My Document

Anonymous Anteater

2024年2月5日

概 要

1 Introduction

1.1 Background

In astrophysics and astronomy, to numerically calculate the dynamical evolution of N particles interacting gravitationally, N-body simulations are required. Figure 1 shows the equation for interparticle interactions in N-body simulations. If the equation is naively computed, the time complexity of calculation of interparticle interactions is $O(N^2)$, where N is the number of particles. Therefore, parallelization is required to speed up numerical simulations. To write a parallelized code for a numerical simulation, a user needs to understand the architecture of computer systems in detail. If a parallelized code is automatically generated by only describing the formulas and data of the numerical simulation, the above problems are solved. To realize the parallelization, we will develop Domain Specific Language DSL), which is a programming language specialized to a domain, for example, SQL and HTML.

1.2 parallelization

SIMD is one of the categories in Flynn's taxonomy 1, related to computer arithmetic processing. Using SIMD, arithmetic operations can be applied simultaneously to multiple data. In this paper, we compute and parallelize multiple particles. For example, as shown in Figure 1, the computation of acceleration for different particles is independent of each other, allowing the process to be parallelized. As a result, for instance Assuming that the variables for particles are double-precision floating-point numbers, we can execute an operation on four elements

simultaneously using the AVX2 instruction, it is possible to perform operations on four elements. Therefore, compared to non-parallelized code, there is a potential to accelerate the computation by four times. We will describe the design of a DSL in the next section to generate such parallelized code from the formulation of particle-particle interactions.

1.3 Overview of DSL

私たちは Pyker という DSL を作成しました。この言語の処理系は粒子相互作用の式を記述したコードを読み込み、その計算を高速化する C++言語のコードを生成することが出来る。この言語は変数を定義と相互作用の式を記述できる。変数を定義する部分では、粒子の質量や位置、結果を保存する変数、それ以外に必要となる変数を定義できる。式の部分では、相互作用の式に必要な四則演算や sqrt, べき乗, ベクトルの内積を計算できるようにした。この記述から kernel 関数を生成し、それを呼び出して使う。

2 Construction

私たちは DSL の実装に Python の代数計算ライブラリの Sympy を使った。Sympy では、数式をシンボルと演算子を用いて構文木として扱うことが出来る。そしてその構文木から実行可能なコードを生成するモジュールがある。この DSL では、それらの機能を用いて数式を読み込み、並列に実行できるコードを生成する。並列化には、Advanced Vector Extensions 2(AVX2) という命令セットを用いた。AVX2 とは、インテル社の CPU に実装された拡張命令セットの一つである。次のセクションでは、どのように AVX2 で並列化を行うかを説明する。

2.1 AVX2 for parallelization

SIMD を行うには通常の計算と同じように、まずメモリからデータを load し、そのデータをもとに計算し、結果を store する。しかし、SIMD の場合は、SIMD 用のレジスタを使って計算をするため load や計算、store の仕方が違う.load をするとき, SIMD 用のレジスタに同じ命令を行う複数の要素を格納する。そしてそれを使って同時に計算し、計算結果をそれぞれ store する。

例えば、倍精度小数の配列 x,y の要素を計算する C=X + Y のような計算を考えるとき,AVX2 を使用して SIMD を行う場合レジスタは 256 ビットのレジスタなので、x と y の要素をそれぞれ 4 つずつ load する。そしてそれを同時に足し、結果を格納する配列 C に store する。SIMD レジスタに load をするとき、1 次元配列なら連続するメモリをアクセスすればいいが、Figure a. に示す Array of Structures(AoS) のデータ構造の場合、要素ごとにロードしてこなくてはいけない。

例えば、粒子の 3 次元の座標を倍精度小数で pos [n] [3](n は粒子の総数とし、pos [i][0],pos[i][1], pos [i][2] をそれぞれ i 番目の粒子の x 座標,y 座標,z 座標) とするとき、粒子 i, j の x 座標の差を求めるとき、dx = pos [i][0]- pos[j][0] のようになるが, これを SIMD で並列化するとき、

```
dx0 = pos[i][0] - pos[j][0]
dx1 = pos[i + 1][0] - pos[j][0]
dx2 = pos[i + 2][0] - pos[j][0]
dx3 = pos[i + 3][0] - pos[j][0]
```

