PNBsolver: A Domain-Specific Language for Modeling Parallel N-body Problems

Ferosh Jacob

Data Science R&D, Careerbuilder LLC

Md Ashfakul Islam

Department of Computer Science, University of Alabama

Weihua Geng

Department of Mathematics, Southern Methodist University

Jeff Gray, Susan Vrbsky, Brandon Dixon

Department of Computer Science, University of Alabama

Purushotham Bangalore

Computer and Information Sciences,University of Alabama at Birmingham

Abstract

With the advent of multicore processors, parallel computation has become a necessity for next generation applications. It is often a tedious task for domain users to optimize their programs for a specific platform, algorithm, and problem size. We believe that domain users should be freed from this task and they should be equipped with tool support to reuse the optimized solutions written by expert parallel programmers. In this paper, we introduce a two-stage modeling approach that allows domain users to express the problem using domain constructs and reuse the available optimized solutions. This approach has been applied successfully to N-body problems using a domain-specific language called PNBsolver, which allows domain users to specify the computations in an N-body problem without any implementation or platform-specific details. Using the PNBsolver, the domain users are allowed to control the platform and implementation of the generated code. The accuracy and execution time of the generated code can also be fine-tuned based on the parameters provided in

*Corresponding author

Email addresses: ferosh.jacob@careerbuilder.com (Ferosh Jacob), mislam@crimson.ua.edu (Md Ashfakul Islam), wgeng@smu.edu (Weihua Geng), gray@cs.ua.edu, vrbsky@cs.ua.edu, dixon@cs.ua.edu (Jeff Gray, Susan Vrbsky, Brandon Dixon), puri@cis.uab.edu (Purushotham Bangalore)

PNBsolver. We also show how common N-body interactions are implemented using PNBSolver.

1. Introduction

Even though the first computing machine built by Charles Babbage is believed to be a parallel machine [1], until the last few years parallel computing was reserved for the scientific community. Recently, parallel programming has emerged as an essential skill for software developers, primarily due to two reasons: 1) As the chip industry hit the power wall, manufactures shifted to chip multiprocessors (CMPs), and 2) Introduction of Graphical Processing Units (GPUs) for general-purpose computation (GPGPU) [2]. To utilize the increased computation power available in desktops and powerful GPUs, parallel computing is an essential skill for the next generation of software engineers. Because of the nature of parallel applications, programmers will have to adapt to a new style of programming to use the resources, as well as to maintain and evolve for future applications [3].

Popular parallel programming paradigms include: 1) OpenMP [4] (Shared memory), 2) Message Passing Interface[1] (Distributed memory), and 3) CUDA[2] and OpenCL[3] for GPU platforms. All of these parallel programming paradigms have their advantages and disadvantages for a given problem, based on the execution environment, implementation, and even size of the problem. In the current scenario, to find the optimized execution time for a problem of a given size in a specific platform, a programmer must manually write a parallel version, optimize it and compare the execution time with other platforms. It is often hard for programmers to optimize these programs because their domain of expertise is the problem domain and not the High Performance Computing (HPC) domain. To solve this problem, programmers should be freed from implementation and optimization of parallel versions, but should have the flexibility to express the problem and switch between execution environments (optimal execution time is also a function of problem size). There is no magical tool to convert any sequential program to an optimized parallel program. However, for a restricted domain this can be achieved [5, 6, 7]. In this paper, we address two research questions that are summarized in the following subsections.

1.1. Q1: Separating specification and implementation

Which algorithm on a specific platform can provide the optimized execution for a problem of fixed size? To answer this question, the best way is to develop and execute programs implementing various algorithms in that platform for a fixed size. This leads to the task of rewriting thousands of lines of code. Can

[1] MPI, `http://www.open-mpi.org/`
[2] CUDA, `http://developer.nvidia.com/category/zone/cuda-zone`
[3] OpenCL, `http://www.khronos.org/opencl/`

we reuse the implementation for another similar problem? This is possible only if the implementation details are separated from the problem specification. Is there a better algorithm which can give the optimized execution for the specific-problem? This cannot be linked with the existing code even if the problem specification is separated from the implementation.

