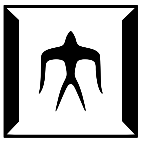


TOKYO INSTITUTE OF TECHNOLOGY

MASTER THESIS

Supporting Multiple Inheritance in a Python DSL for GPGPU

Author:
Fathul Asrar ALFANSURI

Supervisor:
Prof. Hidehiko MASUHARA

Student Number:
19M38053

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

February 28, 2022

Declaration of Authorship

I, Fathul Asrar ALFANSURI, declare that this thesis titled, "Supporting Multiple Inheritance in a Python DSL 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: _____

"It excites me that with a sequence of words called 'codes' given to machines called 'computers', you can create many things!
From a simple `'hello world!'` *to a visually stunning and complex open space world within video games!"*

Fathul Asrar Alfansuri

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

School of Computing
Department of Mathematical and Computing Science

Master of Science

Supporting Multiple Inheritance in a Python DSL for GPGPU

by Fathul Asrar ALFANSURI

Object-support in GPGPU domain specific languages enables highly parallel object-oriented programming on GPUs. This paper improves object-support in Sanajeh, a Python DSL. Sanajeh only supports simple inheritance, in which it is difficult to represent multiple classes in distinct class hierarchies that share common behaviors. We address this problem by creating a version of Sanajeh that supports multiple inheritance. This Sanajeh version transforms multiple inheritance class hierarchy into single inheritance class hierarchy through class flattening and code refactor at the Python level. The transformation process analyzes each class' definitions in the hierarchy, reduces the hierarchy relationship by flattening non-primary parent classes into its child class through linearization, and refactors any types and references they have to the other parts of the code. We evaluate this work by reimplementing agent-based modeling simulation programs used as benchmark by DynaSOAr [8] using multiple-inheritance, and comparing their execution result and code effectiveness. We shown that this extension does not change the expected result of a program rewritten in multiple-inheritance. While the execution time is 2% slower than the original code, the source code become easier to maintain through code reuse and improves its scalability. Through this result, this work contributes to the improvement of NVIDIA-based GPU utilization on high-level Python language.

Acknowledgements

I would like to express my deepest gratitude and acknowledgements to people and entities who have helped me in finishing this dissertation.

The first and foremost is to my supervisor, Prof. Hidehiko Masuhara, who accepted me as a Master student in his laboratory of Programming Research Group (PRG). He helped me numerous times with great patience, not just as a supervisor to aid my studies, but also as a secondary parent that helped me on my daily life in Japan, especially during the very hard times of COVID-19.

I also would not forget the PRG lab members, who allowed me to experience a multi-cultural environment for studies and daily lives. Very special mention to Luthfan Anshar Lubis, a fellow Indonesian student who helped me getting accustomed into my new life in Japan, as well as his great help in this research.

I would also like to thank the PPI Tokodai, a community of Indonesian students in Tokyo Tech. As a first timer of living in another country, they helped me with the daily life guides as well as a familiar environment that resembles home.

Last but not least, is the Japanese Government for the MEXT scholarship, in which this dissertation as well as my whole new experience of living in Japan would not be possible without it.

Contents

Declaration of Authorship iii

Abstract vii

Acknowledgements ix

1 Introduction 1

2 Background 3
2.1 Sanajeh . . . 3
2.2 DynaSOAr . . . 4
2.2.1 Structure of Array . . . 5
2.2.2 Memory blocks . . . 6
2.2.3 Fake pointer . . . 6

3 Problem statement 7
3.1 Single inheritance expressiveness . . . 7
3.2 Custom memory layout . . . 8

4 Proposal 9
4.1 Implementation approach . . . 9
4.2 Multiple inheritance design . . . 10
4.3 Hierarchy conversion algorithm . . . 11
4.3.1 Inspecting class hierarchy . . . 11
4.3.2 Remove mixin-like classes from hierarchy . . . 12
4.3.3 Reinsert mixin-like classes . . . 12
4.3.4 Resolve ambiguity . . . 13

5 Implementation 17
5.1 ClassInspector . . . 17
5.2 Sanajeh modification . . . 18
5.2.1 CallGraphAnalyzer . . . 18
5.2.2 Field expansion . . . 18
5.3 Virtual function . . . 18

6 Case study 19
6.1 Evaluation method . . . 19
6.2 Wa-Tor Simulation . . . 19
6.3 Result . . . 20

7 Related Work 23
7.1 GPGPU with OOP approach . . . 23
7.2 Techniques of multiple inheritance . . . 23
7.3 Language feature conversion . . . 23

8 Conclusion **25**

Bibliography **27**

List of Figures

2.1 Sanajeh process flow. 3
2.2 DynaSOAr Structure of Array layout. 6

3.1 An OOP relationship of bird, horse and pegasus. 8

4.1 Mixin and Sanajeh2 inheritance of Wa-Tor simulation and their usage of linearization. Top-left: Mixin inheritance. Top-right: Mixin inheritance viewed in Python/Sanajeh. Bottom-left: Linearization used to resolve inheritance conflict in mixin inheritance. Bottom-right: Linearization used to modify hierarchy. 11
4.2 An example of Sanajeh2 inheritance hierarchy (top-left), and its conversion (top-right) 12
4.3 SanajehBaseClass inserted to converted hierarchy to be the common ancestor for all classes. 13

6.1 Class hierarchy of Wa-Tor simulations. 20

Listings

2.1 A excerpt of Sanajeh code 4
2.2 Compiled Sanajeh code into CUDA/C++ 4
2.3 Difference between Array-of-Structure and Structure-of-Array layout . 5
2.4 Difference between Array-of-Structure and Structure-of-Array layout . 5
2.5 Example of field access calculation 6
4.1 An example of WingedAnimal being used as parameter type. In converted hierarchy, WingedAnimal class is duplicated and become ambiguous. 14
4.2 The solution to listing 4.1, by renaming the type into SanajehBaseClass and down casting. 14
4.3 An example of super call ambiguity and its solution. 15
6.1 Breed Behavior on Fish class 21
6.2 Breed Behavior on Shark class 21

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 programming for Graphics Processing Unit (GPGPU) allows programmers to write generic programs and run them on Graphical Processing Units (GPU). Originally, GPU was meant to compute shaders and graphics in 3D applications, but nowadays GPU is used in broader applications such as artificial intelligence and crypto mining. Programming languages for GPGPU have been improved in recent years to provide more efficient parallel code execution that runs on GPUs.

One such improvement is the support for object-oriented programming. This enables programmers who are already familiar with object-oriented programming (OOP) to utilize the GPU more effectively. In a similar fashion to Single-Instruction Multiple Data (SIMD) architecture, OOP uses the Single-Method Multiple-Object (SMMO) [7] programming model. In this model, the code runs a single method for every object of the class in parallel.