としたいので、(pos[j][0], pos[j][0], pos[j][0], pos[j][0), (,(dx0,dx1,dx2,dx3) とそれぞれパックして、同時に引き算を行う。しかし、SIMD レジスタにデータを一度にそれぞれ乗せなくてはいけないので、gather 命令を使い粒子 i のデータを load する。gather 命令とは、アドレスを指定してメモリアドレス上の連続していないデータ要素を読み込むのに使用する。gather 命令の説明を図 x に示す。このように load したデータを SIMD の命令を実行し同時に計算する。

計算結果を store する際もアドレスが連続していない場合がある。その時は、scatter 命令と呼ばれる gather 命令と対象になる命令があるが、AVX2 にはサポートされていないため、それぞれポインタを使ってデータを store する。このようにして SIMD による並列化をする。

また、これらの命令は instrinsincs 関数を使うことで C,C++などの**high-level programming language において**明示的に扱える。C,C++において instrinsincs 関数を扱うには、言語標準で使用できる immintrin.h というファイルをインクルードする。そしてデータは template.cpp を x 座標の差を計算する C++のコードを例として示す。Pyker は記述された数式を template.cpp のように SIMD 命令を実行できるような C++のコードを生成する。次のセクションでは、Pyker の説明する。

2.2 Pyker

この DSL では、変数の定義と相互作用の式の定義の 2 つを記述しそれをコンパイルすることでコードを生成する。最初に変数定義について説明する。

この DSL では、変数には class があり、EPI,EPJ,FORCE, その他と分かれる。EPI は相互作用を受ける粒子を示し、EPJ が相互作用を与える粒子になる。FORCE は相互作用の計算結果を保持し、そのほかの変数はソフトニングパラメータ等である。これらの変数を使って計算をするには、C++では、template.cpp のようになる。

Listing 1: template.cpp

```
78  int kernel(EPI epi, EPJ epj, FORCE force, double eps2, int n){
79    // (1). preprocess
80    for(int i = 0;i < n;i += 1) {
81      // (2). load EPI
82      // (3). Initialization of tmporary force
83      for(int j = 0;j < n;j += 1) {
84          // (4). calculate interparticle interactions
85      }
86      // (5). Store calculation result in the FORCE
87    }
88
89  }
90
```

91 (1) では、計算に必要な変数の定義などの前準備。(2) では、EPI の変数をロードする。(3) では、相互作用の計
92 算結果を一時的に保持する変数を初期化。(4) では、相互作用の計算を行い、結果を force の一次変数に保存。(5)
93 で，FORCE の変数に結果を store する。

94 このコードを生成するためには、変数の定義では、変数の class と、型、次元の情報がそれぞれ必要になる。そ
95 のため pyker では、変数の定義を正規表現で表すと以下のようになる。

96 ((EPI|EPJ|FORCE).(\w+))? (vec3<)?(F64)>? \w+

97 1 列目は class であれば、class の名前とそのメンバ変数名を書く。2 列目はベクトルの次元があれば、明示的に
98 書き、型は'F64' と書き、64bit 浮動小数を表す。3 列目では、pykg 上で扱う変数の名前になる。例えば、"EPI.pos
99 vec3¡F64¿ xi "は、EPI の pos というメンバ変数で、3 次元の 64bit 浮動小数型の xi という変数名となる。

100 相互作用の式の定義は、定義した変数を使って、FORCE の変数に結果をアキュムレートしていくように記述す
101 る。数式の記述に必要な一次変数はそれまでに定義された変数を用いてを新たに定義できる。使用できる演算は、
102 四則演算と sqrt とべき乗を表す'**' を使うことが出来る。結果は FORCE の変数に'=' で store することを表す。