1.2. Q2: Abstract problem specification

How can we maintain and switch between many program versions to provide the optimized execution time? As mentioned earlier, there can be many implementations for a given problem specification. If all such implementations are based on an abstract problem specification, any of these implementations can be used with the given problem specification. If the domain is limited, we can provide optimized solutions in required platforms for each of these implementations.

We used modeling techniques from software engineering to provide separation of specification and implementation and abstraction in a restricted domain. Our solution approach is explained in the following subsection.

1.3. Solution approach: Modeling parallel programs

Models in the context of software modeling can be defined as the abstract representation of a system. Model-driven Engineering [8] is a methodology that makes use of a Domain-Specific Modeling Language (DSML) [9, 10] such that it can: 1) define a system using domain information, 2) analyze the domain, and 3) automatically generate source code or other artifacts by code generation techniques. A model is defined by a set of entities, associations and constraints, commonly called a metamodel [11]. In the case of a Domain-Specific Language (DSL) [12], a metamodel is the grammar defining the DSL. The domain experts define domain problems in a DSL file (an instance of the model), which is capable of capturing information to represent the whole system.

We developed a two-stage technique for modeling parallel programs. In the first stage, the domain experts create a model (DSL file) specifying the problem (this model has no platform-specific or implementation-specific details) and in the second stage, this model is mapped to different platforms and implementations based on user preferences. The main advantage of this two-step modeling technique is that it can be extended to include additional platforms and implementations and is well-suited for a domain like parallel programming.

In this paper, we introduce PNBsolver, which is a DSL based on our two-stage modeling technique. PNBsolver can represent N-body problems without any implementation details and also has the flexibility to allow users to switch between implementations (algorithms) and platforms. Section 2 overviews N-body problems and analyzes a commonly used algorithm in such problems. Section 3 describes PNBsolver from a user's perspective and the implementation details are explained in Section 4. Section 5 gives details of a comparison study of PNBsolver solutions with existing implementations. Additional examples using PNBsolver are shown in Section 6. Related works are reviewed in Section 7 and the paper is concluded in Section 8.

2. N-body problems and tree code algorithm

The N-body problem is a generalized concept of problems related to the physical properties of a system. The system can contain billions of interacting bodies. The classic N-body problem of celestial mechanics originated from Newton's Principia [13]. The bodies can be giant celestial objects in a solar system or minuscule particles in an atomic system. The physical properties can be motion, energy, charge, momentum, or force. There are many areas in science that use N-body problems (e.g., astrophysics, plasma physics, molecular physics, fluid dynamics, quantum chemistry and quantum chromo-dynamics) [14, 15, 16, 17]. Barnes and Hut [18] introduced a hierarchical $\mathcal{O}(N \log N)$ algorithm to calculate an N-body force equation and was later used in different N-body problems [19, 20, 21]. In this section, the tree code algorithm is reviewed, explaining how the efficient implementations of this algorithm can be reused in N-body computations.

2.1. Tree code algorithm

In the Tree code algorithm for computing N-body interactions, the 3-D space is divided into clusters in a hierarchical way. If the target particle is well-separated from a cluster satisfying the multipole acceptance criterion (MAC), the particle-particle interactions between the target particle and all source targets in the cluster are replaced by a particle-cluster interaction. To this end, a truncated (specified by ORDER, the number of terms of the expansion) Taylor expansion of all particles in the cluster about the center of the cluster is performed. The computation involves two parts: 1) the Taylor coefficients as functions of the distances between the target particle and the center of the cluster, which is computed by an efficient recurrence relation, and 2) the moments as functions of the distance between the center and all particles of the cluster, which is pre-computed and stored. Please refer to [19] for more details.

