {0.0, 0.0, 0.0, 0.0};
35          v4sf pot = {0.0, 0.0, 0.0, 0.0};
36
37          for(int j=0; j<nj; j++){
38              const v4sf jp = *(const v4sf *)xmj[j];
39  #define SHUFPS __builtin_ia32_shufps
40              const v4sf xj = SHUFPS(jp, jp, 0x00); // {x, x, x, x}
41              const v4sf yj = SHUFPS(jp, jp, 0x55); // {y, y, y, y}
42              const v4sf zj = SHUFPS(jp, jp, 0xaa); // {z, z, z, z}
43              const v4sf mj = SHUFPS(jp, jp, 0xff); // {m, m, m, m}
44  #undef SHUFPS
45              const v4sf dx = xj - x;
46              const v4sf dy = yj - y;
47              const v4sf dz = zj - z;
48              const v4sf r2 = veps2 + dx*dx + dy*dy + dz*dz;
49              const v4sf ri1 = __builtin_ia32_rsqrtps(r2); // approx. r2^(-1/2)
50              const v4sf mri1 = mj * ri1;
51              const v4sf ri2 = ri * ri;
52              const v4sf mri3 = mri1 * ri2;
53
54              ax += mri3 * dx;
55              ay += mri3 * dy;
56              az += mri3 * dz;
57              pot -= mri1;
58          }
59          *(v4sf *)(axi + i) = ax;
60          *(v4sf *)(ayi + i) = ay;
61          *(v4sf *)(azi + i) = az;
62          *(v4sf *)(poti + i) = pot;
63      }
64  }
```

ここでは v4sf が float 4 語に相当する SSE のデータ型である。いくつかの__builtin_ia32_xxx や汚いポインタの読み替えには驚かされるかもしれないが、ループの内容そのものはリスト 1 と殆ど変わっていない。しかしここで重要なのは、引数側の配列をこの計算ループに適合するように合わせてあるということである。N 体計算のカーネル部分には、この計算用の配列を作っておき計算コードから必要なときにコピーするというアプローチが有効である。応用によっては、メモリ消費やコピーのコストが無視できなくなってしまうかもしれない。

4 To be continued...