TOKYO INSTITUTE OF TECHNOLOGY

MASTER THESIS

Nested object support in an object-oriented domain-specific language for GPGPU

Author:
Jizhe CHENXIN

Student Number:
19M30407

Supervisor:
Prof. Hidehiko MASUHARA

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Programming Research Group
Department of Mathematical and Computing Science

April 1, 2021

Declaration of Authorship

I, Jizhe CHENXIN, declare that this thesis titled, "Nested object support in an object-oriented domain-specific language for GPGPU" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

School of Computing
Department of Mathematical and Computing Science

Master of Science

Nested object support in an object-oriented domain-specific language for GPGPU

by Jizhe CHENXIN

GPGPU (general purpose computing on graphics processing units) is one of the cost-effective methods of parallel programming. DynaSOAr is a memory allocator that enables high performance Object-Oriented GPGPU programming. It requires programmers to write code in C++/CUDA, which are low-level programming languages, while paying attention to memory allocation. Nested objects are not supported in DynaSOAr, which restricts users from programming in a more Object-Oriented way. We propose Sanajeh, an Object-Oriented Python DSL (Domain-Specific Language) that supports nested objects. Sanajeh code which runs on the GPU is compiled into C++/CUDA. During the compilation, nested objects are decomposed into top-level objects' fields with basic types. The top-level objects are then allocated in the GPU memory in SOA (Structure of Arrays) data layout by DynaSOAr. We evaluate Sanajeh with an N-Body simulation program.

Acknowledgements

I would like to express my appreciation to all people who have helped me in finishing this dissertation, especially to Prof. Hidehiko Masuhara. Prof. Hidehiko Masuhara accepted me as a master program student in the Programming Reasearch Group and spared his time for supervising and guiding me throughout two years. Without him I couldn't have finished this dissertation. I also would like to thank Matthias Springer for his kind guidance. He kept supporting me with the issues of DynaSOAr, which is his doctor's research, even after his graduation.

For the financial supports, I would like to thank the Financial Aid Group of Tokyo Institute of Technology for exempting half of my tuition. We have all been through a tough time in 2020 due to COVID-19 and with the exempting of the tuition I got the chance to continue and focus on my research.

Contents

List of Figures

Listings

List of Abbreviations

GPGPU	**G**eneral-**P**urpose computing on **G**raphics **P**rocessing **U**nits
OOP	**O**bject-**O**riented **P**rogramming
HPC	**H**igh-**P**erformance **C**omputing
SMMO	**S**ingle-**M**ethod **M**ultiple-**O**bjects
DSL	**D**omain **S**pecific **L**anguage
SIMD	**S**ingle-**I**nstruction **M**ultiple-**D**ata
AOS	**A**rray **o**f **S**tructures
SOA	**S**tructure **o**f **A**rrays
FFI	**F**oreign **F**unction **I**nterface
AST	**A**bstract **S**yntax **T**ree

Chapter 1

Introduction

General-purpose computing on graphics processing units (GPGPU) is a method that uses GPUs (graphics processing units) to perform computations that are usually performed by CPUs (central processing units). Although a single core on a GPU operates at a lower frequency than a CPU does, by utilizing the computation power of multiple GPU cores, programmers can gain high performance in parallel programs in a more economical way.

However, GPGPU requires programmers to optimize the programs manually in a low-level language. This makes GPGPU programs hard to develop, debug and maintain. Object-oriented programming (OOP) is a widely used programming paradigm that has high level of abstraction which can be helpful in improving these issues but it was seem as inefficient in high-performance computing (HPC) since the data layout it uses limited the efficiency of data access on GPU memory.

Nonetheless, recent work has shown that efficient OOP is feasible on GPUs when following the *Single-Method Multiple-Objects* (SMMO) [12] programming model. In the Single-Instruction Multiple-Data (SIMD) architectures, one instruction is run on multiple data to apply parallelism while in SMMO architectures, one method is run on all objects of a class. SMMO model allows programmers to write GPGPU programs in OOP style. The performance of many SMMO applications depends on efficient memory (de)allocation.

A CUDA framework for SMMO applications is DynaSOAr [13]. DynaSOAr enables users to write GPGPU programs in OOP style in C++/CUDA. It allocates the data of objects on GPU memory dynamically and ensure whenever user creates/deletes objects the data is always stored in an SOA data layout, which is important for improving performance (section 2.1). However, programmers must adhere to a certain coding style (section 3.1) and follow certain coding conventions when using DynaSOAr (section 3.2). Also, DynaSOAr are not optimized for programs with "`nested object`" (section 3.3.1) so that nested objects are difficult to be implemented efficiently. (section 3.3)

In this dissertation, we propose a Domain-Specific Language (DSL) called **Sanajeh**. Sanajeh's goal is to simplify and improves DynaSOAr, by a more easier syntax and the support for nested object. Sanajeh is built on a high-level language Python, while its low-level parts are based on DynaSOAr. Sanajeh provides API (section 5.1) for programmers to write SMMO applications with high performance in normal Python syntax. It frees programmers from bothering with memory allocation of the data which resides on GPU, including nested object. Also, Sanajeh allows users to use Python libraries in HPC programs. We evaluated Sanajeh's with an N-Body simulation (chapter 6).

Chapter 2

Background

This chapter gives an overview of two kinds of data layout that used for storing data on GPU memory (section 2.1), an introduction of N-Body simulation (section 2.2) and a mechanism for calling C++ functions from Python (section 2.3).

2.1 Data layout for data allocation on GPU memory

The performance of an OOP GPGPU program is highly related to what data layout it applies for storing the objects on GPU memory since it decides whether data can be loaded from GPU memory efficiently. We discuss two kinds of data layout in this dissertation: Structure Of Arrays (SOA) and Array Of Structures (AOS)

In AOS, objects of a class are stored as contiguous blocks in the memory. Listing 2.1 shows an example of a C++ program which uses an AOS data layout for the array `bodies` and figure 2.1 shows how data is loaded from the GPU memory in this program.

LISTING 2.1: A C++ program with an array of objects (AOS)

```
class Body: {
private:
    float pos_x;
    float pos_y;
    float vel_x;
    float vel_y;
    float force_x;
    float force_y;
    float mass;
}

Body bodies[1000]
```

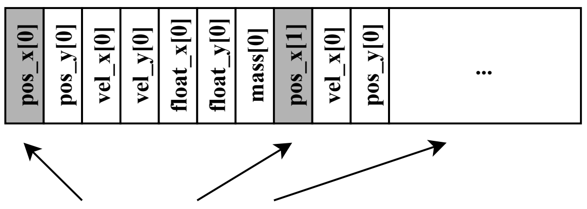

FIGURE 2.1: Data access in AOS data layout

Different from AOS, data is not stored contiguously in units of objects in SOA, instead it is stored in units of fields. Same fields of different objects are stored together in one array on GPU memory. Listing 2.2 shows an implementation of using SOA data layout in a same program with the program in Listing 2.1 and figure 2.2

LISTING 2.2: A C++ program with arrays of fields (SOA)

```
float Body_pos_x[1000];
float Body_pos_y[1000];
float Body_vel_x[1000];
float Body_vel_y[1000];
float Body_force_x[1000];
float Body_force_y[1000];
float Body_mass[1000];
```

shows the according data access example. As shown, for the array `bodies` above, it is disassembled into seven different arrays by the fields of class `Body`.