103 例えば、Figure 1 の重力相互作用の式を計算するコードを生成する場合、この DSL で記述すると template.pyker
104 となる。

Listing 2: hoge

```
105
106  EPI.pos vec3<F64> xi
```

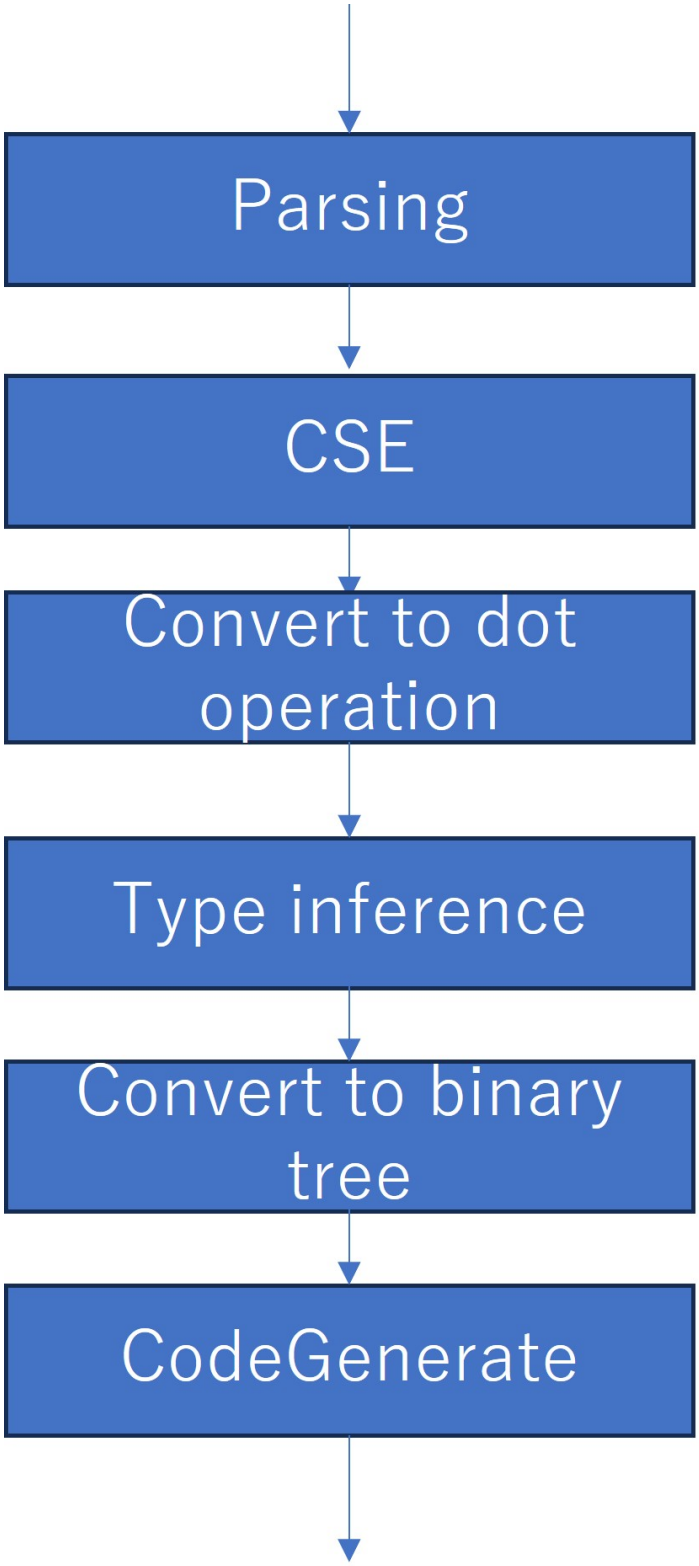
```
EPJ.pos vec3<F64> xj
EPJ.m F64 mass
FORCE.acc vec3<F64> ai
F64 eps2
F64 g
dr = xj - xi
ai = g * mass *  dr / sqrt(dr ** 2 + eps2) ** 3

```

次のセクションでは、どのようにして、このコードから目標のコードを生成するかを説明する。

2.3 Implementation

コードを生成する流れは以下の flowchart に示す。Parse では、DSL のコードから定義された変数の情報を変数名を key に持つハッシュに変換する。数式は、Sympy 形式の構文木に変換し、そのリストを取得する。次に構文木に変換した数式に対して Common subexpression elimination(CSE) を行う。CSE とは、数式に含まれる共通部分式をあらかじめ計算し、その計算結果を使って数式を計算することによって演算回数を減らすことである。type inference では、数式の一次変数の型推論を行う。その後数式を SIMD 演算に合わせるため構文木をすべて二分木にする。最後に以上の処理を行った構文木を辿り、相互作用の計算を行うコードを生成する。以下は、Parse から CSE,type inference, convert to binary tree, Codegenerate の順にどのように実装を行ったかを説明していく。



