Domain Specific Language for high performance computing

Ryuki Hiwada s1280076 Supervised by Naohito Nakasato

Abstract

1 Introduction

1.1 Background

宇宙物理学や流体力学等、様々な分野で粒子系シミュレーションが行われている. 例えば、惑星間の重力による力学的進化をシミュレーションしたN体シミュレーションや水の流体運動のモデル化がある。これらのシミュレーションには粒子の数に対して計算量が膨大になる。例としてN体シミュレーションの場合を説明する。N体シミュレーションでは, ほとんどの計算時間を重力相互作用の計算が占めている。(1)は、重力相互作用の式である.

$$a_i = \sum_{j \neq i}^{N} G m_j \frac{\mathbf{x}_j - \mathbf{x}_i}{\left(\left|\mathbf{x}_j - \mathbf{x}_i\right| + \epsilon^2\right)^{3/2}} \tag{1}$$

ここで a_i, $\mathbf{x}_i$ はそれぞれ粒子 i の加速度と位置を表し、m_j,$\mathbf{x}_j$ はそれぞれ粒子 j の質量と位置を表し、N,G,ϵ はそれぞれ粒子の数、重力定数発散を防ぐためのソフトニングパラメータである。(1) から計算量は粒子数 N に比例して、$O(N^2)$ となる。N体シミュレーションでは、粒子の数が膨大になる場合が多い。例えば、球状星団という天体には星が約 10^4 10^5 個存在する。このため非常に長い計算時間がかかるため高速化が求めれる。粒子間相互作用の計算は、粒子同士の計算が独立しているためデータ並列性が高い。そのため、単一の命令に対して複数のデータを処理する SIMD による並列に適している。SIMD による並列計算の例として、小池らが行った GRAPE-DR [1] による重力多体問題シミュレーションがある。[2] しかし、SIMD 計算を行うには, コンピュータのアーキテクチャを考慮したコードを記述しなくてはならない。

この研究の目的は、ユーザーが粒子間相互作用の計算を並列化を意識することなく、並列に実行できるようにすることである。そのために Pyker という Domain Specific Language(DSL) を開発した。Pyker では、ユーザに必要な変数の定義と数式から相互作用の計算を SIMD 命令によって並列化したコードを生成する。Pyker の評価方法としては、C 言語のコンパイラが自動で行う SIMD 化と Pyker による SIMD 化の性能を比較する。そのために、3 つの相互作用の計算を行うコードをそれぞれに Pyker から生成された SIMD 命令を適応しない場合のコードと、SIMD 命令を適応したコードの実行時間を計測し比較する。また、Pyker を作成するにあたって粒子間相互作用の計算するコードを生成する PIKG を参考にした。

2 Construction

We use Sympy, a library in Python for implementing the implementation of pyker. SymPy supports various operations, and the formulas using these operations are internally treated as syntax trees. We leveraged this functionality to read formulas and generate code capable of parallel execution. For parallelization, we used an instruction set named Advanced Vector Extensions 2 (AVX2). AVX2 is one of the extended instruction sets implemented in Intel's CPUs. The next section will explain how parallelization is carried out using AVX2.

2.1 AVX2 for parallelization

To perform SIMD operations, just like with regular computations, data is first loaded from memory, computed based on that data, and the results are stored. However, in the case of SIMD, computations are performed using SIMD-specific registers, so the methods of loading, computing, and storing differ. When loading, multiple elements that will undergo the same instruction are stored in a SIMD-specific register. These are then used to perform computations simultaneously, and the results are stored for each.

For instance, consider a calculation like C = X + Y, where elements of double-precision floating arrays x and y are computed . When using AVX2 for SIMD, since the registers are 256 bits, four elements from x and y are loaded each. These are then added simultaneously, and the results are stored in the array C. When loading into SIMD registers, if it's a 1-dimensional array, contiguous memory access is sufficient, but for data structures like Array of Structures (AoS)shown in Figure a., elements need to be loaded individually.