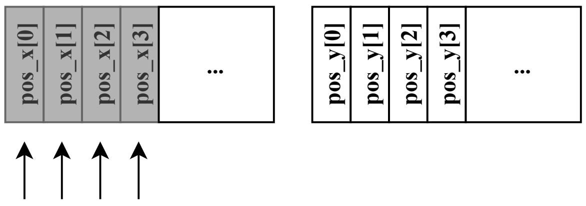

FIGURE 2.2: Data access in SOA data layout

Although using SOA data layout makes programs less readable, it can speed up HPC applications by several factors compared to a traditional AOS layout [1, 11, 14]. As shown in figure 2.1 and figure 2.2, data access in SOA is contiguous while data access in AOS is stridden. Since current NVIDIA GPU can coalesce data loads in contiguous 128-byte into one data load, fewer vector loads are required to cover all data access of field `pos_x` in SOA than in AOS. This improves the performance of the program since fewer loads means less consumption in access time.

2.2 N-Body simulation

N-Body is a 2D particle simulation. It simulates movements of a large number of bodies. Each body has position, velocity and mass. Every iteration of the simulation computes force between every two bodies according to Newton's theory of gravity, by advancing the simulation by a small period of time, the new velocity and position of the bodies are updated accordingly.

2.3 Foreign Function Interface

Sanajeh is a DSL run in Python but since Python code cannot run on the GPU, the core parts of Sanajeh code which run on the GPU are compiled into C++/CUDA. These parts are then called from the other parts which are written in Python. A mechanism is needed to build a bridge between the two different language.

Foreign Function Interface is a mechanism used to call functions written in another language. Sanajeh uses `CFFI` [10], which is a Python library, to call C++ function from Python. We choose the "in-line", "ABI mode" provided by the `CFFI` library. Listing 2.3 shows a example of using `CFFI`. Declarations like function prototypes are written in "$C - likedeclarations$", and the path of the compiled shared library (in Sanajeh, a ".so" file) is written in "$libpath$". By this, users can call C++ function directly through "$lib.function\_name$".

LISTING 2.3: Example of using `CFFI`, “in-line”. “ABI mode”

```
import cffi

ffi = cffi.FFI()
ffi.cdef("C-like declarations")
lib = ffi.dlopen("libpath")
```

Such code for using `CFFI` is hidden by Sanajeh API (section 5.1).

Chapter 3

Problem statement

DynaSOAr provides a choice for programmers to write OOP GPGPU programs, but programmers must adhere to a certain coding style (section 3.1) and follow certain coding conventions when using DynaSOAr (section 3.2). Also, DynaSOAr is inefficient for specific programs (section 3.3).

3.1 Distinguish device code and host code

In GPGPU programs, code is separated into two parts: **Host code** and **Device code**. Host code runs on the host (CPU), includes code that invokes the GPU kernel, while device code runs on the device (GPU).

In CUDA programs, users have to write "*__device__*" keywords before functions to specify that these functions run on the GPU. Since DynaSOAr is a CUDA framework, programmers must annotate functions and certain fields with such modifier. Listing 3.1 shows an example of a part of the functions in an N-Body simulation.

LISTING 3.1: "__device__" keywords before function declarations in an N-Body simulation.

```
__device__ void Body::apply_force(Body* other) {
  // Update 'other'.
  if (other != this) {
    float dx = pos_x_ - other->pos_x_;
    float dy = pos_y_ - other->pos_y_;
    float dist = sqrt(dx*dx + dy*dy);
    float F = kGravityConstant * mass_ * other->mass_
        / (dist * dist + kDampeningFactor);
    other->force_x_ += F*dx / dist;
    other->force_y_ += F*dy / dist;
  }
}

__device__ void Body::update() {
  vel_x_ += force_x_*kDt / mass_;
  vel_y_ += force_y_*kDt / mass_;
  pos_x_ += vel_x_*kDt;
  pos_y_ += vel_y_*kDt;

  if (pos_x_ < -1 || pos_x_ > 1) {
    vel_x_ = -vel_x_;
  }

  if (pos_y_ < -1 || pos_y_ > 1) {
    vel_y_ = -vel_y_;
  }
}
```

Such annotations can be annoying to programmers when there are a large number of device functions in the program. Even one miss of such keywords can cause fatal errors.

3.2 Pre-declarations for fields

DynaSOAr requires user to pre-declare all the fields of a class that resides on the device (we call these classes "device classes") through a special syntax (Listing 3.2).

LISTING 3.2: Field pre-declarations (DynaSOAr) in an N-Body simulation

```
class Body:public AllocatorT::Base {
public:
    declare_field_types(Body, float, float, float, float, float, float,
        float)
private:
    Field<Body, 0> pos_x;
    Field<Body, 1> pos_y;
    Field<Body, 2> vel_x;
    Field<Body, 3> vel_y;
    Field<Body, 4> force_x;
    Field<Body, 5> force_y;
    Field<Body, 6> mass;
}
```

LISTING 3.3: Ordinary C++ field declarations (equiv. to Fig. 1)

```
class Body {
private:
    float pos_x;
    float pos_y;
    float vel_x;
    float vel_y;
    float force_x;
    float force_y;
    float mass;
}
```

Such syntax is different from ordinary C++ (Listing 3.3). It is likely to make mistake when declaring field and it types separately.

3.3 Inefficiency of nested objects

There are some specific programs that DynaSOAr cannot run efficiently. Programs with nested object are among them. In this section we introduce what is "nested object" (section 3.3.1), why we need "nested object" and discuss why it is inefficient in DynaSOAr (section 3.3.3).

3.3.1 Definition of nested object

Definition 3.3.1 (Nested object) *A nested object is an object that is exclusively stored in another object. By exclusive, we mean the object is not shared by other objects.*

Listing 3.4 shows a DynaSOAr sample program with nested object. It defines a `Vector` class with fields `x` and `y` and uses fields with `Vector` type in class `Body`. Object `pos`, `vel` and `force` are nested objects.

LISTING 3.4: DynaSOAr program with nested object)

```
class Vector:public AllocatorT::Base {
private:
    float x;
    float y;
}

class Body:public AllocatorT::Base {
private:
    Vector pos;  //Vector* pos
    Vector vel;  //Vector* vel
    Vector force;  //Vector* force
    float mass;
}
```

3.3.2 Advantages of nested objects: code reuse

Program with nested object is easier to implement and extend. Use the code in Listing 3.4 as an example.

- `Implement`: When we use a `Vector` class, the similar behavior in `pos`, `vel` and `force` can be defines together. High code reusability means easiness in implementation.
- `Extend`: Suppose we need to modify the dimension from 2D to 3D, an extension on program with "nested object" will be easy since we just need to add a `z` field in `Vector` class. But without "nested object", we need to add `pos_x`, `pos_y` and `pos_z` and modify all the code related in `Body` class.

3.3.3 Disadvantages of nested objects: inefficiency

There the following three ways in DynaSOAr to handle nested object:

- `Pointer`: If we use pointer for these nested object fields (as shown in the comments in Listing 3.4), the space of these fields is allocated dynamically and DynaSOAr cannot ensure that these space are arranged in SOA data layout which obviously influences the performance.
- `Structure`: DynaSOAr will allocate such fields in a Structure Of Arrays Of Structures (SOAOS) data layout. Figure 3.1 shows how data is accessed in DynaSOAr in this data layout. Above is the data loaded and below is the data needed. Although all `pos` `pos` fields are stored contiguously on the memory, when the exact value of a field in `pos` field is accessed, the whole structure is loaded from the memory. That means that if we access `x` and `y` once for each sequentially, the whole structure is loaded twice. The extra load will be a waste of resource.
- `Flattened fields`: To flatten these fields is to avoid using objects as fields in classes. Programmers needs to write programs like Listing 3.3. DynaSOAr makes sure these fields are allocated on GPU memory in a most efficient data layout but this restricts the level of abstraction of programs. This makes programs complex (more fields), redundant (`x` and `y` fields have similar behavior) and unreusable (similar behavior in `x` and `y` fields need to be redefined for different fields).