2.3.1 Parsing

Parse では、変数定義と数式で処理を分ける。変数定義では,class, ベクトル、型、変数名を持つ自作した Var というクラスのオブジェクトをそれぞれ生成する。そのオブジェクトと key を変数名に持つハッシュを作る (name_variable_map)。数式は Sympy のライブラリにあるメソッドの sympify() を使い、文字列として読み込んだコードを数式の構文木

に変換し、それらの構文木のリスト (expr_list) を取得する。次に CSE を行う。

2.3.2 CSE

CSE とは数式にある共通する部分式を先に計算し、その結果を使うことで演算回数を減らすこと。これには sympy にある cse() という関数に数式を渡して行う。これにより新たに部分式の数式が生成されて、その部分式の結果を使った計算式に変換される。

2.3.3 translate to dot method

この DSL では、ベクトル同士の積を内積として扱う。そのために、構文木をたどっていき該当する部分式を見つけてそこを置き換えて木を再構築する。sympy の構文木の構造は演算子が親となり、その子としてほかの演算子やシンボルがある。例えば、x,y,z を変数として x + y * z は図 1 のような構文木になる。しかしこの要素らがベクトルであるかが sympy が知らないため例えば、v1,v2 をベクトル、v3 をスカラーととこの DSL 上で定義されていたとすると、(v1 * v2) * v3 は一つの Mul が親となり、v1,v2,v3 が子となる構文木になる。そのため、完全二分木にする際、どこが内積に該当するのかわからない。なので内積を dot 関数に置き換える。sympy には FunctionCall という class があり、独自に定義した関数を構文木に組み込むことが出来る。そのため、構文木をたどり該当する部分木を FunctionCall で dot 関数に置き換えていく。その構文木をたどる際、ノードが部分式の場合、その部分式が返すのがベクトルかどうかわからない。そのため key にノードを持ち value にベクトルかスカラーかの情報を持つ hash(arg_ret_map) を作り、それを使って判定する。そして、dot に置き換え更新した部分式は arg_ret_map にその都度更新する。こうして dot 演算に置き換え再構築した構文木のリストを次の処理に渡す。

2.3.4 type inference

一時変数は数式の結果を一時的に保存する。その一時変数の型やベクトルを判断するには、数式の結果がどんな型なのかどんな vec なのかでわかる。またこの DSL では型は倍精度少数しか実装していないので、ベクトルかどうかだけが分かればいい。そのため、CSE のセクションで説明した arg_ret_map を使う。

2.3.5 convert to binary trees

simd 演算は instrcin 関数で行う際 2 項演算なので引数が 2 つの関数を使って数式を計算するコードにしなければならない。そのため、数式の構文木を完全二分木である必要がある。しかし、sympy の数式の構文木では演算の順序を考えなくてよい場合、一つの演算子を親としてその項をまとめる。そのため、子を 3 つ以上持つような演

算子のノードでは、その演算子で二分木に変換する。例を図 1 に示す。このとき sympy が自動で簡約化して二分木をまとめてしまわないように、演算子に置き換えるときに演算子の関数を使い、オプションで evaluate=False にする。例えば a * b * c は Mul(a, Mul(b, c, evaluate=False), evaluate=False) となる。

