Supporting Multiple Inheritance in an Object-Oriented DSL for GPGPU by Class Hierarchy Transformation

Fathul Asrar Alfansuri, Hidehiko Masuhara,
Luthfan Lubis, Youyou Cong

Object-support in GPGPU domain specific languages (DSL) enables highly parallel object-oriented programming on GPUs. This paper improves object-support in Sanajeh, a Python DSL for GPGPU, by adding a multiple inheritance feature. Existing Sanajeh is restricted to single inheritance, which has limited flexibility. This restriction is due to its backend library which is not trivial to be extended. The limitation makes it difficult for Sanajeh to represent multiple classes that share common behaviors but belong to different class hierarchies. Our approach transforms a multiple inheritance class hierarchy into a single inheritance class hierarchy through mixin linearization, class duplication and wrapper functions at the Python level. We evaluate this work by rewriting agent-based modeling simulation program. We then compare the original and rewritten version with respect to their execution result and code effectiveness. The execution time is 2% slower than the original code.

1 Introduction

Ease-of-use for General Programming for Graphical Processing Unit (GPGPU) is an active research topic in parallel programming. One approach to make GPGPU easier to use is to implement Object-Oriented Programming (OOP).

One feature of OOP is multiple inheritance. This feature offers flexibility an code reuse. Due to its complexity however, it is interpreted and implemented differently for each of OOP-based languages.

Sanajeh [3] is a Python DSL GPGPU. While Python itself supports multiple inheritance, Sanajeh is restricted to only support single inheritance. This restriction is due to the DSL using a custom framework for CUDA/C++ called DynaSOAr [8], which only supports single inheritance.

This study describes an implementation of multiple inheritance for Sanajeh. The multiple inheritance in this study is a mixin-like inheritance [2], which differs from Python's own multiple inheritance. The implementation is done through code transformation that converts a multiple inheritance class hierarchy to a single inheritance class hierarchy. The transformation algorithm is similar to hierarchy linearization in mixin.

The rest of the paper is organized as follows. Section 2 explains about Sanajeh and its backend framework, DynaSOAr. We then discuss the problem in implementing multiple inheritance on Sanajeh in Section 3. In Section 4, we describe the code transformation and its implementation details. Section 5 shows our testing of the transfor-

Fathul Asrar Alfansuri, Hidehiko Masuhara, Luthfan Lubis, Youyou Cong, School of Computing, Department of Mathematical and Computing Science, Tokyo Institute of Technology.

mation, its results, and the discussion. Section 6 discuss works related to this study. Finally we conclude the paper in section 7.

2 Background

2.1 Sanajeh and DynaSOAr

Sanajeh is a Python DSL for GPGPU. This DSL utilizes DynaSOAr, a CUDA/C++ framework for GPGPU, as its backend to execute GPU code. As a result, Sanajeh uses ahead-of-time compilation to run its programs instead of using interpreter. During compilation, the DSL splits user code into two: host code and device code. Host code remains as Python code, while device code is converted into DynaSOAr code and compiled into shared library. Upon runtime, Sanajeh uses Python FFI to interact with compiled shared library in order to execute the GPU code.

DynaSOAr is a CUDA/C++ framework for GPGPU. This framework uses custom memory layout in order to optimize memory coalescing. The framework also supports dynamic object allocation and deallocation, in which allocated objects are given a fake pointer to access its members. DynaSOAr only supports single inheritance, which in turn limits Sanajeh to also supports single inheritance only.

2.2 DynaSOAr memory layout

DynaSOAr uses a custom Structure-of-Array (SoA) memory layout in order to optimize memory coalescing. This framework creates a memory heap which is converted into blocks with fixed size (figure 1a). Each blocks may only be allocated for one class type, which may vary in size. Therefore, the block varies in the maximum number of objects it can contain, depending on the size of the class it currently designated to (figure 1b and 1c). This number is known at compile time. Objects stored within the blocks are arranged in SoA layout.