The tree code algorithm has proved its efficiency for N-body problems [18]. It reduces the computational cost of these problem from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log(N))$. The tree code algorithm is explained in two perspectives: 1) Mathematical formulation; and 2) Source code. In the mathematical formulation, the algorithm is explained with the help of a 3D cube, which itself is divided in all dimensions into subunits recursively. In the source code, this is represented as a tree structure.

2.1.1. Tree code algorithm: A mathematical perspective

$$I_i = \sum_{j=1, j\neq i}^{N} M_j G(x_i, y_j) \tag{1}$$

In a system consisting of N bodies, the interaction of body i with other $N-1$ bodies is given in Equation 1. According to the tree code algorithm, for bodies within a cell/cluster (one of the subunits of the 3-D cube) satisfying the MAC (θ) [22] in Equation 2, their interaction with body i can be calculated more

efficiently with controllable errors. In Equation 2, r_c represents the location of the cell center and R represents the distance between the ith body and the cell center. On satisfying the MAC, the more efficient particle-cluster interaction between body i and bodies inside a cell is given as $I_{i,c}$ in Equation 3 from [19, 23, 24]. Please note that k, x, and y are vectors with three components.

$$\frac{r_c}{R} < \theta \tag{2}$$

$$I_{i,c} \approx \sum_{||k||=0}^{p} \frac{1}{k!} D_x^k G(x_i, y_c) \sum_{y_j \in c} M_j (y_j - y_c)^k \tag{3}$$

$$I_{i,c} \approx \sum_{||k||=0}^{p} TCOEFF(k, G) \times MOMENTS(c) \tag{4}$$

In Equation 3, for each 3D value k, the first term computes the Taylor coefficients for every target body (using recurrence relations) and the second term computes the moment of the cluster, which is only evaluated once for each cluster. In general, Equation 3 can be rewritten as Equation 4. From an implementation perspective, to implement this algorithm for any interaction, programmers only need to choose one of our included interactions or provide the Taylor coefficients for user-specified interactions. In Equations 3 and 4, p represents the ORDER of the Taylor coefficients.

2.1.2. Tree code algorithm: A source code perspective

A general implementation of the tree code has three stages: 1) Tree creation, 2) node values calculation, and 3) Tree traversal. Details of these stages are explained as follows.

Tree creation. At this stage, a tree-based structure is used to maintain a list of bodies. Bodies of the system are added to the root node and if any node has more than a fixed number of bodies (MAXPARNODE), the node is further branched to eight child nodes. This is equivalent to dividing a 3D space recursively until each of the existing subunits has at most MAXPARNODE number of bodies.

Node values calculation. At this stage (usually implemented along with tree creation), the node properties are calculated. This includes calculating the resultant center of the node and arranging inputs such that bodies in the same node are arranged closer for faster access. Note that each intermediate node of the tree contains the approximate interaction information (the second term, moments, in Equation 3) of the sub-units under that intermediate node. Hence, the root node can approximate the entire system.

Table 1: An analysis of existing tree code implementations

Name	Equation	Language	LOC (files)
Total PE	$\sum_{i=1}^{N-1} Q_i \sum_{j=i+1}^{N} \frac{Q_j}{\lvert R_i - R_j \rvert}$	FORTRAN	966 (3)
PE_i	$\sum_{j=i, j\neq i}^{N} \frac{Q_j}{\lvert R_i - R_j \rvert}$	FORTRAN	1006 (3)
Screened_i	$Q_i \sum_{j=i, j\neq i}^{N} \frac{Q_j e^{-\lvert R_i - R_j \rvert}}{\lvert R_i - R_j \rvert}$	FORTRAN	1023 (3)
Grav. F_i	$M_i \sum_{j=i, j\neq i}^{N} \frac{M_j (R_i - R_j)}{\lvert R_i - R_j \rvert^3}$	C	1921 (15)
Grav. F_i	$M_i \sum_{j=i, j\neq i}^{N} \frac{M_j (R_i - R_j)}{\lvert R_i - R_j \rvert^3}$	Javascript	908 (6)

Tree traversal. In this stage, the interactions are calculated. The bodies are traversed from the root node. If a node satisfies the MAC, traversal is stopped and a node's approximated value is used as the interaction. If MAC is not satisfied, the traversal is continued until it reaches the leaves. At the leaves, interaction is computed by direct summation.