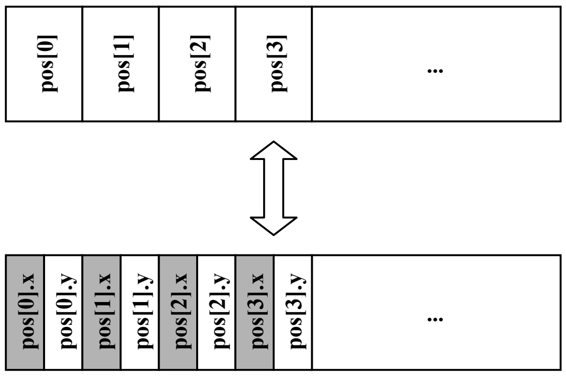

FIGURE 3.1: Data access for structured nested object

Since all these three choices are not ideal for handling nested object, for now DynaSOAr can only run programs with nested object inefficiently.

Chapter 4

Proposal

This chapter gives out the proposal of Sanajeh, explains how Sanajeh solves the problems in DynaSOAr which is explained in chapter 3. Solutions include an algorithm for distinguishing host code and device code (section 4.1), an algorithm for processing device code (section 4.2) and an algorithm for supporting nested object (section 4.3).

4.1 An algorithm for distinguishing host code and device code

Device code and host code are written together by users, while the former runs on the GPU and the latter runs on the CPU. Sanajeh's goal is to avoid writing extra keywords like described in section 3.1 to annotation whether code runs on the GPU or the CPU.

Listing 4.1 shows a part of a Sanajeh program. `Body` is a class whose objects are created on the GPU, this means every function of class `Body` is a device function. `run_bodies` is a function invokes the functions in class `Body`, but it is a host function itself that runs on the CPU.

Since there is no syntax to distinguish host functions and device functions in class, we proposed an algorithm, `CallGraphAnalyzer` (section 5.2), to detect all device code and mark them. `CallGraphAnalyzer` only requires the declaration of device classes. It will track all device code through the calling relationships between functions and the using of variables.

4.2 An algorithm for processing device code

Python code cannot run on the GPU directly, therefore we have to compile Python device code into C++/CUDA code. During the compilation, we collect the information of classes and its fields so there is no need for users to pre-declare all of the field types (section 3.2). We propose an algorithm, `Py2Cpp` (section 5.4), to process Python device code and generate C++/CUDA code. The C++/CUDA code generated are then compiled into a shared library using the Nvidia CUDA Compiler (NVCC), and called in host code through FFI (section 2.3)

Listing 4.2 and Listing 4.3 show the compiled code of the device code in Listing 4.1. Noticed function `run_bodies` is not compiled since it is a host function. In the compiled code, the syntax needs for pre-declaration and the keywords for annotation of device functions are generated automatically.

LISTING 4.1: A part of a Sanajeh program

```
class Body():
    pos_x: float
    pos_y: float
    vel_x: float
    vel_y: float
    force_x: float
    force_y: float
    mass: float

    def compute_force(self):
        self.force_x = 0.0
        self.force_y = 0.0
        DeviceAllocator.device_do(Body, Body.apply_force, self)

    def apply_force(self, other: Body):
        if other is not self:
            dx: float = self.pos_x - other.pos_x
            dy: float = self.pos_y - other.pos_y
            dist: float = math.sqrt(dx * dx + dy * dy)
            f: float = kGravityConstant * self.mass * other.mass / (dist *
                dist + kDampeningFactor)
            other.force_x += f * dx / dist
            other.force_y += f * dy / dist

    def body_update(self):
        self.vel_x += self.force_x * kDt / self.mass
        self.vel_y += self.force_y * kDt / self.mass
        self.pos_x += self.vel_x * kDt
        self.pos_y += self.vel_y * kDt
        if self.pos_x < -1 or self.pos_x > 1:
            self.vel_x = -self.vel_x
        if self.pos_y < -1 or self.pos_y > 1:
            self.vel_y = -self.vel_y

def run_bodies():
    for x in range(itr):
        PyAllocator.parallel_do(Body, Body.compute_force)
        PyAllocator.parallel_do(Body, Body.body_update)
```

LISTING 4.2: Compiled header file of the program in Listing 4.1

```
class Body : public AllocatorT::Base {
    public:
        declare_field_types(Body, float, float, float, float, float, float
            , float)
    private:
        Field<Body, 0> pos_x;
        Field<Body, 1> pos_y;
        Field<Body, 2> vel_x;
        Field<Body, 3> vel_y;
        Field<Body, 4> force_x;
        Field<Body, 5> force_y;
        Field<Body, 6> mass;
    public:
        __device__ void compute_force();
        __device__ void apply_force(Body* other);
        __device__ void body_update();
};
```

LISTING 4.3: Compiled source file of the program in Listing 4.1

```
__device__ void Body::compute_force() {
    this->force_x = 0.0;
    this->force_y = 0.0;
    device_allocator->template device_do<Body>(&Body::apply_force, this);
}

__device__ void Body::apply_force(Body* other) {
    if (other != this) {
        float dx = this->pos_x - other->pos_x;
        float dy = this->pos_y - other->pos_y;
        float dist = sqrt((dx * dx) + (dy * dy));
        float f = ((kGravityConstant * this->mass) * other->mass) / ((dist
            * dist) + kDampeningFactor);
        other->force_x += (f * dx) / dist;
        other->force_y += (f * dy) / dist;
    }
}

__device__ void Body::body_update() {
    this->vel_x += (this->force_x * kDt) / this->mass;
    this->vel_y += (this->force_y * kDt) / this->mass;
    this->pos_x += this->vel_x * kDt;
    this->pos_y += this->vel_y * kDt;
    if (this->pos_x < -1 || this->pos_x > 1) {
        this->vel_x = -this->vel_x;
    }
    if (this->pos_y < -1 || this->pos_y > 1) {
        this->vel_y = -this->vel_y;
    }
}
```

4.3 An algorithm for supporting nested object

To support programs with nested object which is inefficient in DynaSOAr (section 3.3), we propose an algorithm, `FieldExpander` (section 5.3), to automatically expand these fields to ones with basic types and to inline all functions related to these objects.

Listing 4.4 is a program similar to the one in Listing 4.1 but uses nested object. In `Body` class, `Vector` class type is used for field `pos`, `vel` and `force`. The functions in `Vector` class are called in `Body` class, for example in `compute_force` function, `minus` function is called in line 44 to compute the displacement vector between two points. Our goal is to dissemble `pos`, `vel` and `force` into `pos_x`, `vel_x`, `force_x`, `pos_y`, `vel_y` and `force_y`, inline all `Vector` class's member function in class `Body` and finally generate code close to the code in Listing 4.1.