Another improvement to the GPGPU is the high-level language support. Sanajeh [3] is a Python Domain Specific Language (DSL) for GPGPU that uses both OOP and SMMO paradigms. The DSL supports a single inheritance mechanism for code reuse and type references. It is backed by a lower-level CUDA framework for SMMO applications called DynaSOAr [8]. This framework manages GPU memory allocation and deallocation in Structure-of-Array (SOA) data layout to improve data coalescing; a critical optimization to GPU programming.

However, while Sanajeh's single-inheritance feature provides a good code structure, it is often not enough to represent several kinds of behaviors. For example, single inheritance limits programmers to implement common behaviors that are shared between classes from different hierarchies. This leads to code duplication as well as reduces the scalability of the applications themselves.

In this dissertation, we propose a version of Sanajeh, named Sanajeh2, which supports multiple inheritance. This version of Sanajeh allows programmers to utilize code reuse more flexibly and implements common behaviors such as mixins [1] and traits [2]. We achieve this by designing the OOP with mixin-like multiple inheritance, as well as an algorithm to convert the class hierarchy into the one with single inheritance.

The structure of this paper is as follows. We first address the current situation of Sanajeh (Section 2) and its problem (Section 3). We then explain our approach to this problem by designing the multiple inheritance feature as well as its challenges and how to solve those challenges (Section 4). After that, we discuss the implementation (Section 5). Next comes the evaluation of Sanajeh2 by examining a DynaSOAr benchmark program, Wa-Tor simulation (Section 6). Finally, we discuss related works (Section 7), and then conclude the paper (Section 8).

Chapter 2

Background

This section describes current Sanajeh library [3] as well as DynaSOAr [8], its underlying CUDA/C++ library.

2.1 Sanajeh

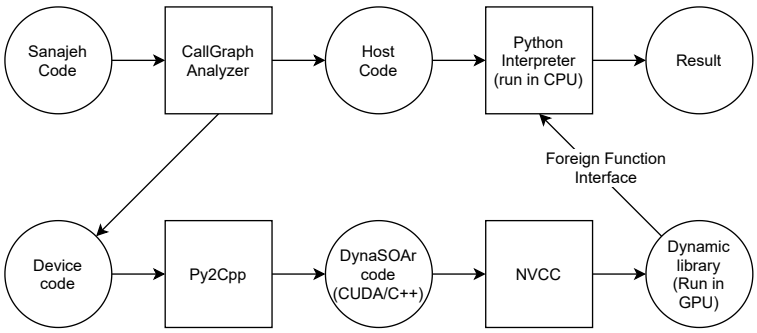


FIGURE 2.1: Sanajeh process flow.

Sanajeh uses ahead-of-time compilation to execute its code. During compilation, the original code is inspected and divided into two parts: *host code* and *device code*, as shown in Figure 2.1. The *host code* is preserved as Python code and will be executed on CPU, while the *device code* will be executed on GPU. The *device code* is converted into CUDA/C++ code then compiled into a dynamic library, as shown in listing 2.1 and 2.2. The *host code* calls the dynamic library through *Foreign Function Interface* (FFI).

To identify which part is the *device code*, Sanajeh uses `CallGraphAnalyzer` to find classes and functions that will be called in GPU. Any classes that will be created or perform SMMO operations on GPU will be flagged as device code. Any classes, functions, and variables that call or get called by a device code will also be flagged as device code as well.

Sanajeh uses a slightly different syntax than the original Python, as shown in listing 2.1. As a Python DSL, Sanajeh inherits Python language's base design. However, as an intermediary to DynaSOAr, Sanajeh follows some of the library's language design instead. In particular, Sanajeh uses static types by utilizing Python's annotation feature. At the same time, Sanajeh uses single inheritance since DynaSOAr is currently designed to only support single inheritance.

LISTING 2.1: A excerpt of Sanajeh code

```
class Cell:
  def __init__(self):
    self.neighbors_: list[Cell] = [None]*4
    self.agent_ref: Agent = None
    self.id_: int = None
    self.agent_type_: int = 0
    self.neighbor_request_: list[bool] = [None]*5

  def Cell(self, cell_id: int):
    random.seed(cell_id)
    self.agent_ref = None
    self.id_ = cell_id
    self.agent_type_ = 0
    self.prepare()
    cells[cell_id] = self

cells: list[Cell] = DeviceAllocator.array(kSizeX*kSizeY)
```

LISTING 2.2: Compiled Sanajeh code into CUDA/C++

```
class Cell : public AllocatorT::Base {
        public:
                declare_field_types(Cell, curandState, DeviceArray<Cell*,
                    4>, Agent*, int, int, DeviceArray<bool, 5>)
                Field<Cell, 0> random_state_;
                Field<Cell, 1> neighbors_;
                Field<Cell, 2> agent_ref;
                Field<Cell, 3> id_;
                Field<Cell, 4> agent_type_;
                Field<Cell, 5> neighbor_request_;

                __device__ Cell(int cell_id);
                __device__ void setup();
                // ....
};

__device__ Cell::Cell(int cell_id) {
        curand_init(kSeed, cell_id, 0, &random_state_);
        this->agent_ref = nullptr;
        this->id_ = cell_id;
        this->agent_type_ = 0;
        this->prepare();
        cells[cell_id] = this;
}

__device__ Cell* cells[kSizeX * kSizeY];
```

2.2 DynaSOAr

DynaSOAr is a CUDA/C++ framework for SMMO applications. This framework provides object support for GPGPU by using classes and single inheritance. The SMMO paradigm is provided through its API, where it allows a parallel invocation of a class method, in which all objects belonging to that class will execute that method. DynaSOAr also supports dynamic object creation and destruction.

2.2.1 Structure of Array

This framework uses Structure-of-Array (SOA) layout for its GPU memory allocation (Figure 2.2). In most implementation of C++, the data members of a class are stored sequentially. On the other hand, SoA data layout groups each field member of a class in an array. For example, a Point class will have an array of `pos_x` and array of `pos_y` instead of an array of contiguous (`pos_x`, `pos_y`), as shown in Listing 2.3).

Memory layout of a child class is placed similar to the AoS layout in that the array of child class members are put after the array of its parent class members, as shown in Listing 2.4.