A summary of five tree code implementations collected from different sources [25, 26, 27] implementing different interactions is shown in Table 1. We leave out some constants such as dielectric or gravitational constants for simplicity. As shown in the table, all the implementations had roughly 1000 lines of code (LOC).

2.2. Analysis summary

There are parallel implementations of tree code available [28, 29, 30]. In general, there are two parallel implementations: 1) Parallelizing the node interactions, interactions are distributed equally among the parallel instances (threads or processes) and every instance creates their own tree (same tree) for computations; 2) Parallelizing tree creation and node interactions, the tree creation, as well as tree walking, is distributed among the instances. In most cases, direct computation is embarrassingly parallel. Hence, when such a computation is executed in a highly parallelized device like a GPU, the direct computation can outperform the tree code algorithm for a range of size. For a larger size problem, the $\mathcal{O}(N \log(N))$ tree code can outperform the $\mathcal{O}(N^2)$ direct summation. This justifies our combination of parallel direct summation implementation in a GPU, parallel implementation of tree code in a CPU, and parallel implementation of tree code in a GPU. The research questions introduced in Sections 1.1 and 1.2 are revisited in the context of N-body problems in the following subsections.

1. **Separating specification and implementation in N-body problems:** The domain users should be able to specify the N-body problems without any implementation details (e.g., which algorithm to use). They

should also be able to provide the implementation details (e.g., desired accuracy, allowed error, platform). However, the problem specification should be logically separated from the other details.

2. **Abstract problem specification in N-body problems:** The problem should be represented at the correct abstraction level. It should not be too high such that code optimizations cannot be applied (e.g., users should be allowed to configure parameters like MAXPARNODE, ORDER, THETA) and not too low level (e.g., users might not want to specify the bodies as charge or mass).

We designed PNBsolver to address these questions. PNBsolver is introduced in the following section from a user's perspective.

3. Working with PNBsolver

The equation to calculate the gravitational force for each body in an N-body system with masses M_i and positions R_i is shown in Equation 5. An equivalent representation of the force using PNBsolver is shown in Figure 1. In the figure, the `Force` kernel is defined in space `R`, more specifically a three dimensional position vector $\mathbb{R}^3$. A "pnb" (parallel N-body) file can be logically divided into three sections: 1) Declarations, 2) Calculations, and 3) Generation. Each file section is explained in the following subsections.

$$F_i = KM_i \sum_{j=1, j\neq i}^{N} \frac{M_j(R_i - R_j)}{|R_i - R_j|^3} \tag{5}$$

3.1. Declarations

Every variable used in the calculation section should be declared before usage. PNBsolver identifies three types of variables: 1) vector, 2) scalar, and 3) constants. Vectors and scalars are linked to every body in the system. The total force that acts on a body or position of a body at an instance are examples of vectors; the mass or charge of a body are example of scalars. The constant data type is used to specify the constants in the calculation and also the global properties of the system. The gravitational constant or the dielectric constant are examples of constant data types. The scalar and the constant variables can be initialized along with their declaration, as shown in Figure 1 (line 6). In the case of a scalar, all the `N` bodies will be initialized with the given value.