LISTING 4.4: Use nested object in the program in Listing 4.1

```python
class Vector:
    x: float
    y: float

    def __init__(self, x_, y_):
        self.x = x_
        self.y = y_

    def add(self, other: Vector) -> Vector:
        self.x += other.x
        self.y += other.y
        return self

    def minus(self, other: Vector) -> Vector:
        return Vector(self.x - other.x, self.y - other.y)

    def multiply(self, multiplier: float) -> Vector:
        return Vector(self.x * multiplier, self.y * multiplier)

    def divide(self, divisor: float) -> Vector:
        return Vector(self.x / divisor, self.y / divisor)

    # Distance from origin
    def dist_origin(self) -> float:
        return math.sqrt(self.x * self.x + self.y * self.y)

    def to_zero(self) -> Vector:
    self.x = 0.0
    self.y = 0.0
    return self

class Body:
    pos: Vector
    vel: Vector
    force: Vector
    mass: float

    def compute_force(self):
        self.force.to_zero()
        DeviceAllocator.device_do(Body, Body.apply_force, self)

    def apply_force(self, other: Body):
        if other is not self:
            d: Vector = self.pos.minus(other.pos)
            dist: float = d.dist_origin()
            f: float = kGravityConstant * self.mass * other.mass / (dist *
                 dist + kDampeningFactor)
            other.force.add(d.multiply(f).divide(dist))

    def body_update(self):
        self.vel.add(self.force.multiply(kDt).divide(self.mass))
        self.pos.add(self.vel.multiply(kDt))

        if self.pos.x < -1 or self.pos.x > 1:
            self.vel.x = -self.vel.x
        if self.pos.y < -1 or self.pos.y > 1:
            self.vel.y = -self.vel.y
```

Chapter 5

Implementation

This chapter explained the implementation datails of Sanajeh, includes the introduction of Sanajeh's API (section 5.1), a program for detecting device code (section 5.2), a program for expanding nested object (section 5.3) and a compiler for compiling Python device code into C++/CUDA code (section 5.4).

5.1 Sanajeh's API

Sanajeh provides API for device code and host code separately.

5.1.1 Device API

Device code is not run by the Python interpreter, instead they are compiled into according C++/CUDA code by `Py2Cpp`. Therefore, API for device code is only used for recognizing purpose. The current API for device code is shown below:

- `device_do(cls, func, *args)`: Run function `func` on all objects of class `cls` sequentially with the arguments `*args`. Here `func` has to be a function declared in the device classes.
- `array_size(arr, s)`: Define the size `s` of an array `arr`. `arr` has to be a Python list that contains elements with the same type. Sanajeh computes the total size of the `arr` by the size of each element and array size `s` and then allocates it on GPU memory. Listing 5.1 shows a using example. `cells` is a Python list contains objects with type `Cell`. `array_size` is called to define the size of `cells` is 1000. Sanajeh then allocates memory of 1000 `Cell` objects on GPU memory.
- `new(cls, *args)`: Create an object dynamically with type `cls` by calling the constructor function with arguments `*args` .
- `destroy(obj)`: Free the memory of `obj` on the device.
- **Random API**: Random API is provided for random number generation, for example the `rand_init` function and the `rand_uniform` function. Corresponding to the cuRAND library.
- **Math API**: Math API is provided for Math computation, for example the `sqrt` function. Corresponding to the CUDA Math API.

LISTING 5.1: An example of using `array_size`

```
cells: List[Cell]
DeviceAllocator.array_size(cells, 1000)
```

5.1.2 Host API

Host code is executed by the Python interpreter. Sanajeh provides the following host API:

- `initialize()`: Load the above shared library to Python FFI module.
- `device_class(*clss)`: Specify the device classes through variadic arguments `*clss`.
- `parallel_do(cls, func, *args)`: Run function `func` on all objects of class `cls` parallelly with the arguments of `func`. Here `func` has to be a function declared in the device classes. Notice that there is no `parallel_do` API in device API (section 5.1.1), since device code is executed in units of one thread and parallelism cannot be applied further more.
- `parallel_new(cls, object_num)`: Create objects of a class `cls` on device memory with the number `object_num`.
- `do_all(cls, func)`: Run function `func` on copies of all objects of class `cls` sequentially. Here `func` is a function which receives an object as an argument. The object is a copy of an object of class `cls`. Functions like `print` can be used in `func`.

5.2 CallGraphAnalyzer

`CallGraphAnalyzer` is designed to free users of Sanajeh from explicitly specifying all host code and device code. It uses the abstract syntax tree (AST) of Python source code as input, tracks calling relationships between functions and using of variables, then creates a `CallGraph` data structure (section 5.2.2). After user specifies the device classes, by analyzing the `CallGraph`, all device code will be marked.

5.2.1 Definition of Sanajeh's device code

We define the device code of Sanajeh as belwo.

Definition 5.2.1 (Device code) *Code for declaring classes whose objects are created on the GPU and code invoked in device code is device code.*

5.2.2 CallGraph data structure

`CallGraph` is a data structure which has three kinds of nodes: `ClassNode`, `FunctionNode` and `VariableNode`:

- `ClassNode`: Represents a class. There are four sets in this node, each stores functions declared in that class, functions called in that class, variables declared in that class and variable used in that class. Each element in these sets is stored as either `FunctionNode` or `VariableNode`. Sanajeh does not support nested class.
- `FunctionNode`: Represents a function. Similar to `ClassNode`, `FunctionNode` has sets to stores variables declared and used in the function. Sanajeh does not support nested functions too.

- `VariableNode`: Represents a variable. Name and type of the variable is stored in `VariableNode`. Notice a global variable cannot be used both in device and host code.

5.2.3 Marking of device data

After the `CallGraph` structure is created, devices code will be marked by tracking the declaration, calling and using relationships between the nodes. `CallGraphAnalyzer` first marks all device class `ClassNodes` and all `FunctionNodes` and `VariableNodes` in them as device nodes. Then all nodes used in device nodes are marked recursively until every device node has no unmarked nodes.

5.3 FieldExpander

`FieldExpander` is a program used to expand nested object into dissembled fields. Its input is Python device code with nested object and output is also Python device code but without nested object. We add an underscore between the nested object's name and its fields to create new fields. For example, for a `pos` field with `Vector` type, while `Vector` has field `x` and `y`, then the new fields will be `pos_x` and `pos_y`.

It is simple to rewrite the names of the fields but there are challenges when inlining all `Vector` class functions. We divide the inlining process into 3 steps and implement three programs for each.`Normailzer` for normolizing nested expressions (section 5.3.1), `Inliner` for inlining non-nested functions (section 5.3.2) and `Eliminator` for elminating all nested object and creating generate new fields according to the object's fields (section 5.3.3). Each of these three is implemented as a child class of `ast.NodeTransformer`, which is provided by the `ast` Python library. Listing 5.2 shows an input example and Listing 5.4 is the target output for it, using the definition in Listing 5.3.

LISTING 5.2: A sample expression in program with nested object

```
self.vel.add(self.force.multiply(kDt).divide(self.mass))
```

LISTING 5.3: Definition of `Vector` class

```
class Vector:

    def __init__(self, x_: float, y_: float):
        self.x: float = x_
        self.y: float = y_

    def add(self, other: Vector) -> Vector:
        self.x += other.x
        self.y += other.y
        return self
        return self

    def multiply(self, multiplier: float) -> Vector:
        return Vector(self.x * multiplier, self.y * multiplier)

    def divide(self, divisor: float) -> Vector:
        return Vector(self.x / divisor, self.y / divisor)
```

LISTING 5.4: Target output of the code in Listing 5.2

```
self.vel_x += self.force_x * kDt / self.mass
self.vel_y += self.force_y * kDt / self.mass
```

5.3.1 Normalizer

`Normalizer` normalizes nested expressions so that we call easily inline each call expressions later. Normalizer processes call nodes in all of the AST nodes and follows the following rules:

1. If the argument node of the current call node is nested, visit it, since they are always be evaluated first.
2. If the argument node of the current call node is not nested but an expression, refer its type from the annotation of the function called in the argument node and create an annotate assign node that assigns the whole expression to a new variable with the type referred. Then visit the callee of current node and after visiting replace the callee with a new AST Name node which represents the new created variable.
3. If the argument node of the current call node is a variable, and current node calls another node, create an annotate assign node that assigns the expression of current call node to a new variable and annotate it with the return type of the callee of current call node. Replace current node with a new AST Name node which represents the new created variable. Then visit the next call node.
4. If the argument node of the current call node is a variable, and current node does not call another node, return the call node itself.