LISTING 2.3: Difference between Array-of-Structure and Structure-of-Array layout

```
1  // Point class definition
2  class Point {
3          float pos_x;
4          float pos_y;
5  }
6
7  // AoS memory layout
8  struct AoS_Heap_Point {
9          Point p[100];
10 };
11
12 // SoA memory layout
13 struct SoA_Heap_Point {
14         float pos_x[100];
15         float pos_y[100];
16 };
```

LISTING 2.4: Difference between Array-of-Structure and Structure-of-Array layout

```
1  // Point3D class definition
2  class Point3D : public Point {
3          float pos_z;
4  }
5
6  // AoS memory layout
7  struct AoS_Heap_Point3D {
8          Point3D p3d[100];
9  };
10
11 // SoA memory layout
12 struct SoA_Heap_Point3D {
13         // members of Point
14         float pos_x[100];
15         float pos_y[100];
16
17         // members of Point3D
18         float pos_z[100];
19 };
```

2.2.2 Memory blocks

DynaSOAr splits the heap memory into blocks. Each block can only contain one class type, and stores the object members in SoA layout. Blocks are dynamically initialized and reassigned into a class type at runtime. Every block has a fixed size determined at compile time, which is based on the size of the smallest class. As a result, every block can only hold a set amount object depending on class size: Blocks of a larger class will contain fewer objects of that class.

To further illustrate, consider the heap representation in figure 2.2. Each blocks has the same size. Blocks of class Point have 64 slots of Point objects, while Blocks of class Point3D will only have 42 slots. These maximum number of slots per block are known at compile time, but different for each class type.

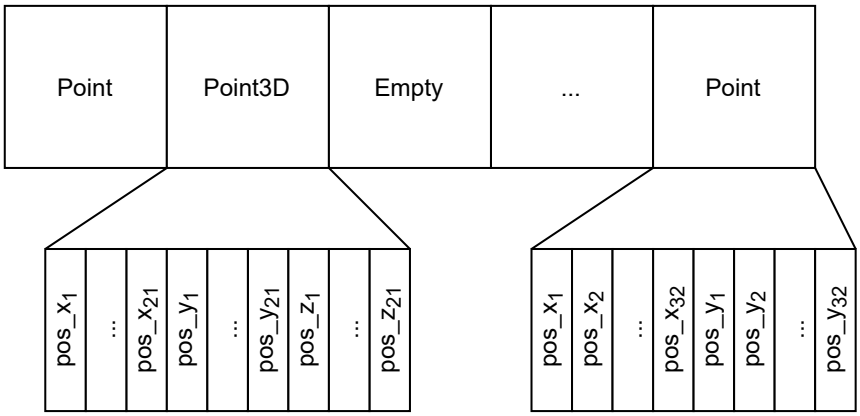


FIGURE 2.2: DynaSOAr Structure of Array layout.

2.2.3 Fake pointer

In order to access an object's field, a custom operation is required. In DynaSOAr, a pointer to an object is a fake pointer. This custom pointer consists of:

- `b`: Memory address of the block this object is stored in.
- `i`: The index position of this object in this block. (i-th position in the array)
- `n`: The block size, which is dependant of the class type of the block.

The exact location of an object's field is calculated as represented in Listing 2.5. The calculation of member `pos_y` is the same between Point and Point3D class, which means that DynaSOAr supports single inheritance.

LISTING 2.5: Example of field access calculation

```
1 struct Point3D_pointer {
2         Block* b;
3         int i;
4         int n;
5 }
6
7 float* pos_y = b +                 // address of the block
8                 sizeof(int) * n + // array of pos_x
9                 sizeof(int) * i   // array of pos_y
```

Chapter 3

Problem statement

In GPGPU, implementing real-world application is difficult due to single-inheritance. For example, in a simulation program such as agent-based simulations, parallel OOP is desirable. This is due to the usually large number of objects and each object executes the same operations on each timestep. Implementing these applications in Sanajeh would beneficial. However, often there are multiple object types that share several common behaviors. It is difficult to express this model only through single inheritance.

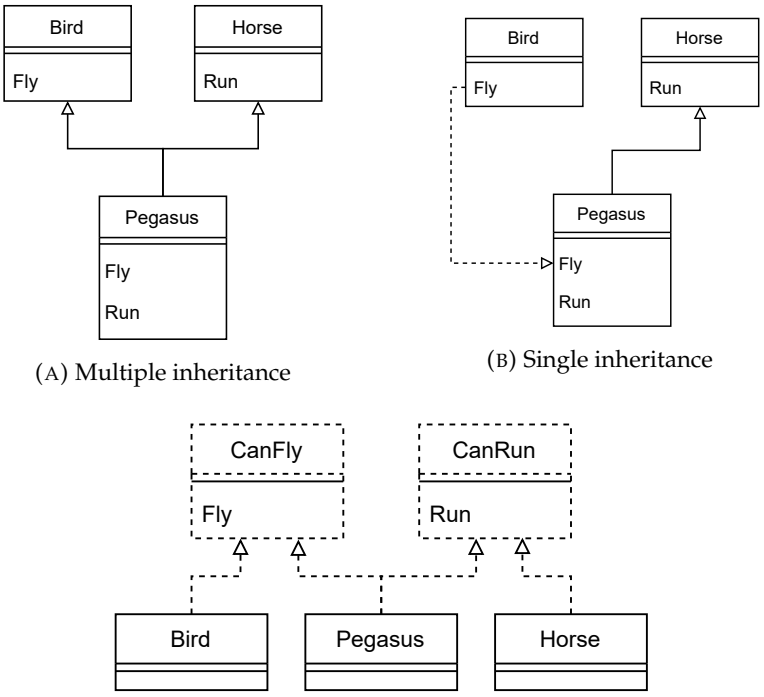
3.1 Single inheritance expressiveness

A concrete example of the aforementioned difficulties can be described as follows. Consider a simple class hierarchy of horse, bird and pegasus (a mythical horse with wings and could fly) as shown in Figure 3.1a. A bird may have method *fly* as its movement behavior, while a horse uses method *run* instead. Logically, we can define an "IS-A" relationship between them; i.e. pegasus is a horse and a bird. Therefore, a pegasus may be able to both *fly* and *run*. In the simulation, there are many horses, birds, and pegasus on a field, and on each tick of time, all horse instances (and its children instances, the pegasi) executes *run* method and all birds and pegasi doing *fly* method.

To express these kinds of hierarchy, single inheritance is not enough. While Python and CUDA actually supports multiple inheritance, Sanajeh is restricted into single inheritance design. There are two approaches that can be used as a workaround for this representation by using only single inheritance: simply copying methods/-fields and using mixin structure. These approaches have their own disadvantages that can be reduced by using multiple inheritance.

On the first approach, the hierarchy can be defined by letting pegasus class to inherit from the horse class, and then copy bird class' definition directly into the pegasus class (Figure 3.1b). However, whenever a user updates the implementation of bird class she must also updates the copied definition in the pegasus class.

The other approach for this hierarchy is to redefine the logical structure by deriving the classes of the three animals from mixin instead, i.e. *CanFly* and *CanRun* mixin (Figure 3.1c). Unfortunately, this approach does not translate well in Python since the mixin will still be considered as a parent class instead. The graph will be viewed as multiple inheritance hierarchy and current Sanajeh will not accept this structure.