3.2. Calculations

Actual calculation for the kernel is specified in this section. As shown in Figure 1 (line 16), the calculation involves two expressions. The first expression, $K * M$ corresponds to the KM_i in Equation 5. For discussion, this expression is called the outer expression and the second expression is called the inner expression. PNBsolver expects the inner equation to perform the `SUM` operation.

```
kernel Force in R

// Kernel declarations
  vector F
  scalar M
  constant K=1

 /*
  * Read positions and mass from file
  * formatted as <x y z>m
  *
  */
  read "<R_1,R_2,R_3>M", "data.dat"

  //Actual computation
  F=K*M SUM(M*R/(R_*R_*R_))

  // Write force to file
  write "F_1,F_2,F_3", "out.dat"

endkernel

// Generate CUDA code for force kernel
generate CUDA ACCURATE Force.
```

Figure 1: Gravitational force kernel in space R using PNBsolver

```
  //A cluster not satisfying MAC

  //Variable declarations
  double dx, dy, dz, dist, distsq, temp1;

  //Calculating distance variables
  dx = R[i] - tarpos[0];
  dy = R[i] - tarpos[1];
  dz = R[i] - tarpos[2];
  distsq = dx * dx + dy * dy + dz * dz;

  //Avoiding i==j
  if(distsq > 0.0f){
            dist = sqrt(distsq);

            //Calculating specified variables
           temp1 = dist * dist * dist;

            //Actual calculation
            Forcex = Forcex + M[i] * dx / temp1;
            Forcey = Forcey + M[i] * dy / temp1;
            Forcez = Forcez + M[i] * dz / temp1;
  }
```

Figure 2: Code generated for the tree code algorithm with MPI/OMP

In addition to the variables declared, PNBsolver identifies the position vector in both inner and outer expressions. For the `Force` kernel, the position vector is R (line 1). In the outer expression, R represents the actual position, and in the inner expression R gives the relative position with the iteration value. For calculations, the magnitude of a vector can be obtained by adding the "_" to the name of the variable (`R_` in line 16). PNBsolver achieves separation of

```
1  //Variable declarations
2  TYPE dx, dy, dz, distsq, invdist, temp1;
3  //Calculating distance variables
4  dx = Ri.x - tarpos.x;
5  dy = Ri.y - tarpos.y;
6  dz = Ri.z - tarpos.z;
7  distsq = dx * dx + dy * dy + dz * dz;
8
9  if(distsq > 0.0f){
10            //Using CUDA Fast functions
11            if(ACCURATE)
12                invdist = rsqrt(distsq);
13            else
14                invdist = rsqrtf(distsq);
15
16            //Calculating specified variables
17            temp1   =  invdist * invDist * invDist;
18
19            //Actual calculation
20            Force.x += dx * tarpos.w * temp1;
21            Force.y += dy * tarpos.w * temp1;
22            Force.z += dz * tarpos.w * temp1;
23 }
```

Figure 3: Kernel code generated for direct summation with CUDA

calculations within the expression statement with "(" brackets. The expression used in the brackets are computed to a temporary variable before the final computation and the temporary variable is replaced with the occurrences of the expression in the final expression statement. This is done to optimize the computation.

Read and write statements: This section helps PNBsolver to integrate with existing programs or functions. In this section, the vectors and scalars that are declared are initialized. The read statements read the values from a file to the variables and write statements write the computed results to a file. Both of these commands take two parameters: 1) Format and 2) File name. The format argument specifies how values for each body are formatted in a line. A single kernel can have many read and write statements. As an example, if masses were specified in a different file, another read statement could be added before the calculation statement. For reading and writing formats, the three co-ordinates of vectors can be accessed by adding "_1, _2," and "_3," respectively, to their names (line 13 and line 19).

3.3. Generation

The first two sections can express the N-body problem efficiently without any implementation details. This section controls the implementation and code generation. The default code generation language is "C" and the algorithm is selected based on the mode and target parallel programming paradigms. PNBsolver supports CUDA, OMP, and MPI parallel programming paradigms. There are three modes for each paradigm: 1) ACCURATE, 2) AVERAGE, 3) FAST. The program runs faster when a mode is changed from ACCURATE to FAST and a program gives more accurate results when the mode is changed from

FAST to ACCURATE. More details about how algorithms are selected based on the mode and paradigms are explained in the implementation section.