For example, when considering the 3-dimensional coordinates of particles as double-precision floats in pos[n][3](where n is the total number of particles, and pos[i][0], pos[i][1], pos[i][2]represent the x, y, z coordinates of the i-th particle, respectively), and calculating the difference in x coordinates between particles i and j, it becomes dx = pos[i][0]- pos[j][0]. However, for parallelization with SIMD, desiring 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], the

data are packed as (pos[i][0], pos[i+1][0], pos[i+2][0], pos[i+3][0]), (pos[j][0], pos[j][0], pos[j][0], pos[j][0), (dx0, dx1, dx2, dx3)and subtraction is performed simultaneously. However, since data needs to be loaded onto SIMD registers all at once, the gather instruction is used to load data for particle i. The gather instruction is used to read non-contiguous data elements from memory addresses by specifying addresses. An explanation of the gather instruction is shown in Figure x. Data loaded in this way are computed simultaneously using SIMD instructions.

```
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]
```

```
(x1, x2, x3, x4) + (y1, y2, y3, y)
```

When storing computation results, addresses may not be contiguous. In such cases, there is an instruction called scatter, which is the counterpart to gather, but it is not supported in AVX2, so data are stored using pointers individually. In this way, parallelization is achieved with SIMD. Moreover, these instructions can be explicitly handled in high-level programming languages like C and C++ using intrinsic functions. To handle intrinsic functions in C and C++, include the file immintrin.h, which is available as a standard in the language. An example of C++ code that calculates the difference in x coordinates, simd_tmp.cpp, is shown. Pyker generates C++ code capable of executing SIMD instructions like simd_tmp.cpp from the described formulas. The next section will explain the Pyker.

2.2 Pyker

In this DSL, code is generated by writing and compiling two definitions: the definition of variables and the definition of interaction formulas. First, let us explain the variable definition.

In this DSL, variables are classified into classes: EPI, EPJ, FORCE, and others. EPI represents particles that receive interactions, while EPJ represents particles that provide interactions. FORCE holds the results of the interaction calculations, and other variables include softening parameters and the like. To perform calculations using these variables, in C++, it would be similar to 2.

Listing 1: skeleton code

```
int kernel(double xi[][3], double xj
    [][3], double ai[][3], double eps2
    , int n){
  // (1).preprocess
  for(int i = 0;i < n;i += 4) { //
      (2)Increment the number of
      parallels
    // (3).load EPI
    // (4).Initialization of tmporary
        force
    for(int j = 0;j < n;j += 1) {
        // (5) load EPJ
        // (6) calculate interparticle
            interactions
    }
    // (7). Store calculation result in
        the FORCE
  }
}
```

(1) involves preparatory steps such as defining variables necessary for the calculation. (2) involves loading the variables of EPI. (3) is about initializing variables that temporarily hold the results of the interaction calculation. (4) involves performing the interaction calculation and saving the results in the primary variable of force. (5) involves storing the results in the FORCE variables. To generate this code, in variable definition, information about the class of the variable, its type, and its dimension is necessary. Therefore, in pyker, variable definitions are written as follows. . The first column, if it is a class, writes the name of the class and its member variable name. The second column explicitly writes the dimensions of the vector if there are any, specifying 3 or 4 dimensions, and the type is either 32-bit or 64-bit floating-point. The third column becomes the name of the variable handled on pykg. For example, "EPI.pos vec3<F64> xi" would be a member variable of EPI named pos, a 3-dimensional 64-bit floating-point variable named xi. The definition of interaction formulas is written to accumulate the results in the FORCE variable using the defined variables. Primary variables necessary for the mathematical description can be newly defined using previously defined variables. Available operations include basic arithmetic, sqrt, and the power symbol "**". The result is stored in the FORCE variable using '='.For instance, to generate code that c alculates the gravitational interaction formula shown in Figure 1, it would be written in this DSL as figure1.pyker.

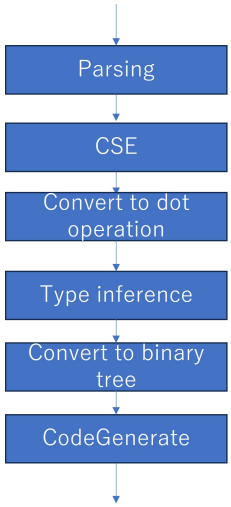
Listing 2: hoge

```
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
```

2.3 Implementation

The process of generating code is illustrated in the following flowchart. In Parsing, the information of the

variables defined in the DSL code is converted into a hash with the variable names as keys. Formulas are converted into syntax trees using the sympify() method of Sympy, and a list of these trees is obtained. Next, Common Subexpression Elimination (CSE) is performed on the syntax trees of the converted formulas. CSE is the process of reducing the number of operations by pre-calculating common subexpressions in formulas and using their results in subsequent calculations. In type inference, the types of the primary variables in the formulas are inferred. Afterward, all syntax trees are converted into binary trees to align them with SIMD operations. Finally, the code for calculating interactions is generated by traversing the processed syntax trees. The following sections describe how the implementation was carried out in the order of Parsing, CSE, type inference, conversion to binary trees, and Code generation.