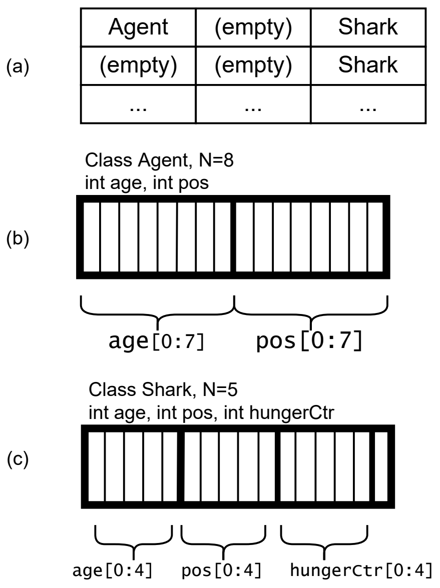

Figure 1 DynaSOAr custom memory layout. (a) Memory heap is split into blocks of fixed size. (b) Block layout for Class Agent. (c) Block layout of Class Shark.

2.3 DynaSOAr fake pointer

DynaSOAr enables dynamic allocation and deallocations at runtime. During allocation, this framework provides *fake pointer* as reference to the object. The fake pointer contains the pointer of the block containing the object and the object's index within the block. Field access for the object is calculated by using these 2 values.

For example, consider the following code:

```
class Agent { int age; int pos; };
A *a;
a->pos = 10;
```

The block that contains class Agent is arranged as illustrated in figure 2. During allocation, object `a` is assigned into a block with address (`*P`) and it is the 4-th object in the block (`I`). These two information are stored within the object's fake pointer. Meanwhile, the maximum number of objects (`N`) is known at compile time, and obtained as a constant.

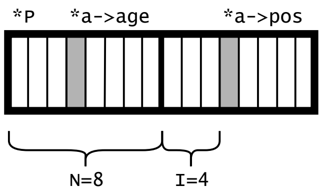

Figure 2 How the block stores member fields of object a of class Agent.

Therefore, field access of `a->pos` is then calculated as:

```
a->pos = *P
       + N * sizeof(age)
       + I * sizeof(pos)
```

3 Problem

It is not trivial to implement multiple inheritance directly on DynaSOAr. The difficulty comes from the fake pointer mechanism described in Section 2.3. Simply reusing that mechanism to extend into multiple inheritance would lead to incorrect result. On the other hand, recreating another mechanism for the pointer operation could significantly make memory access slower.

To illustrate the problem, consider the following code:

```
class Breed { int eggCtr; }
class Agent { int age; int pos; };
class Shark : Agent { int hungerCtr; };
class NewShark: Breed, Agent {
    int hungerCtr;
};

Agent *a; Shark *b; NewShark *c;
a->pos = 10;
b->pos = 11;
c->pos = 12;
```

The block layout for class A, B, and C is illustrated in figure 3. As such, field access for `a->pos` is calculated as:

```
a->pos = *Pa
       + Na * sizeof(age)
       + Ia * sizeof(age);
```

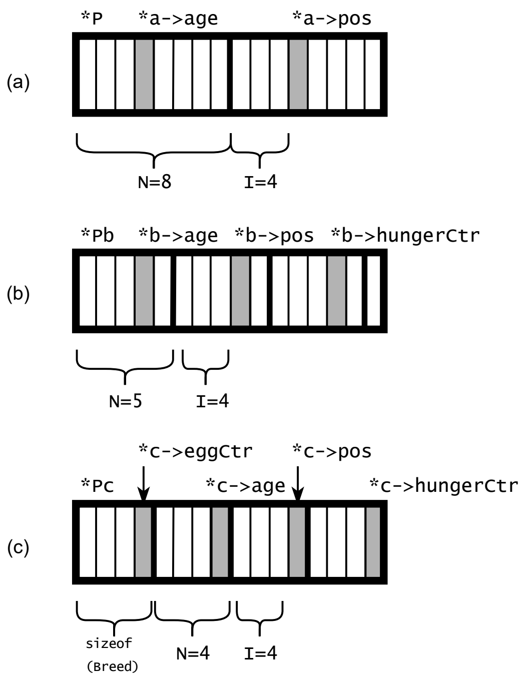

Figure 3 Block layouts for (a) Class Agent, (B) Class Shark, and (C) Class NewShark.