Listing 5.5 and Listing 5.6 show the processing procedure of Normalizer on the code in Listing 5.2. At first `self.force.multiply(kDt)` is replaced by `__auto_v0` then `__auto_v0.divide(self.mass)` is replaced by `__auto_v1`.

LISTING 5.5: Process the first call node in the argument node

```
__auto_v0: Vector = self.force.multiply(kDt)
self.vel.add(__auto_v0.divide(self.mass))
```

LISTING 5.6: Normalizer Output

```
__auto_v0: Vector = self.force.multiply(kDt)
__auto_v1: Vector = __auto_v0.divide(self.mass)
self.vel.add(__auto_v1)
```

Finally all call nodes are not nested and all arguments in call nodes are variables.

5.3.2 Inliner

Inliner inlines all function calls from nested objects. It processes call nodes. It just replaces the variable name nodes in the function implementation with the name nodes of the arguments, then replaces the function calling with the rewritten function implementation. Listing 5.7 shows the result of inliner on processing the code in Listing 5.6.

LISTING 5.7: Inliner Output

```
__auto_v0: Vector = Vector((self.force.x * kDt), (self.force.y * kDt))
__auto_v1: Vector = Vector((__auto_v0.x / self.mass), (__auto_v0.y / self.
    mass))
self.vel.x += __auto_v1.x
self.vel.y += __auto_v1.y
```

5.3.3 Eliminator

Eliminator eliminates all objects with the types which nested objects has used. Not only nested objects is eliminated but also part of the new created variables is eliminated, since some function calls may return types which nested objects has used. Eliminator also rewrites reference to the fields of nested objects. Eliminator processes all kinds of AST nodes under the following rules:

1. For an annotation assign node, if it annotates the variable with a type which nested object has used, replace this node with multiple annotation nodes that assigns each field of the nested object to the according values with according types in the value node itself.

2. For an attribute node, if its value represents a nested object, then replace this node with a name node. The name node's id is the old value connects the old attribute with an underscore.

For example, Listing 5.8 shows the output of Eliminator using the code in Listing 5.7 as input. The annotation assignment of `__auto_vo` is replaced by the assignments of `__auto_vo_x` and `__auto_vo_y`, using the values `self.force_x * kDt` and `self.force_y * kDt` which are passed as arguments in the constructor functions and are used for initializing the `x` and `y` fields of `__auto_vo`. The reference to `self.vel.x` is also changed to `self.vel_x`, since `vel` is a nested object and has been dissembled into `vel_x` and `vel_y`.

LISTING 5.8: Eliminator Output

```
__auto_v0_x: float = (self.force_x * kDt)
__auto_v0_y: float = (self.force_y * kDt)
__auto_v1_x: float = (__auto_v0_x / self.mass)
__auto_v1_y: float = (__auto_v0_y / self.mass)
self.vel_x += __auto_v1_x
self.vel_y += __auto_v1_y
```

5.4 Py2Cpp

`Py2Cpp` is a compiler which compiles Python device code to C++/CUDA code. It uses the check result of `CallGraphAnalyzer` (section 5.2) to transform the Python AST of device code to C++ AST code (section 5.4.1). Then C++ code will be builded from C++ AST (section 5.4.2)

5.4.1 Transform Python AST to C++ AST

The transform uses the Python `ast` library. The library provides is designed following Visitor Pattern. There is a visitor function for every kind of node in the AST. Behavior when visiting the node is defined in it. `Py2Cpp` checks the mark results of `CallGraphAnalyzer` when visiting class, function and variable nodes. If the related `CallGraph` node of a Python AST node is marked as device, all child node of that node will be transformed into C++ AST nodes. Finally, a C++ AST of all device code is constructed.

Data type is an important issue during the transformation. Type annotations are supported from Python 3.6. Although Python runtime does not enforce function and variable type annotations, since type information is important for memory allocation, Sanajeh requires users to explicitly write type hints for device data. There is

LISTING 5.9: Function for calling `parallel_new` on class `Body`

```
extern "C" int parallel_new_Body(int object_num){
        allocator_handle->parallel_new<Body>(object_num);
        return 0;
}
```

a type converter in `Py2Cpp`. For now, it only supports `bool`, `int` and `float` data types and the array type of these types.

5.4.2 Build C++ code

After C++ AST is generated, C++ code will be generated from the C++ AST. We generate two files from C++ AST, one is ".h" header file and another is ".cu" source file. The "*__device__*" keyword and other syntax will be add during this process. Other syntax includes:

- **Pre-declarations of fields**: Pre-declarations of fields are generated from class AST nodes in C++ AST. Sanajeh collects field information from all of the nodes for annotate assignments under class nodes and generates code in Listing 3.2 automatically.
- **code for API callings**: API calling in DynaSOAr uses C++ templates to pass argument like class and function. Although Python has mechanism to pass class and function as an argument, templates cannot be called through FFI (section 2.3) except for functions. We generate a function for each class. The function includes C++ template, which is used to call DynaSOAr API. For example, Listing 5.9 shows this the function for calling `parallel_new` on class `Body`.
- **code for callback**: In Sanajeh's API (section 5.1), there is a `do_all` function. This function receives a Python function, say `PF`, while `PF` receives a Python object as a parameter. The constructor function of the class, say `CF`, will be included together with `PF` in such a lambda expression:

  $$lambda\{*args\} : PF(CF(\{*args\}))$$

  $\{*args\}$ is the fields of the device class. This lambda expression `lexp` is passed to C++ as a callback function. In c++, there is a `do_all` function for every class that receives `lexp` and passes it to another function `_do` using DynaSOAr API. In `_do` function, all fields of the class are passed to `lexp`. As a result, `lexp` is called multiple times in C++ territory until copies of all device objects are created by `CF` and passed to `F`. Finally, the fields of those copy objects are accessed in `F`. Example code for `_do` and `do_all` is shown in Listing 5.10. `Py2Cpp` collects the information of fields of device classes then generates such code automatically.

LISTING 5.10: `_do` and `do_all` function of class `Body`

```
void Body::_do(void (*l)(float, float, float, float, float, float, float))
    {
        l(this->pos_x, this->pos_y, this->vel_x, this->vel_y, this->
            force_x, this->force_y, this->mass);
}

extern "C" int Body_do_all(void (*l)(float, float, float, float, float,
    float, float)){
        allocator_handle->template device_do<Body>(&Body::_do, l);
        return 0;
}
```

Chapter 6

Case study

This chapter explained the case study of evaluating Sanajeh. We use an N-Body simulation as an sample program. We evaluate two factors: the complexity of a program (section 6.1) and the efficiency of running nested object programs. (section 6.2).

6.1 Complexity of program

Since we can only write OOP GPGPU programs in either Sanajeh or DynaSOAr, we do not take other languages into consideration. We implement N-Body programs similarly in Sanajeh and DynaSOAr, both in OOP style with and without nested object. Each has a single class `Body`. Listing 4.1, Listing 4.2 and Listing 4.3 show the part of our implementation in Sanajeh and DynaSOAr. Figure 6.1 shows the total number of words needed for a N-Body simulation program in Sanajeh and DynaSOAr.