(A) Multiple inheritance

(B) Single inheritance

(C) Mixin-based inheritance

FIGURE 3.1: An OOP relationship of bird, horse and pegasus.

3.2 Custom memory layout

As described in Background section, the underlying library, DynaSOAr, utilizes a custom memory layout to optimize GPU computation. However, it makes DynaSOAr difficult to implement multiple inheritance, even if the CUDA/C++ itself supports multiple inheritance. Current Sanajeh adheres to this restriction by limiting its OOP mechanism to single inheritance. In particular, the *device code* part of Sanajeh must be in single inheritance hierarchy.

Chapter 4

Proposal

This chapter describes our proposal to address the problem: Sanajeh2, a version of Sanajeh that supports multiple inheritance and virtual function. First, we discuss different ways of implementing multiple inheritance for Sanajeh2, and the reasoning for doing the hierarchy conversion. Then, we will discuss the basis of the hierarchy conversion, as well as the general algorithm of the conversion. Meanwhile, Virtual Function support does not require in-depth discussion, and will be described in the next chapter.

4.1 Implementation approach

We consider two approaches of implementing multiple inheritance for Sanajeh2:

1. Implement multiple inheritance on DynaSOAr (C++ level).
2. Implement multiple inheritance on Sanajeh (Python level).

The first approach relates to the fact that Sanajeh only supports single inheritance due to DynaSOAr limitation. By implementing multiple inheritance directly in DynaSOAr, Sanajeh no longer be limited to single inheritance. The second approach is to implement multiple inheritance at Python-level, while still using DynaSOAr's single inheritance structure. This would require a hierarchy conversion to allow Sanajeh2 codes to run on DynaSOAr.

We decided to implement multiple inheritance at Python level. Compared to the first approach, implementation at DynaSOAr level is more difficult. DynaSOAr requires different, custom implementation for multiple inheritance than the standard C++.

Designing a new multiple inheritance implementation such as data layout and calculating field access in DynaSOAr would require many considerations, as C++ is a powerful low-level language with many features. One such considerations is virtual inheritance, in that in diamond pattern a parent class can be virtual (shared parent data) or non-virtual (duplicated parent data). The paper [11] noted that these variations are part of the complexity of C++ multiple inheritance. Among the examples provided in the paper to show the problems of C++ multiple inheritance, more than half of them discussed the non-virtual inheritance and casting between parent and child class.

Calculating field access in DynaSOAr with multiple inheritance is not simple. Standard C++ multiple inheritance uses a simple casting operation to access parents field members. In non-virtual inheritance case, casting between child and parent type is done by using an offset. Therefore, to access a parent field, the child pointer is moved by an offset to parent data starting location, and then offsetting the pointer again to the member data. Meanwhile, DynaSOAr object pointer is a fake pointer

that basically points to its block position. Casting from child type to parent type and vice versa would need additional computation and information, as opposed to simple offset. This is further complicated when accomodating the virtual inheritance case.

On the other hand, Python's multiple inheritance is simpler. Inheritance is always virtual: parent fields are never duplicated in the child class. Member functions are also virtual, which means that the complexity of casting `this` pointer between parent and child classes are not present.

This second approach requires a mechanism to convert multiple inheritance hierarchy into single inheritance hierarchy. Sanajeh2 also uses DynaSOAr as its lower level CUDA framework, with its limitation of single inheritance unchanged. Since Sanajeh uses ahead-of-time compilation, we can apply a code transformation to user-created code before its translation into CUDA code.

4.2 Multiple inheritance design

We use a design similar to Mixin for Sanajeh2 multiple inheritance. Mixin design allows a class to inherit from two types of parent class: a "true parent" as in single inheritance, and several "mixin parent" to grant additional functionalities. To resolve inheritance conflict, mixin uses linearization. Both mechanism (two types of parent and linearization) is used in Sanajeh2 multiple inheritance design.

The difference between mixin and Sanajeh2 design lies in the hierarchy and how linearization is used. Mixin design is actually a single inheritance hierarchy, while Sanajeh2 is a multiple inheritance hierarchy. Mixin linearization is used to "view" the class hierarchy as a single inheritance to resolve inheritance conflict. On the other hand, Sanajeh2 uses linearization to actually *modify* the hierarchy itself.

To illustrate, consider the example shown in figure 4.1. In mixin design, both Fish and Shark class has one "true parent" named Agent, and using a "mixin parent" named BreedingBehavior (top-left hierarchy). The inheritance conflict resolution for both Fish and Shark are resolved by linearization of mixin parents (bottom-left hierarchy). Meanwhile, there is no distinction between mixin and non-mixin classes in Python: it is a multiple inheritance hierarchy (top-right). Instead resolving inheritance conflict, linearization is used to modify the hierarchy so that hierarchy becomes single inheritance hierarchy (bottom-right).

In detail, we categorize the classes in the hierarchy according to its inheritance usage:

1. **Prime Parent**. Prime Parent is a parent class that retain its parent-child relationship in the single-inheritance version.

2. **Mixin-like Parent**. Mixin-like Parent is also treated as parent class, but due to conversion into single inheritance, the parent-child relation is severed and preserved through other means.

To simplify the conversion algorithm, we define that the Prime Parent of a class is the first parent in the class' definition. Users can reorder parent list of a class to determine a Prime Parent that will retain its parent-child relationship with minimum modification. All other parents will be considered as Mixin-like Parents and have their relationship modified in the resulting hierarchy.