A section of code generated for the tree code algorithm and CUDA direct implementation is shown in Figures 2 and 3. This code computes the Force for the body located at R_i due to another body located at `tarpos`. Both implementations have pre-defined variables, `dx, dy, dz, distsq`. The `distsq` calculates the magnitude of the distance. The `if` statement could be avoided if we set $M_i = 0.0$ before the computation, but our current implementation generates the `if` statement. However, if there is a softening factor to the magnitude, the PNBsolver parser identifies this and removes the `if` statement. An example for such a case is explained in Figure 15.

In Figures 2 and 3, `temp1` (line 4 and line 2, respectively) is a variable created by using brackets in the expression statement. As shown in the figure, the calculation is implemented in two different ways for the two versions (line 17 in Figures 2 and 3). For the CUDA implementation, the type of the variable is defined based on the mode, for faster implementations `float` is used and for accurate implementations `double` is used. Hence, the type of the variable is defined as a C++ template variable `TYPE` (variable is removed by actual code during compilation). `ACCURATE` is another template variable to tune optimization and speed. For distance, because CUDA has a fast math function, to perform the square root and inverse, we use that function (Figure 3), instead of the square root followed by the division operator (Figure 2) as in the CPU code. Use of the `float4` data type for the position and mass is another GPU optimization. If the variable `FAST` is set, CUDA fast functions (e.g., `rsqrtf,__expf(x)`) are used.

4. Implementation details of PNBsolver

A block digram showing the implementation of PNBsolver is illustrated in Figure 4. The "pnb" file is passed to the parser and the parser identifies the mode and platform for selecting the proper template. The parser also generates the kernel code which is further optimized and given to the code integrator. The optimizations applied at this stage are very general and they are not coupled to any specific platform. Removing the brackets with new temporary variables, finding expressions that are repeated, and checking whether softening is applied are examples of optimizations at this stage. The code integrator merges the kernel code to the template and fills the template values. This includes updating function declarations and parameters, function calls and arguments, and declaring type variables and initialization. After this, platform-specific optimizations are applied. Rewriting functions with CUDA fast math functions is an example of such an optimization.

The project is implemented in Java and ANTLR[4] is used for implementing the "pnb" file parser. The template store is implemented using StringTem-

[4] ANTLR, `http://www.antlr.org/`

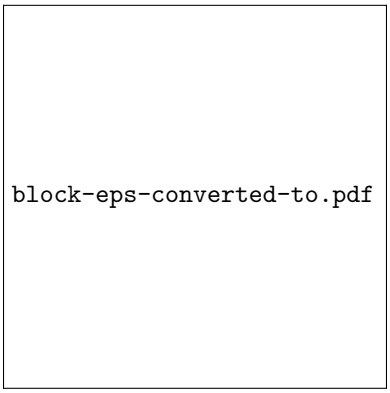

Figure 4: Block diagram showing the implementation of PNBsolver

plate[5]. Two important parts of the implementation are "pnb" file parsing and code generation using templates. The code generation varies based on the mode and platform selected in the "pnb" file. However, the code generated will be either a parallel direct summation implementation or a parallel tree code implementation. The PNBsolver Parser, PNBsolver code generator, and Modes of operation are explained in the following subsections.

4.1. PNBSolver Parser

The simplified EBNF grammar for the PNBsolver language is shown in Figure 5. As shown in the figure, the file allows one or more kernels, but code is generated for only one kernel. This is designed to support future extensions, where the output of a kernel can be passed to the input of another kernel. After parsing, the expression statement is captured in two expression objects, the inner expression object and the outer expression object. All expressions in a "pnb" file are made of variables defined in the declaration section. To use a constant inside the expression statement, it should be declared as a constant and use the variable name in the expression statement. As a rule, the read statements should be defined before the expression statement, and write statements after the expression statement. This seems logical as a program usually requires reading before the calculation and writing after the calculation. The EBNF rules for `variabledeclaration`, `writestmts`, and `parameters` are not shown in the figure to provide clarity.