2.3.6 code generation of SIMD

CodeGen では、c++で実行可能な関数のコードを生成する。生成するコードの疑似コードを template.cpp に示す。

Listing 3: template.cpp

```
//(1) include part
#include<math>
#include<immintrin.h>
int kernel( (2) double xi[][3], double xj[][3], double eps2, double ai[][3], ...) {
  //(3) decleare tmp variable
  int i;
  int j;
  __m256d xi_tmp_v0;
  __m256d xi_tmp_v1;
  __m256d xi_tmp_v2;
  __m256d eps2_tmp;
  // ...
  // ...
  // ...
  //(4) def not class variables
  eps2_tmp = _mm256_set1_pd(eps2);
  // ...
  // ...
  // ...

  // (5) loop increment
```

```
182     for(i = 0;i < (n);i += 4) {
183       // (6) Load EPI variable part
184       int index_gather_0[4] = {0, 3, 6, 9};
185       __m128i vindex_gather_0 = _mm_load_si128((const __m128i*)index_gather_0);
186       xi_tmp_v0 = _mm256_i32gather_pd(&xi[i][0], vindex_gather_0, 8);
187       // ...
188       // ...
189       // ...
190       // (7) initialize result temporary variable
191       ai_tmp_v0 = _mm256_set1_pd(0.0);
192       ai_tmp_v1 = _mm256_set1_pd(0.0);
193       ai_tmp_v2 = _mm256_set1_pd(0.0);
194
195       for(j = 0;j < (n);j += 1) {
196         // (8) Load EPJ variable part
197           xj_tmp_v0 = _mm256_set1_pd(xj[j][0]);
198           xj_tmp_v1 = _mm256_set1_pd(xj[j][1]);
199           xj_tmp_v2 = _mm256_set1_pd(xj[j][2]);
200          // ...
201          // ...
202          // ...
203
204
205         // (9) calculate interparticle
206       }
207
208       // (10) Store result part
209
210     }
```

```
    return 0;
  }
```

コード生成の際に必要な要素を template.cpp に書かれた (1), (2),(3) の順に説明する。それぞれは (1) インクルード文の挿入、(2) 関数の引数の定義、(3) 相互作用の計算に必要な一時変数,(4)EPI,EPJ,FORCE でない変数の定義、(5) ループのインクリメント幅、(6)EPI の変数を load する部分、(7) 結果を一時的に保持する変数の初期化、(8) EPJ の変数をロードする部分、(9) 実際に計算する内容、(10)SIMD の store 部分である。また DSL 上で定義した EPI,EPJ,FORCE の変数は、ソースコード上では単なる変数として扱われる。

(1) では、SIMD 演算などで必要となるインクルード文を挿入する。math.h は、SIMD 化しない場合に、sqrt() を実行するために、immintrin.h では、SIMD 演算のための関数を呼び出すために必要となる.

(2) では、DSL のコードで変数定義されていた変数を引数とする。そのために変数をすべてみて、DSL のコードで変数定義されていれば引数に追加する。

(3)Sympy では、変数の定義を Variable というクラスのオブジェクトを Declaration というクラスのコンストラクタに渡して生成したオブジェクトで表す。

一時変数の名前はベクトルであれば、(DSL 上で記述した名前)_tmp_vi(i は次元数に合わせて 0 インデックスで定義), スカラーであれば、(DSL 上で記述した名前) _tmp とする。変数の型は今回は倍精度少数だけなので、__m256d にする。__m256d は AVX2 の 256bit のレジスタで要素が倍精度少数をセットできる型である.

(4) では、class でない変数の定義では、定義とともにその変数の値を並列する数分初期化する。それには,_mm256_set1_pd() という関数を使う。この関数は引数で与えられた倍精度少数の値を__m256d に 4 つセットする関数で、eps2_tmp のデータ構造は (eps2, eps2, eps2, eps2) のようになっている。これとループで使用するインデックスである,i と j を定義する.

(5) では、並列数分だけインクリメントするようにする。今回は倍精度少数しか実装していないので並列数は常に 4 にする.

(6) では、EPI の変数の load を行う。load には、AVX2 for Parallelization のセクションで説明したように gather 命令を使用する。gather 命令は、(6) に書かれている 3 行のコードは gather 命令による load の一連の流れである。index_gather_0[] は& xi[i][0] から何個分のメモリにあるかを示している。&xi[i][0] から&xi[i + 1][0] は double 型の変数の 3 つ分先のメモリアドレスになる。&xi[i + 2][0] も同様になるので index_gather_0[] には、0, 3, 6,9 となる。vindex_gather_0 は index_gather_0 を 128 ビットレジスタに乗せていて、これで_mm256_i32gather_pd を使うことが出来る。_mm256_i32gather_pd の第 3 引数の 8 は、要素が何バイトであるかを表し、double 型なので 8byte としている。これにより、&xi[i][0],&xi[i + 1][0],&xi[i + 2][0],&xi[i + 3][0] をロードする。また EPI の要素がスカ

241 ラーであれば、_mm256_load_pd(&m[i]) という関数を使い、連続する m[i],m[i+1],m[i+2],m[i+3] を 256 ビットの

242 レジスタに load する。これを EPI の変数すべてに行う。