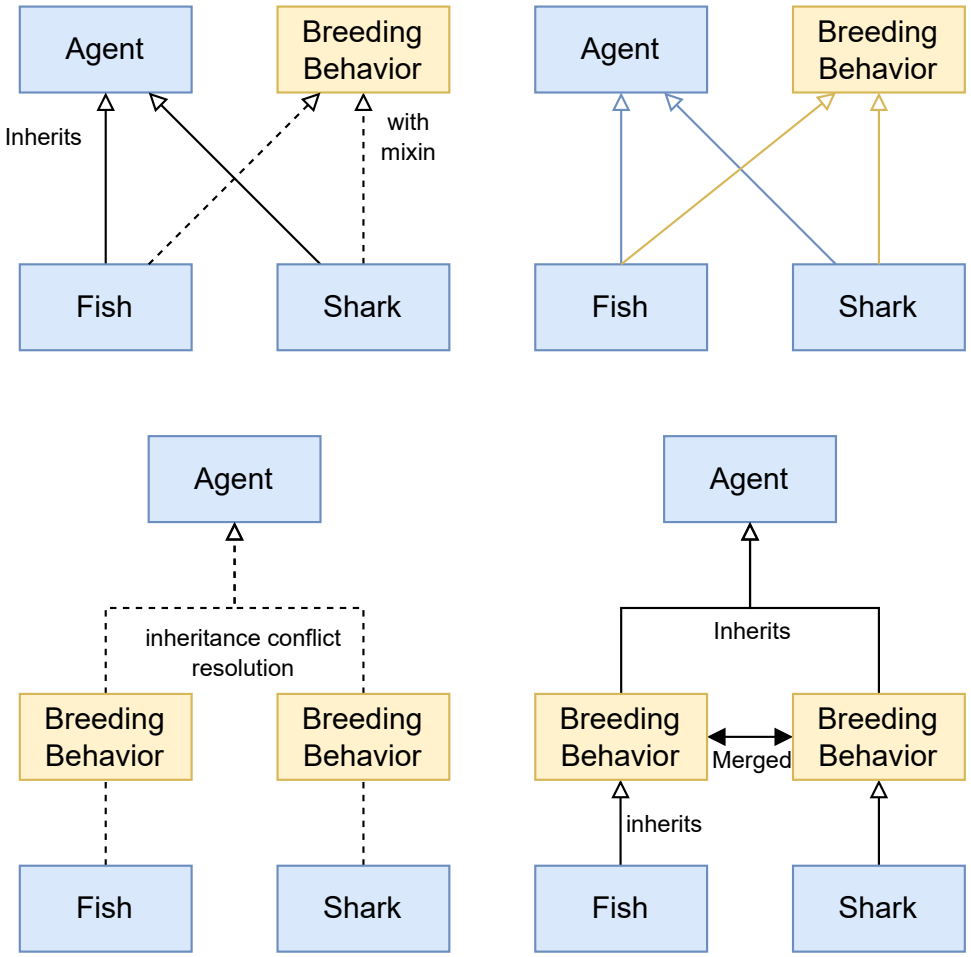


FIGURE 4.1: Mixin and Sanajeh2 inheritance of Wa-Tor simulation and their usage of linearization. Top-left: Mixin inheritance. Top-right: Mixin inheritance viewed in Python/Sanajeh. Bottom-left: Linearization used to resolve inheritance conflict in mixin inheritance. Bottom-right: Linearization used to modify hierarchy.

4.3 Hierarchy conversion algorithm

Our algorithm for hierarchy conversion is as follows:

1. Inspect class hierarchy, labelling classes into Prime Parent and Mixin-like Parent classes.
2. Remove Mixin-like classes from original hierarchy, which leaves the remaining classes to form single inheritance hierarchy.
3. Flatten the Mixin-like classes into its inheriting classes by linearization and re-insertion in between the original inheritance, and duplicate if necessary.
4. Resolve ambiguities from duplication and reinsertion of Mixin-like classes, such as type rename.

4.3.1 Inspecting class hierarchy

The first step is to generate the class hierarchy and label each classes. The general flow of the algorithm is to traverse the hierarchy for classes that has multiple parents.

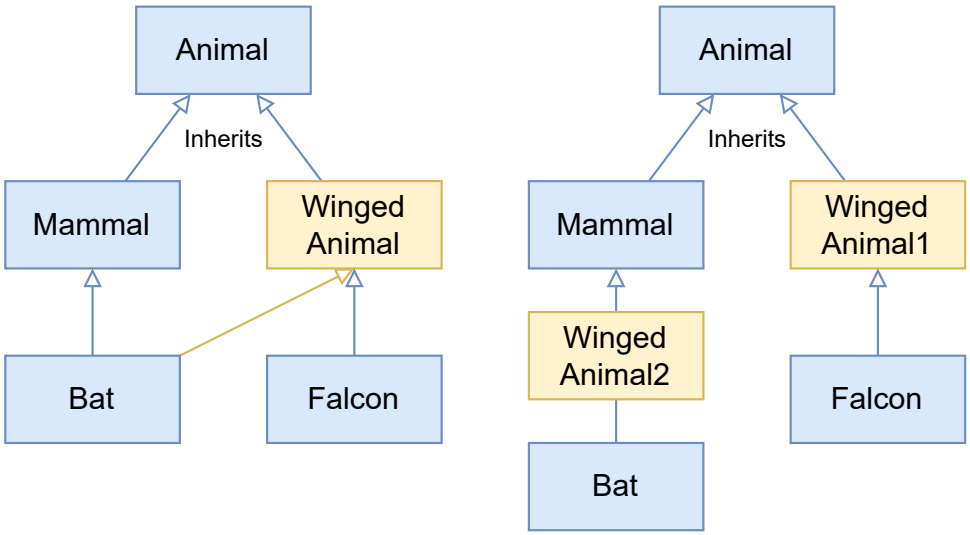


FIGURE 4.2: An example of Sanajeh2 inheritance hierarchy (top-left), and its conversion (top-right)

Parent classes that are not the first in the parent list are marked as Mixin-like classes. Other classes are defaulted to Prime Parent classes.

Consider the hierarchy example in figure 4.2. It shows that Bat class has 2 parents: Mammal and WingedAnimal. It is up to the user to order the parent list in the class definition, and in this example the Mammal is set as the Prime Parent, and WingedAnimal becomes Mixin-like Parent. It is possible for a class to be a Prime Parent of one inheriting class, and be a Mixin-like Parent of another inheriting class, like the WingedAnimal class. In this case, the Mixin-like Parent label takes precedence.

The class hierarchy will be fully labeled as either Prime Parent or Mixin-like Parent classes. The class relationship between Prime Parent classes will result in a single inheritance hierarchy, which will become the main hierarchy. Mixin-like parents will be linearized into its inheriting classes based on the next step.

4.3.2 Remove mixin-like classes from hierarchy

To form single inheritance hierarchy, all inheritance relationships between a class and its mixin-like parent are removed. This is similar to Mixin design that without mixin parents, the class hierarchy is a single inheritance hierarchy.

4.3.3 Reinsert mixin-like classes

Mixin-like classes are reinserted back to the hierarchy. To preserve single inheritance property, these classes are reinserted by placing them between the existing relationship.

The general flow of this step is to inspect each class in the hierarchy, starting from the top. For each inspected class do the following:

1. List all Mixin-like classes that this class inherits, respecting the order from the parent list.

2. Linearize these classes by ordering them from left to right: each classes inherits the class to the right. If the class has parent(s), Linearize them first, and set the topmost ancestor to inherit the class to the right.

3. The Mixin-like classes are now a stack of single inheritance hierarchy. Remove classes that are already inherited by class from the stack.

4. Put the stack in between this class and its Prime Parent class.

Every inspected class linearize their Mixin-like classes separately. This may result in the duplication of some classes. At the end of inspections, if the duplicated classes inherits the same class, they can be merged. If merging is not possible, the duplicated classes are renamed by appending a number.