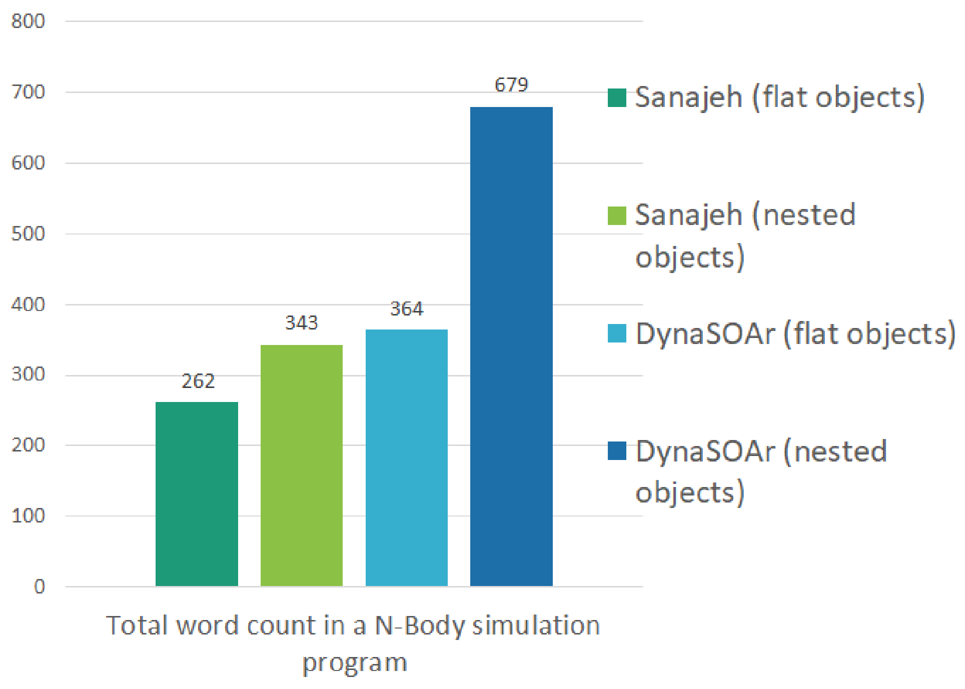

FIGURE 6.1: Total word count in a N-Body simulation program (lower is better)

As shown, in Sanajeh there are few words needed for declaring functions and fields of class `Body`. Sanajeh uses 262 words with flat objects and uses 343 words with nested objects, while DynaSOAr uses 364 words with flat objects and uses 679 words with nested objects. Sanajeh needs less words than DynaSOAr not only because it simplifies the field declaration and omits keywords of defining device functions, but also because it uses Python rather than C++. Previous research has shown that Python takes half as much time as writing a same program than C++ and the result program is also half as long [9]. Also, the reason why programs with nested objects are using more words can be considered as the more words using in declaring the class of nested objects.

6.2 Efficiency in running programs with nested objects

For a program with nested objects, we evaluate the performance in three different processing methods. The first one is manually flattening all nested objects, which means fields of all objects have only types with basic types. This has the best performance since all objects can be allocated in SOA data layout by DynaSOAr on GPU memory. The second one is using `FieldExpander` (section 5.3), which is the algorithm Sanajeh uses for processing nested objects. By `FieldExpander`, nested objects are flattened automatically, and this should have same performance with the first method in ideal case. The third one is remaining all nested objects unflattened. We only take those programs that implements nested object in a structure data layout into consideration since using pointer means data is allocated dynamically by the default CUDA allocator, not DynaSOAr (described in section 3.3.3). In this case the performance relies on the CUDA allocator, which has worse performance than DynaSOAr [13].

We run 100 iterations and measure the average execution time of one iteration in each. In each iteration program computes the force between every two bodies and updates the position of all bodies according to the force and a constant period. We use a NVIDIA TITAN Xp GPU with 12 GB device memory and an OS of Ubuntu 18.04.4. For the compilation of device code, we use NVCC (-O3) with CUDA Toolkit 10.1 and for the execution of host code we use Python 3.7.4. Figure 6.2 shows the result.

We take the manually flattened program as reference and compute the ratio of the other two to it. As shown, the automatically flattened program has nearly the same performance with the manually flattened program while the unflattened program takes more time for execution. As the objects number becomes large, the ratio of the execution time of unflattened program to manually flattened becomes large. This means there is more performance loss when using unflattened nested objects compared to the origin one which has no nested objects. It is conceivable that Sanajeh can process program with nested object efficiently.

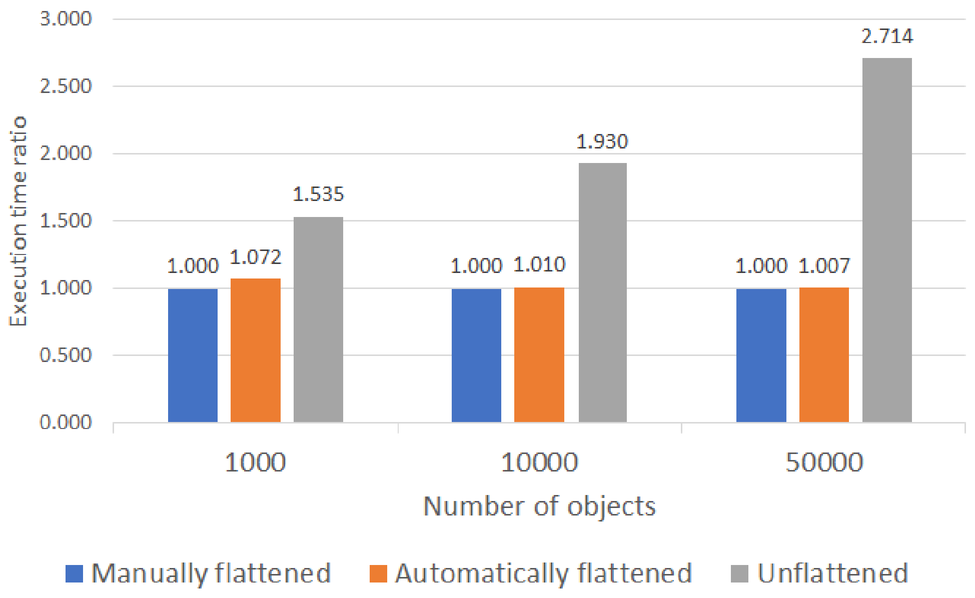

FIGURE 6.2: The ratio of average execution time for one iteration in running a N-Body simulation program in Sanajeh (lower is better)

Chapter 7

Related works

There have been high-level DSLs for GPGPU programming. Such as *SPOC* [2], which uses stream processing with OCaml, and *Ikra* [5], which is a data-parallel extension to Ruby. In Python, Klöckner et al. designed *PyCUDA* and *PyOpenCL* [4], two Python libraries for GPU computing based on CUDA. They are designed for experienced CUDA users and require users to write code in CUDA syntax as a string. This is not friendly to a Python programmer who does not understand C++/CUDA syntax. Another library for GPU computing is called *CUPY* [6]. Its syntax is close to *NumPy* [7], a widely-used Python library for data computing. It provides the same functionality as NumPy but executes on a GPU.

Although the above libraries have good performance in most of the cases in GPGPU programming by Python, they are not designed and optimized for OOP. In Python, a list of objects is stored as an Array of Structures (AOS) [3, 8]. Previous works [1, 11, 14] have shown that switching to a Structure of Arrays (SOA) data layout can speed up HPC applications by several factors compared to a traditional AOS layout. Therefore, traditional Python implementation of a list of objects is not a good choice for GPU programs. Our work supports objects including nested object used in GPGPU programs. We use DynaSOAr [13] as memory allocator, which allocates memory dynamically with SoA performance characteristics.

Chapter 8

Conclusion

OOP style has been regarded as inefficient to be applied in GPGPU programs, which take performance into consideration. DynaSOAr has proved that OOP programs can also achieve competitive performance under a model called SMMO. Programs with nested object are common use cases of OOP but It is either inefficient or hard to write GPGPU programs with nested object in DynaSOAr. programmers must also adhere to a certain coding style and follow certain coding conventions when using DynaSOAr, including the pre-declarations of fields and the annotations for device code.

