Domain Specific Language for high performance computing

Ryuki Hiwada s1280076 Supervised by Naohito Nakasato

1 Abstract

2 Introduction

2.1 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.

2.2 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.

2.3 Overview of DSL

We have created a Domain Specific Language (DSL)named (DSL name). This language's processor can read code describing particle interaction formulas and generate code that accelerates these computations. The language allows for the definition of variables and the description of interaction formulas. In the variable definition section, it's possible to define variables such as the mass and position of particles, variables for storing results, and other necessary variables. For the formula part, we have enabled calculations including basic arithmetic operations, sqrt, power functions, and computing the norm of vectors. From this description, a kernel function is generated, which can then be called and used. We have created a Domain Specific Language (DSL)named (DSL name). This language's processor can read code describing particle interaction formulas and generate code that accelerates these computations. The language allows for the definition of variables and the description of interaction formulas. In the variable definition section, it's possible to define variables such as the mass and position of particles, variables for storing results, and other necessary variables. For the formula part, we have enabled calculations including basic arithmetic operations, sqrt, power functions, and computing the norm of vectors. From this description, a kernel function is generated, which can then be called and used.

2.4 Aim of this paper

3 implementation

We use Sympy, a library in Python for implementing the implementation of our (DSL). 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.

3.1 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. 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. DSL name generates C++ code capable of executing SIMD instructions like simd_tmp.cpp from the described formulas. The next section will explain the design of DSL name.

3.2 design of name of DSL

```
#include <iostream>

int main(){
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

3.3 implementation of (name of DSL)

3.4 Use Sympy for DSL development

4 Conclusion

5 Acknowledgement