4.3.4 Resolve ambiguity

The linearization from the previous step may cause ambiguity and inconsistency.

The ambiguities and inconsistencies that may arise are as follows:

1. A duplicated class is used as a variable type.

2. The `super()` call is used, but the direct parent of the class is changed from the hierarchy.

To illustrate the first example, consider the converted hierarchy in figure 4.2. The WingedAnimal class is duplicated and cannot be merged. If it is used as variable or parameter type, the type will be ambiguous between WingedAnimal1 or WingedAnimal2.

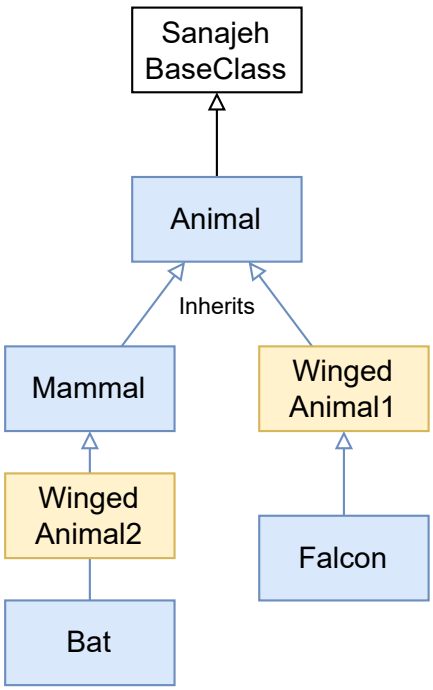


FIGURE 4.3: SanajehBaseClass inserted to converted hierarchy to be the common ancestor for all classes.

A solution to this ambiguity is to rename the type to be the common ancestor class of the duplicated classes. We insert a new, empty base class named SanajehBaseClass to the converted hierarchy (figure 4.3. All classes that does not have a parent will inherit from this class. This way, SanajehBaseClass is the common ancestor for all classes.

LISTING 4.1: An example of WingedAnimal being used as parameter type. In converted hierarchy, WingedAnimal class is duplicated and become ambiguous.

```
1  // Original method
2  __device__ void Bat::Mate( WingedAnimal* other );
3
4  // A function that calls the method
5  __device__ void Bat::Propose() {
6    WingedAnimal *other;
7    this->Mate( other );
8  }
```

LISTING 4.2: The solution to listing 4.1, by renaming the type into SanajehBaseClass and down casting.

```
1  // Original method, now using template
2  template<typename T>
3  __device__ void Bat::Mate( T* req );
4
5  // wrapper to the method, generated by ClassInspector
6  __device__ void Bat::Mate_Cast( SanajehBaseClass* other ) {
7    // Attempt to downcast to WingedAnimal1
8    WingedAnimal1 other_WingedAnimal1 = other->cast<WingedAnimal1>();
9    if (other_WingedAnimal1!=nullptr) return this->Mate<WingedAnimal1>(
         other_WingedAnimal1);
10
11   // Attempt to downcast to WingedAnimal2
12   WingedAnimal2 other_WingedAnimal2 = other->cast<WingedAnimal2>();
13   if (other_WingedAnimal2!=nullptr) return this->Mate<WingedAnimal2>(
         other_WingedAnimal2);
14 }
15
16 // A function that calls the method
17 __device__ void Bat::Propose() {
18   WingedAnimal *other;
19   this->Mate_Cast( other );
20 }
```

Since the code will be translated into CUDA/C++, down casting from SanajehBaseClass to its actual runtime class is required to access the class' field members and functions. We resolve this by iteratively casting them into its possible classes, as shown in listing 4.1 and 4.2.

The second ambiguity can also be seen in the hierarchy of figure 4.2. If the original WingedAnimal contains a `super()` call in its methods, it will be ambiguous. Python's Method Resolution Order (MRO) for WingedAnimal1 will refer to Animal class, and Mammal class for WingedAnimal2. This ambiguity can be easily solved by changing the `super` parameter into its appropriate classes, as shown in listing 4.3.

LISTING 4.3: An example of super call ambiguity and its solution.

```
1  ###################################
2  # Definition in original hierarchy
3  class WingedAnimal(Animal):
4    def Run(self):
5      super().Run() # will refer to Animal.Run()
6
7  ###################################
8  # Definition in converted hierarchy
9  class WingedAnimal1(Animal):
10   def Run(self):
11     super(WingedAnimal1, self).Run() # will refer to Animal.Run()
12
13 class WingedAnimal2(Mammal):
14   def Run(self):
15     super(Mammal, self).Run() # will refer to Animal.Run()
```

Chapter 5

Implementation

We did several things to implement Sanajeh2. First, we implemented the hierarchy conversion algorithm in the form of ClassInspector. In addition, we also implement virtual function as part of Sanajeh2. Furthermore, we decided to disable Field Expansion optimization of the original sanajeh.

There are parts of Sanajeh2 specifications described in previous section that hasn't been implemented yet. Specifically, the last step of hierarchy conversion has not been implemented. However, current stage of implementation is sufficient to test our algorithm for the example program that we use, the Wa-Tor simulation.

5.1 ClassInspector

The `ClassInspector` class generates a Directed Acyclic Graph (DAG) of the class hierarchy, as well as defining the class node information. The graph is generated using Python's `ast` library and using visitor pattern. `ClassInspector` generates the AST of the source code then visits the nodes to gather information about class relationships. The class node stores the following informations:

- Class name.
- AST node of the class.
- list of parents class nodes.
- list of children class nodes.
- Number of duplications.

`ClassInspector` does the conversion algorithm in two steps. The first step is to use `NodeVisitor` to traverse the AST of input code to gather information of class hierarchy. Second step is to use `NodeTransformer` to transform the AST so that it become single inheritance hierarchy.

The general flow of the first step is as follows:

1. Generate AST of the input source code.
2. Visit AST to generate DAG class nodes and obtain parent names. The AST does not contain information about class' children.
3. cross-reference each class nodes to obtain parent/children node references.
4. Visit AST to mark classes as Prime Parent or Mixin-like classes.

Afterwards, the second step of hierarchy conversion begins, which does the following:

1. Create a new class named SanajehBaseClass, and add it to the Module body.
2. Traverse the AST. For each class definition, do the following:
   (a) If it has no parent, set its parent to SanajehBaseClass.
   (b) Linearize Mixin-like parents as described in 4.3.3.
   (c) If a duplication happens, store the info on the duplicated class, including the numbering.
3. Traverse the AST. For each class definition, do the following:
   (a) Check its children for duplication caused by previous step.
   (b) If found, merge the duplicated class as one class, joining their children.
4. Modify the Module node to generate new definitions for duplicated classes.