We designed and implemented a DSL for GPGPU programming with Python objects called Sanajeh to improve these issues in DynaSOAr. Sanajeh frees users from expressly annotate whether code runs on the GPU or the CPU under an algorithm, `CallGraphAnalyzer`, which automatically distinguishes host code and device code. Sanajeh provides an algorithm, `Py2Cpp`, which compiles Python code into C++/CUDA code, to liberate GPGPU programmers from following the conventions in DynaSOAr and enable users to program GPGPU programs in normal Python style. Sanajeh does not lose performance when processing programs with nested object under an algorithm, `Fieldtransformer`, which automatically expands nested object to ones with basic types and inlines all functions related to these objects in device code.

We evaluated Sanajeh by comparing Sanajeh and DynaSOAr on the complexity of a similar program and by comparing the efficiency of using three different methods of dealing with a program with nested objects in Sanajeh. It shows that Sanajeh program is easier to write than DynaSOAr and Sanajeh can process programs with nested objects efficiently.

Bibliography

[1] Paul Besl. "A case study comparing AoS (Arrays of Structures) and SoA (Structures of Arrays) data layouts for a compute-intensive loop run on Intel Xeon processors and Intel Xeon Phi product family coprocessors". In: *Intel Article* 392271 (2015).

[2] Mathias Bourgoin, Emmanuel Chailloux, and Jean-Luc Lamotte. "SPOC: GPGPU programming through stream processing with OCaml". In: *Parallel Processing Letters* 22.02 (2012), p. 1240007.

[3] Michael T Goodrich, Roberto Tamassia, and Michael H Goldwasser. *Data structures and algorithms in Python*. Wiley Hoboken, 2013.

[4] Andreas Klöckner et al. "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation". In: *Parallel Computing* 38.3 (2012), pp. 157–174.

[5] Hidehiko Masuhara and Yusuke Nishiguchi. "A data-parallel extension to Ruby for GPGPU: toward a framework for implementing domain-specific optimizations". In: *Proceedings of the 9th ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*. 2012, pp. 3–6.

[6] ROYUD Nishino and Shohei Hido Crissman Loomis. "Cupy: A Numpy-compatible library for nvidia GPU calculations". In: *31st confernce on neural information processing systems* (2017), p. 151.

[7] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[8] Dusty Phillips. *Python 3 object-oriented programming*. Packt Publishing Ltd, 2015.

[9] Lutz Prechelt. "An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl". In: *IEEE Computer* 33.10 (2000), pp. 23–29.

[10] Armin Rigo and Maciej Fijalkowski. *CFFI documentation*. 2012.

[11] M Springer and H Masuhara. "Ikra-Cpp: A C++/CUDA DSL for Object-Oriented Programming with Structure-of-Arrays Layout (WPMVP'18)". In: *ACM, Article* 6.9 (2018).

[12] Matthias Springer. "Memory-Efficient Object-Oriented Programming on GPUs". PhD thesis. Tokyo Institute of Technology, 2019.

[13] Matthias Springer and Hidehiko Masuhara. "DynaSOAr: A Parallel Memory Allocator for Object-Oriented Programming on GPUs with Efficient Memory Access". In: *33rd European Conference on Object-Oriented Programming*. 2019.

[14] Robert Strzodka. "Abstraction for AoS and SoA layout in C++". In: *GPU computing gems Jade edition*. Elsevier, 2012, pp. 429–441.

Appendix A

Sanajeh sample program (N-Body)

A.1 Device code in Sanajeh

A.1.1 Code before expanding nested-objects

LISTING A.1: Device code of a N-Body simulation in Sanajeh

```
from __future__ import annotations

import math
from sanajeh import DeviceAllocator

kSeed: int = 45  # device
kMaxMass: float = 1000.0  # device
kDt: float = 0.01  # device
kGravityConstant: float = 4e-6  # device
kDampeningFactor: float = 0.05  # device


class Vector:

    def __init__(self, x_: float, y_: float):
        self.x: float = x_
        self.y: float = y_

    def add(self, other: Vector) -> Vector:
        self.x += other.x
        self.y += other.y
        return self

    def plus(self, other: Vector) -> Vector:
        return Vector(self.x + other.x, self.y + other.y)

    def subtract(self, other: Vector) -> Vector:
        self.x -= other.x
        self.y -= other.y
        return self

    def minus(self, other: Vector) -> Vector:
        return Vector(self.x - other.x, self.y - other.y)

    def scale(self, ratio: float) -> Vector:
        self.x *= ratio
        self.y *= ratio
        return self

    def multiply(self, multiplier: float) -> Vector:
        return Vector(self.x * multiplier, self.y * multiplier)

    def divide_by(self, divisor: float) -> Vector:
```

```python
        self.x /= divisor
        self.y /= divisor
        return self

    def divide(self, divisor: float) -> Vector:
        return Vector(self.x / divisor, self.y / divisor)

    # Distance from origin
    def dist_origin(self) -> float:
        return math.sqrt(self.x * self.x + self.y * self.y)

    def to_zero(self) -> Vector:
        self.x = 0.0
        self.y = 0.0
        return self


class Body:
    pos: Vector
    vel: Vector
    force: Vector
    mass: float

    def __init__(self, px: float, py: float, vx: float, vy: float, fx: float, fy: float, m: float):
        self.pos = Vector(px, py)
        self.vel = Vector(vx, vy)
        self.force = Vector(fx, fy)
        self.mass = m

    def Body(self, idx: int):
        DeviceAllocator.rand_init(kSeed, idx, 0)
        self.pos = Vector(2.0 * DeviceAllocator.rand_uniform() - 1.0,
                          2.0 * DeviceAllocator.rand_uniform() - 1.0)
        self.vel = Vector(0.0, 0.0)
        self.force = Vector(0.0, 0.0)
        self.mass = (DeviceAllocator.rand_uniform() / 2.0 + 0.5) * kMaxMass

    def compute_force(self):
        self.force.to_zero()
        DeviceAllocator.device_do(Body, Body.apply_force, self)

    def apply_force(self, other: Body):
        if other is not self:
            d: Vector = self.pos.minus(other.pos)
            dist: float = d.dist_origin()
            f: float = kGravityConstant * self.mass * other.mass / (dist * dist + kDampeningFactor)
            other.force.add(d.multiply(f).divide(dist))

    def body_update(self):
        self.vel.add(self.force.multiply(kDt).divide(self.mass))
        # self.vel.add(self.force.scale(kDt).divide_by(self.mass))
        self.pos.add(self.vel.multiply(kDt))
        # self.pos.add(self.vel.scale(kDt))

        if self.pos.x < -1 or self.pos.x > 1:
            self.vel.x = -self.vel.x
        if self.pos.y < -1 or self.pos.y > 1:
            self.vel.y = -self.vel.y


```

```
def kernel_initialize_bodies():
    DeviceAllocator.device_class(Body)


def _update():
    DeviceAllocator.parallel_do(Body, Body.compute_force)
    DeviceAllocator.parallel_do(Body, Body.body_update)
```