2.3.1 Parsing

In Parsing, the process is divided between variable definition and formula handling. In variable definition, objects with class, vector, type, and variable name are created respectively. A hash with the variable name as the key is then made (name_variable_map). Formulas use the sympify() method from the Sympy library to convert the code read as a string into a syntax tree of formulas, obtaining a list of these syntax trees (expr_list). Next, Common Subexpression Elimination (CSE) is performed.

2.3.2 CSE

2.3.3 type inference

2.3.4 convert to binary trees

2.3.5 code generatation of non SIMD

2.3.6 code generatation of SIMD

3 Experiments

3.1 gravity interparticle

ほぼ同じコードで実行

Listing 3: Nbody-kernel.pyker

```
EPI vec3<F64> ri
EPJ vec3<F64> rj
EPJ F64 mass
EPJ F64 eps2
FORCE vec3<F64> ai
rij = ri - rj
r2 = rij * rij + eps2
r_inv = 1.0 / sqrt(r2)
r2_inv = r_inv * r_inv
mr_inv = mass * r_inv
mr3_inv = r2_inv * mr_inv
ai = mr3_inv * rij
```

Listing 4: Nbody-kernel.pikg

```
EPI F64vec ri:r
EPJ F64vec rj:r
EPJ F64 mj:m
EPJ F64 eps2:eps
FORCE F64vec ai:acc
rij = ri - rj
r2 = rij * rij + eps2
r_inv = rsqrt(r2)
r2_inv = r_inv * r_inv
mr_inv = mj * r_inv
mr3_inv = r2_inv * mr_inv
ai += mr3_inv * rij
```

non-simd
g++ -O3

N	Pyker	PIKG
25000	6.725156	4.444766
10000	0.738918sec	0.517324sec

Listing 5: Nbody-kernel.pikg

```
g++ -O3 -I ~/school/PIKG/PIKG/inc -
    mavx2 -mfma gravity_interparticle.
    cpp
```

N	Pyker(sec)	PIKG(sec)
50000	5.119071	3.333270
25000	1.218129	1.148511
10000	0.143777	0.101594
1000	0.003521	0.002168

3.2 LennardJones

Listing 6: LennardJones-kernel.pyker

```
EPI F64 rix
EPI F64 riy
EPI F64 riz
EPJ F64 rjx
EPJ F64 rjy
EPJ F64 rjz
EPJ F64 eps

FORCE F64 fx
FORCE F64 fy
FORCE F64 fz
FORCE F64 p
dx = rix - rjx
dy = riy - rjy
dz = riz - rjz
r2 = dx * dx + dy * dy + dz * dz + eps
r2i = 1.0 / r2
r6i = r2i * r2i * r2i
f = (48.0* r6i - 24.0) * r6i * r2i
fx = f * dx
fy = f * dy
fz = f * dz
p = 4.0 * (r6i) * (r6i - 1.0)
```

Listing 7: LennardJones-kernel.pyker

```
  EPI F64 rix:rx
  EPI F64 riy:ry
  EPI F64 riz:rz

  EPJ F64 rjx:rx
  EPJ F64 rjy:ry
  EPJ F64 rjz:rz
  EPJ F64 eps2:eps

  FORCE F64 fx:fx
  FORCE F64 fy:fy
  FORCE F64 fz:fz
  FORCE F64 p:p

  dx = rix - rjx
  dy = riy - rjy
  dz = riz - rjz
  r2 = dx * dx + dy * dy + dz * dz +
       eps2
  r2i = 1.0 / r2
  r6i = r2i * r2i * r2i
  f = (48.0 * r6i - 24.0) * r6i * r2i
  fx += f * dx
  fy += f * dy
  fz += f * dz
  p += 4.0 * r6i*(r6i - 1.0)
```

	N	Pyker	PIKG
non-simd	50000	40.736862	11.656278
	25000	10.662050	3.521253
	10000	0.985475	0.442775

	N	Pyker	PIKG
simd	50000	3.702871	2.838773
	25000	0.838122	0.9904373
	10000	0.134388	0.215270

4 Conclusion

5 Acknowledgement

References

References

[1] J. Makino, K. Hiraki, and M. Inaba, “Grape-dr: 2-pflops massively-parallel computer with 512-core, 512-gflops processor chips for scientific computing,” Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC ’07, New York, NY, USA, Association for Computing Machinery, 2007.

[2] 小池邦昭, 藤野健, 福重俊幸, 台坂博, 菅原豊, 稲葉真理, 平木敬, 牧野淳一郎, *et al.*, “超並列準汎用計算機 grape-dr による重力多体問題シミュレーションおよび lu 分解,” 研究報告計算機アーキテクチャ (ARC), vol.2009, no.26, pp.1–11, 2009.