Aside from implementing hierarchy conversion, we also modify the initialization step. We require users to explicitly call initialization/constructor methods for each of the parent classes. This is to simplify the translation step from Python to CUDA/C++.

5.2 Sanajeh modification

We decide to modify some of the Sanajeh's logic and design in order to support multiple inheritance.

5.2.1 CallGraphAnalyzer

Initially, `CallGraphAnalyzer` marks portion of the code as device code if it is being referenced by another device code. We expand this further by marking a class as device code if it is a parent or a child of a device code class.

5.2.2 Field expansion

Sanajeh implements field expansion as a performance optimization. However, as we implement our extension, class references are used to great extent that we decided to sacrifice this optimization for a more robust code.

5.3 Virtual function

Virtual Function is implemented as a member function of Sanajeh2 library. DynaSOAr does not have a built-in API for virtual function, but provided an example on how to achieve the same result. It iteratively casts the `self` variable into each subclasses on the original hierarchy, starting from the lowest subclass (Subclasses that does not have any children). If the cast is successful, it runs that subclass' method.

Chapter 6

Case study

We use DynaSOAr benchmark programs as a basis for our evaluation. The case study consists of the following steps for the benchmark program:

1. Discuss its single inheritance disadvantages that can be reduced or eliminated by an implementation using multiple inheritance.
2. Rewrite the program into the multiple inheritance alternative.
3. Compare the result of the rewritten program against the original.
4. Compare the source code of both programs.

6.1 Evaluation method

We first discuss the implementation of benchmark program. These programs was written using single inheritance paradigm. However, the implementation code are not necessarily the most optimal way to write these programs. It is due to the single-inheritance limitation of DynaSOAr that this is the only representation currently available. Thus, we examine the parts where it may cause problems and can be solved by using a multiple inheritance approach.

After we found the parts to be improved, we design a multiple inheritance alternative that will address the problems, and rewrite the program for Sanajeh2. This multiple inheritance version is converted into a single inheritance version to be able to run on original Sanajeh.

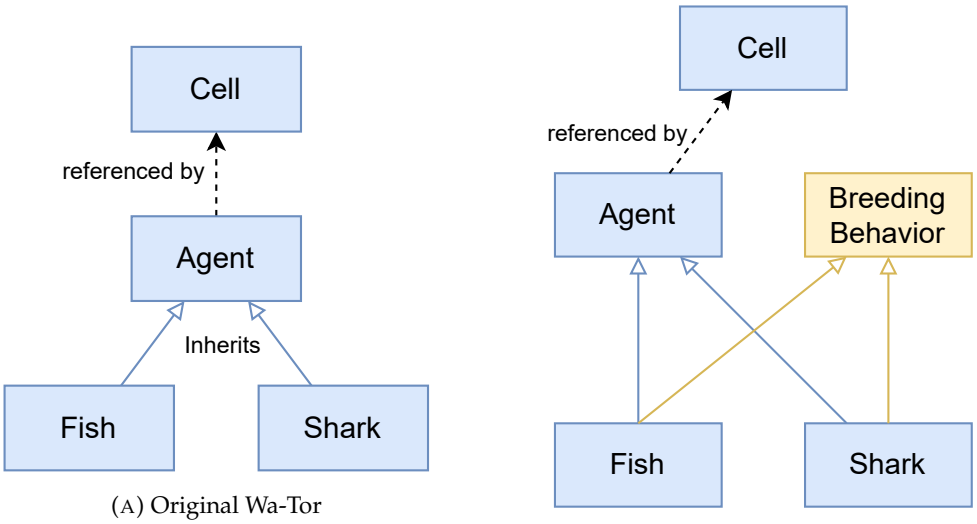
This work is evaluated in two aspects: its correctness and its effectiveness. Since it is difficult to properly define the correctness of our algorithm, the first aspect is evaluated by comparing both its output and the original single inheritance version. Sanajeh2's conversion is said to be correct if the algorithm will not change the expected result of the program.

The effectiveness of our work is evaluated by comparing the original code of both the single inheritance and multiple inheritance. Specifically, we will discuss about the code size, performance, maintainability and scalability between the two of them.

As of now, we have successfully implemented one of the benchmark programs that has a multiple-inheritance alternative which we thought would be more effective. The DynaSOAr program is Wa-Tor, an agent-based simulation program.

6.2 Wa-Tor Simulation

Wa-Tor is an agent-based simulation where two types of agent, fish and shark, coexists in a planet shaped like torus. It simulates a predator and prey relationship and



(A) Original Wa-Tor

(B) Wa-Tor with multiple inheritance

FIGURE 6.1: Class hierarchy of Wa-Tor simulations.

how it affects population. Fish and sharks are able to move and reproduce, while shark will eat nearby fish. It is simulated in a 2D grid: a point may only be filled with either a Fish object, a Shark object, or an empty spot. Both Fish and Shark objects move randomly by checking its empty neighbors. If there are no empty spots, both classes remains on their spot. A Shark will try to occupy a neighboring Fish, which is defined as the shark eating the fish, removing the Fish object. A class diagram of this simulation is represented in Figure 6.1a.

The current implementation shows that there is a code reduplication in Fish and Shark class, as shown in listing 6.1 and 6.2. The code that is duplicated represents the breed logic for the two agents. While one can put this logic directly on Agent class, this may cause some problems in scalability. For example, consider the case that we will add more type of agents which does not necessarily has this breed behavior. If the number of agent types that does not use it outweighs the agent types that uses it, the egg_timer_ field will be a waste of space and the code that implements the breed will be unreachable.

Thus, our implementation aims to reduce this duplication by separating the breed logic from the agents, and put them as another class to be inherited with. This way, whenever we add another agent, we can opt in whether to use the breeding behavior by simply inheriting from it and add few lines of method invocation.

6.3 Result

We run the simulation on a machine equipped with NVIDIA TITAN Xp GPU with 12GB of memory. The simulation parameters are 100x100 cell size, and 100 steps for each run. Rendering mode is used to compare the visual output, and no-render mode is used to compare the execution time.