4.2. PNBsolver code generator

The PNBsolver parser can identify the inputs and outputs for the kernel. The code generator inserts declarations and initialization (if any) into the program. The core computation code is generated as shown in Figures 2 and 3. These two

[5]StringTemplate, `http://www.stringtemplate.org`

```
1
2 grammar PNBsolver;
3
4 content
5   : kernels  execute "." EOF
6   ;
7
8 kernels
9   : ("kernel"  ID "in" ID declarations readstmts
10            expressionstmt writestmts "endkernel")+
11  ;
12
13 declarations
14  : (type variabledeclaration)+
15  ;
16
17 execute
18  : "generate" platform ("[" parameters "]")? mode ID
19  ;
20
21 platform
22  : "CUDA"|"OMP"|"MPI"|"OCL"
23  ;
24
25 mode
26  : "ACCURATE"|"AVERAGE"|"FAST"
27  ;
28 type
29  : "vector"|"scalar"|"constant"
30  ;
31
32 expressionstmt
33  : IDENTIFIER "=" (expression)? "SUM" expression
34  ;
35
36 expression
37  : multidiv( "+" multidiv|   "-" multidiv )*
38  ;
39
40 multidiv
41  : atom ("*" atom |"/" atom)*
42  ;
43
44 atom
45  : IDENTIFIER
46  | "(" expression ")"
47  | "exp" expression
48  | "pow" "("expression "," NUMBER ")"
49  ;
50
51 readstmts
52  : ("read" STRING "," STRING)*
53  ;
```

Figure 5: Simplified EBNF grammar of PNBsolver

code sections are inserted into the template identified by mode and platform. To add a new algorithm, we have defined a new template and configured the parser to route through a different code generator. The same approach can be used to support a language other than C. In Section 5, we generate FORTRAN code using a FORTRAN code generator. There are four "C" templates available in the template store and are explained in the following subsections.

```
struct tnode {
    int numbodies,begin,end;
    double x_min, y_min, z_min,x_max, y_max,z_max;
    double x_mid,y_mid,z_mid,radius;
    int level,children,exist_ms;
    double ***moments;
    struct tnode* child[8];
};
```

Figure 6: Structure `tnode` in the template

4.2.1. Parallel CPU (OpenMP and MPI) tree code templates

The tree code used for PNBsolver is adapted from [19]. We developed parallel versions of the tree code in MPI and OpenMP. More details about the speedup, error and execution time are included in Section 6. To complete a CPU tree code template, we need: 1) Taylor coefficients, 2) Direct implementation, and 3) Tree code parameters such as MAXPARNODE, ORDER (taylor coefficient order) and THETA (MAC). PNBsolver sets default values for all of these parameters, but this can be customized by the user as shown later in Figure 10.

Providing Taylor coefficients: The parameter value TAYLOR is used to find the Taylor coefficients. The PNBsolver code generator reads the file specified as the parameter value of TAYLOR and looks for a function with signature `comptcoeff (struct tnode *t)`, where `tnode` is a structure representing the tree node. In this function, the position of an interacting body can also be accessed. The declaration of structure `tnode` is shown in Figure 6. The values of the Taylor coefficients should be set in a 3D array of size ORDER. In addition to the `comptcoeff` function, users can add two more functions: `setup()` and `teardown()`, for declaring values that might be required for the computation. These two functions are executed only once before and after the tree creation, while the `comptcoeff` function is executed for every target that satisfies THETA. All of these functions can access the user-specified parameters and some global parameters which include NUMPARS (number of bodies). If PNBsolver is executed with a TAYLOR parameter, it will look for a file specified as the value. If PNBsolver cannot find the file in the current directory, a file having the above three function signatures will be created. PNBsolver uses Taylor coefficients mentioned in [19] if TAYLOR is not specified and it is found effective for Force and Potential calculations.