A.1.2 Code after expanding nested-objects

LISTING A.2: Device code of a N-Body simulation in Sanajeh (after expanding nested-object)

```
from __future__ import annotations
import math
from sanajeh import DeviceAllocator
kSeed: int = 45
kMaxMass: float = 1000.0
kDt: float = 0.01
kGravityConstant: float = 4e-06
kDampeningFactor: float = 0.05

class Body():
    pos_x: float
    pos_y: float
    vel_x: float
    vel_y: float
    force_x: float
    force_y: float
    mass: float

    def __init__(self, px: float, py: float, vx: float, vy: float, fx:
        float, fy: float, m: float):
        self.pos_x = px
        self.pos_y = py
        self.vel_x = vx
        self.vel_y = vy
        self.force_x = fx
        self.force_y = fy
        self.mass = m

    def Body(self, idx: int):
        DeviceAllocator.rand_init(kSeed, idx, 0)
        self.pos_x = ((2.0 * DeviceAllocator.rand_uniform()) - 1.0)
        self.pos_y = ((2.0 * DeviceAllocator.rand_uniform()) - 1.0)
        self.vel_x = 0.0
        self.vel_y = 0.0
        self.force_x = 0.0
        self.force_y = 0.0
        self.mass = (((DeviceAllocator.rand_uniform() / 2.0) + 0.5) *
            kMaxMass)

    def compute_force(self):
        self.force_x = 0.0
        self.force_y = 0.0
        DeviceAllocator.device_do(Body, Body.apply_force, self)

    def apply_force(self, other: Body):
        if (other is not self):
            d_x: float = (self.pos_x - other.pos_x)
            d_y: float = (self.pos_y - other.pos_y)
```

```
                dist: float = math.sqrt(((d_x * d_x) + (d_y * d_y)))
                f: float = (((kGravityConstant * self.mass) * other.mass) / ((
                    dist * dist) + kDampeningFactor))
                __auto_v0_x: float = (d_x * f)
                __auto_v0_y: float = (d_y * f)
                __auto_v1_x: float = (__auto_v0_x / dist)
                __auto_v1_y: float = (__auto_v0_y / dist)
                other.force_x += __auto_v1_x
                other.force_y += __auto_v1_y

    def body_update(self):
        __auto_v0_x: float = (self.force_x * kDt)
        __auto_v0_y: float = (self.force_y * kDt)
        __auto_v1_x: float = (__auto_v0_x / self.mass)
        __auto_v1_y: float = (__auto_v0_y / self.mass)
        self.vel_x += __auto_v1_x
        self.vel_y += __auto_v1_y
        __auto_v2_x: float = (self.vel_x * kDt)
        __auto_v2_y: float = (self.vel_y * kDt)
        self.pos_x += __auto_v2_x
        self.pos_y += __auto_v2_y
        if ((self.pos_x < (- 1)) or (self.pos_x > 1)):
            self.vel_x = (- self.vel_x)
        if ((self.pos_y < (- 1)) or (self.pos_y > 1)):
            self.vel_y = (- self.vel_y)

def kernel_initialize_bodies():
    DeviceAllocator.device_class(Body)

def _update():
    DeviceAllocator.parallel_do(Body, Body.compute_force)
    DeviceAllocator.parallel_do(Body, Body.body_update)
```

A.2 Translated device code in Sanajeh

LISTING A.3: Sanajeh host code (N-Body)

```
#include "sanajeh_device_code.h"

AllocatorHandle<AllocatorT>* allocator_handle;
__device__ AllocatorT* device_allocator;

__device__ Body::Body(float px, float py, float vx, float vy, float fx,
    float fy, float m) {
        this->pos_x = px;
        this->pos_y = py;
        this->vel_x = vx;
        this->vel_y = vy;
        this->force_x = fx;
        this->force_y = fy;
        this->mass = m;
}

__device__ Body::Body(int idx) {
        curandState rand_state;
        curand_init(kSeed, idx, 0, &rand_state);
        this->pos_x = (2.0 * curand_uniform(&rand_state)) - 1.0;
        this->pos_y = (2.0 * curand_uniform(&rand_state)) - 1.0;
        this->vel_x = 0.0;
        this->vel_y = 0.0;
        this->force_x = 0.0;
        this->force_y = 0.0;
        this->mass = ((curand_uniform(&rand_state) / 2.0) + 0.5) *
            kMaxMass;
}

__device__ void Body::compute_force() {
        this->force_x = 0.0;
        this->force_y = 0.0;
        device_allocator->template device_do<Body>(&Body::apply_force,
            this);
}

__device__ void Body::apply_force(Body* other) {
        if (other != this) {
                float d_x = this->pos_x - other->pos_x;
                float d_y = this->pos_y - other->pos_y;
                float dist = sqrt((d_x * d_x) + (d_y * d_y));
                float f = ((kGravityConstant * this->mass) * other->mass)
                    / ((dist * dist) + kDampeningFactor);
                float __auto_v0_x = d_x * f;
                float __auto_v0_y = d_y * f;
                float __auto_v1_x = __auto_v0_x / dist;
                float __auto_v1_y = __auto_v0_y / dist;
                other->force_x += __auto_v1_x;
                other->force_y += __auto_v1_y;
        }
}

__device__ void Body::body_update() {
        float __auto_v0_x = this->force_x * kDt;
        float __auto_v0_y = this->force_y * kDt;
        float __auto_v1_x = __auto_v0_x / this->mass;
        float __auto_v1_y = __auto_v0_y / this->mass;
        this->vel_x += __auto_v1_x;
        this->vel_y += __auto_v1_y;
        float __auto_v2_x = this->vel_x * kDt;
        float __auto_v2_y = this->vel_y * kDt;
        this->pos_x += __auto_v2_x;
        this->pos_y += __auto_v2_y;
        if (this->pos_x < -1 || this->pos_x > 1) {
                this->vel_x = -this->vel_x;
        }
        if (this->pos_y < -1 || this->pos_y > 1) {
                this->vel_y = -this->vel_y;
        }
}

void Body::_do(void (*pf)(float, float, float, float, float, float, float)
    ){
        pf(this->pos_x, this->pos_y, this->vel_x, this->vel_y, this->
            force_x, this->force_y, this->mass);
}

extern "C" int Body_do_all(void (*pf)(float, float, float, float, float,
    float, float)){
        allocator_handle->template device_do<Body>(&Body::_do, pf);
        return 0;
}

extern "C" int Body_Body_compute_force(){
        allocator_handle->parallel_do<Body, &Body::compute_force>();
        return 0;
```

```
}

extern "C" int Body_Body_body_update(){
        allocator_handle->parallel_do<Body, &Body::body_update>();
        return 0;
}

extern "C" int parallel_new_Body(int object_num){
        allocator_handle->parallel_new<Body>(object_num);
        return 0;
}

extern "C" int AllocatorInitialize(){
        allocator_handle = new AllocatorHandle<AllocatorT>(/*
            unified_memory= */ true);
        AllocatorT* dev_ptr = allocator_handle->device_pointer();
        cudaMemcpyToSymbol(device_allocator, &dev_ptr, sizeof(AllocatorT*)
            , 0, cudaMemcpyHostToDevice);
        return 0;
}
```

A.3 Host code in Sanajeh

LISTING A.4: Sanajeh host code (N-Body)

```
from sanajeh import PyAllocator
from device_code.sanajeh_device_code_py import Body
import pygame
import sys

screen_width = 300
screen_height = 300
pygame.init()
screen = pygame.display.set_mode((screen_width, screen_height))
pygame.display.flip()

# Load shared library and initialize device classes on GPU
PyAllocator.initialize()

# Create objects on device
obn = int(sys.argv[1])
itr = int(sys.argv[2])
PyAllocator.parallel_new(Body, obn)

def render(b):
    px = int((b.pos_x + 1) * 150)
    py = int((b.pos_y + 1) * 150)
    pygame.draw.circle(screen, (255, 255, 255), (px, py), 2)

def clear_screen():
    screen.fill((0, 0, 0))

# Compute on device
for x in range(itr):
    PyAllocator.parallel_do(Body, Body.compute_force)
    PyAllocator.parallel_do(Body, Body.body_update)
    PyAllocator.do_all(Body, render)
    pygame.display.flip()
    clear_screen()
```