Both implementation yields similar result. Due to the non-deterministic nature, we don't know if changing the code structure may affect the random generation, even if we use the same seed. Thus, we observe the result by checking the rendered simulation of both programs. The two programs renders similar behavior of agent movements, breed behavior and predatory actions. Both simulation also shown similar observation of population dynamics between Fish and Shark, as is expected on

LISTING 6.1: Breed Behavior on Fish class

```
__device__ void Fish::prepare() {
  egg_timer_++;
  // Fallback: Stay on current cell.
  new_position_ = position_;

  // ...
}

__device__ void Fish::update() {
  Cell* old_position = position_;

  if (old_position != new_position_) {
    old_position->leave();
    new_position_->enter(this);

    if (kOptionFishSpawn && egg_timer_ > kSpawnThreshold) {
      auto* new_fish = new(device_allocator) Fish(curand(&random_state_));
      assert(new_fish != nullptr);
      old_position->enter(new_fish);
      egg_timer_ = (uint32_t) 0;
    }
  }
}
```

LISTING 6.2: Breed Behavior on Shark class

```
__device__ void Shark::prepare() {
  egg_timer_++;
  energy_--;

  // ...
}

__device__ void Shark::update() {
  if (kOptionSharkDie && energy_ == 0) {
    position_->kill();
  } else {
    Cell* old_position = position_;

    if (old_position != new_position_) {

      // ...

      if (kOptionSharkSpawn && egg_timer_ > kSpawnThreshold) {
        auto* new_shark =
            new(device_allocator) Shark(curand(&random_state_));
        assert(new_shark != nullptr);
        old_position->enter(new_shark);
        egg_timer_ = 0;
      }
    }
  }
}
```

wa-tor simulation. Therefore, it is safe to say that our algorithm is correct for this benchmark.

There is a tradeoff between performance and code efficiency, as shown in table

	Single Inheritance version	Multiple Inheritance version
Execution time	17.644 seconds	18.018 seconds
Source code size	309 lines	330 lines

TABLE 6.1: Execution result of Wa-Tor simulation.

6.1. On average, our version runs 2% slower than the original. This may be due to the code divergence from using virtual function.

On the other hand, the separation of BreedBehavior logic improves scalability and maintainability. In terms of code size, our version has 349 lines of code (LoC) against 330 LoC of the original. However, this is due to the overhead of BreedBehavior class declaration, and there are only 2 classes that uses them. Further agent types that uses this logic will only need to put this class as its parent without additional LoC, as opposed to repeatedly copying the logic into each class. Furthermore, any changes to breeding logic will be concentrated on the definition of this class rather than individually change every agent types that uses this logic. In addition, should there be another behavior added to the agents (e.g. VirusInfectionBehavior), agent types can choose whether to use this behavior simply by inheriting this class.

Chapter 7

Related Work

In this chapter, we discuss related work in the following three categories: (7.1) OOP support for GPGPU, (7.2) implementation techniques for multiple inheritance, and (7.3) Language feature conversion.

7.1 GPGPU with OOP approach

There is some popular language with OOP features that supports GPGPU utilization, such as Java, Javascript, C-family, and Python itself.

Jcuda [12] for Java, GPU.js [6] for Javascript, Ikra-ruby [5] for Ruby, and PyCUDA [4] for Python provides a simple interface to access GPU functions. However, rather than directly supporting OOP for GPGPU from their language feature, they simply provide an API abstraction for codes to be run on GPU in the fashion of functional programming.

CUDA is a dialect of C++, a language that supports OOP. As a result, programming in CUDA/C++ allows limited OOP-style programming. Ikra-Cpp [9] is a CUDA/C++ DSL for OOP approach with a Structure-of-Array layout to optimize memory access. DynaSOAr [8] is a framework on top of Ikra-Cpp which enables dynamic memory allocation/deallocation, a common feature in general OOP.

7.2 Techniques of multiple inheritance

Multiple inheritance is an OOP feature that is long considered both powerful and complicated, as presented by its diamond problem. Many work tries to workaround the problematic part while maintaining its advantages. Java uses Interfaces as a way to introduce sub-type information while avoiding the diamond problem. Mixin [1] introduces another type of parent class that can be linearized to resolve inheritance conflict. Traits [2] provides a code-reusability mechanism to mimic multiple inheritance.

7.3 Language feature conversion

Modular Class-based Reuse Mechanisms [10] defines a modular meta-level runtime architecture that converts several code reuse mechanisms into a single inheritance environment. The work involves converting a language with several code reuse features (including multiple inheritance) to be able to run in a Virtual Machine (VM) which is optimized for single inheritance. This work is tailored to convert languages with multiple inheritance feature into a Virtual Machine that runs on single inheritance. Compared to that, Sanajeh2 converts its multiple inheritance feature into Sanajeh with single inheritance.

Chapter 8

Conclusion

We present a design of OOP for GPGPU with Mixin-like multiple inheritance and the algorithm to convert the class hierarchies into single inheritance hierarchies, as a version of the Sanajeh library named Sanajeh2. This version of Sanajeh will increase code maintainability and scalability on writing highly parallel object-oriented programming.

We use a DynaSOAr benchmark program, Wa-Tor, to evaluate our work. We had shown that our algorithm does not affect the expected result of the original code. At the same time, we also showed that it provides code maintainability and scalability while discussing its tradeoff with performance.

The hierarchy conversion algorithm is different than Python's built-in C3 Linearization. Thus, in a more complex hierarchy, the inheritance order might be different than what the user expected as Python DSL. In the future, this can be resolved by incorporating Python's built-in linearization into our conversion method.

While it is possible to convert multiple inheritance hierarchy into a single inheritance hierarchy, it does sacrifice some of the safety features such as type-checking. This can lead to a discussion about improved multiple-to-single inheritance conversion algorithms. Furthermore, this paper only shows that our algorithm can convert several multiple inheritance programs into single inheritance. A further examination of the algorithm's soundness and validity to convert any multiple inheritance code into a single inheritance hierarchy is still open to be discussed in the future. In fact, this leads to another discussion about whether any multiple inheritance code can be converted into a single inheritance code without changing the expected result.

Bibliography

[1] Gilad Bracha and William Cook. "Mixin-based inheritance". In: *ACM Sigplan Notices* 25.10 (1990), pp. 303–311.

[2] Stéphane Ducasse et al. "Traits: A mechanism for fine-grained reuse". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28.2 (2006), pp. 331–388.

[3] Chenxin Jizhe. "Nested object support in an object-oriented domain-specific language for GPGPU". MA thesis. Tokyo Institute of Technology, 2021.

[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] Fazli Sapuan, Matthew Saw, and Eugene Cheah. "General-purpose computation on GPUs in the browser using GPU.js". In: *Computing in Science & Engineering* 20.1 (2018), pp. 33–42.

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

[8] 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, 17:1—17:37.

[9] Matthias Springer and Hidehiko Masuhara. "Ikra-Cpp: A C++/CUDA DSL for object-oriented programming with structure-of-arrays layout". In: *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*. 2018, pp. 1–9.

[10] Pablo Tesone et al. "Implementing modular class-based reuse mechanisms on top of a single inheritance VM". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 1030–1037.

[11] Daniel Wasserrab et al. "An operational semantics and type safety prooffor multiple inheritance in C++". In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 2006, pp. 345–362.

[12] Yonghong Yan, Max Grossman, and Vivek Sarkar. "JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA". In: *European Conference on Parallel Processing*. Springer. 2009, pp. 887–899.