4.2.2. Parallel GPU template for direct computation

For the GPU implementation using the direct summation, we have used two optimization techniques: 1) Tiling (partitioning of the computation domain into smaller tiles) with shared memory, and 2) Loop unrolling (replacing a loop with similar independent statements). The effect of these optimizations for Coulomb Potential is shown in Figure 7. In the figure, the direct version is the case when the kernel code generated was executed as a CUDA kernel without any modifications on a Tesla M2070. All kernels in the GPU template can be executed for `double` and `float` mode. In the figure, Speedup is defined as the

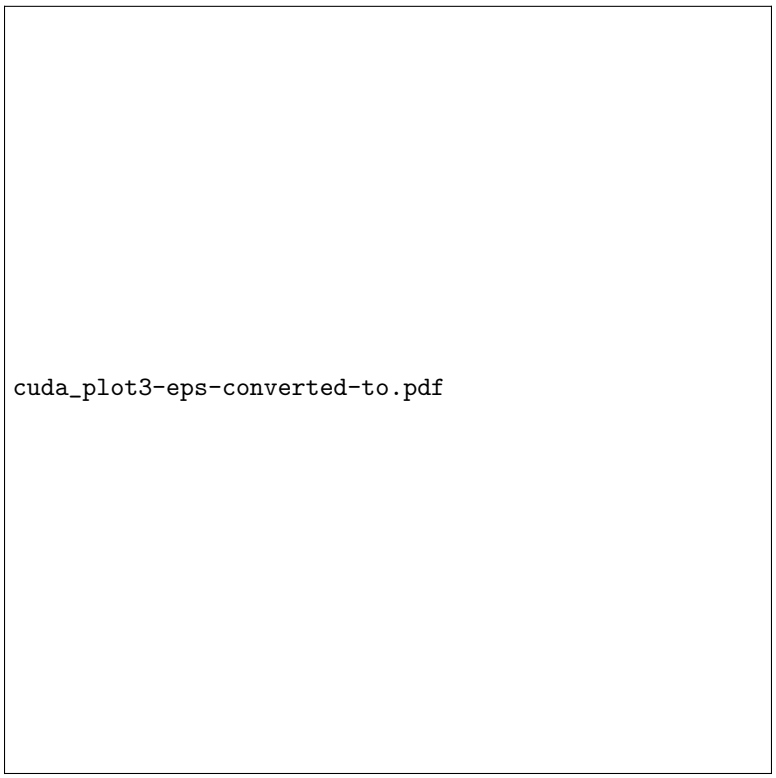

Figure 7: Effect of optimization techniques (Tiling and unrolling) for Coulomb potential

ratio of execution time of sequential implementation of Coulomb potential on a CPU to that of the CUDA implementation. Every reading is an average value of at least three of the same executions.

4.2.3. Parallel GPU tree code template

The implementation we use for GPU tree code is adapted from [29]. Our current implementation only supports zeroth ORDER; hence, there is no need to specify Taylor coefficients for tree code implementation.

4.3. Modes of operation

There are three modes of operation for PNBsolver: 1) ACCURATE, 2) AVERAGE, and 3) FAST. The execution time of the problem decreases from ACCURATE to FAST and error decreases from FAST to ACCURATE. This is achieved by varying the parameters ORDER and THETA. The effect of these two parameters for two problems executed for a PNBsolver OpenMP solution for a fixed size is shown in Figures 8 and 9. As seen from the figure, as the ORDER increases, error is less but the program takes longer to finish execution. In the case of THETA, the program finishes faster for higher values of THETA, but has