Due to the layout similarity between A and B, field access for `b->pos` is calculated the same way:

```
b->pos = *Pb
       + Nb * sizeof(age)
       + Ib * sizeof(age);
```

From this fact, it can be said that DynaSOAr supports single inheritance. On the other hand, field access for `c->pos` would be calculated as:

```
c->pos = *Pc
       + Nc * sizeof(Breed)
       + Nc * sizeof(age)
       + Ic * sizeof(age)
```

Field access of `pos` leads to ambiguous result. The calculation of `pos` for class NewShark objects is different from class Agent objects. On the other hand, C is a subtype of A: variable of type C may contain either A or C objects at runtime. This leads to a situation where it may be unknown which field access calculation should be used at runtime, thus

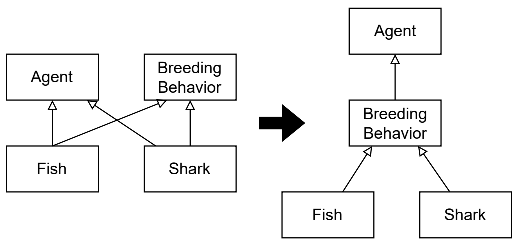

Figure 4 An example of hierarchy transformation.
(a) Original class hierarchy. (b) Transformed class hierarchy

produces an incorrect result.

We also considered to put additional information on the fake pointer in order to set a generic field access calculation through dynamic dispatching. However, access speed is a top priority: using dynamic dispatch to select proper calculation would lower the field access.

4 Hierarchy transformation algorithm

Our solution to support multiple inheritance is to add hierarchy transformation algorithm before Sanajeh's own compilation process. The algorithm transforms user code which has multiple inheritance hierarchy into intermediate code which uses only single inheritance hierarchy (Figure 4). The intermediate code is then further compiled by existing Sanajeh implementation.

The transformation algorithm takes the following 4 steps: remove non-primary parents, reinsert non-primary parents, resolve duplicated classes, and make super calls explicit. The algorithm is illustrated in figure 5.

4. 1 Remove non-primary parents

The first step removes non-primary parents from the given class hierarchy. A primary parent of a class is the parent which has the most priority of preserving its relationship. Users can think of primary parent as the parent that the inheriting class is most resembled to. Users define which of the parent classes is the primary parent by putting it first on the parents list.

In figure 5, class Agent is defined as primary parent for both Fish and Shark (5a). After the removal of non-primary parent (BreedBehavior class), the resulting hierarchy uses only single inheritance (5b).

4. 2 Reinsert non-primary parents

The next step inserts back removed classes into the hierarchy (figure 5c). It reinserts the removed non-primary parent (BreedBehavior) in between the target class (Fish) and its primary parent (Agent). If the same class would be reinserted into 2 different child classes, it may be duplicated as necessary.

A more complex hierarchy may have a class with more than one non-primary parents (Figure 6). In this case, this step linearize the non-primary parents by creating a chain of parent-child relations between them (this process also known as mixin linearization). This chain is then inserted in between the target class and its primary parent.

After this step, we obtain a linearized single inheritance hierarchy, with possible duplicate classes.

4. 3 Resolve duplicated classes

This step attempts to merge duplicated classes to simplify the hierarchy. Duplicated classes are merged if both has the same parent. In our simple example, the BreedBehavior class which got duplicated in previous step is able to be merged (figure 5d).

There are some hierarchies where the duplicated class cannot be merged (figure 7a). Both BreedBehavior classes cannot be merged since they do not have a common parent. Furthermore, there are two cases within this type of hierarchy, based on whether the duplicated class is used as variable type or not (e.g. `BreedBehavior breed`). For the dupli-

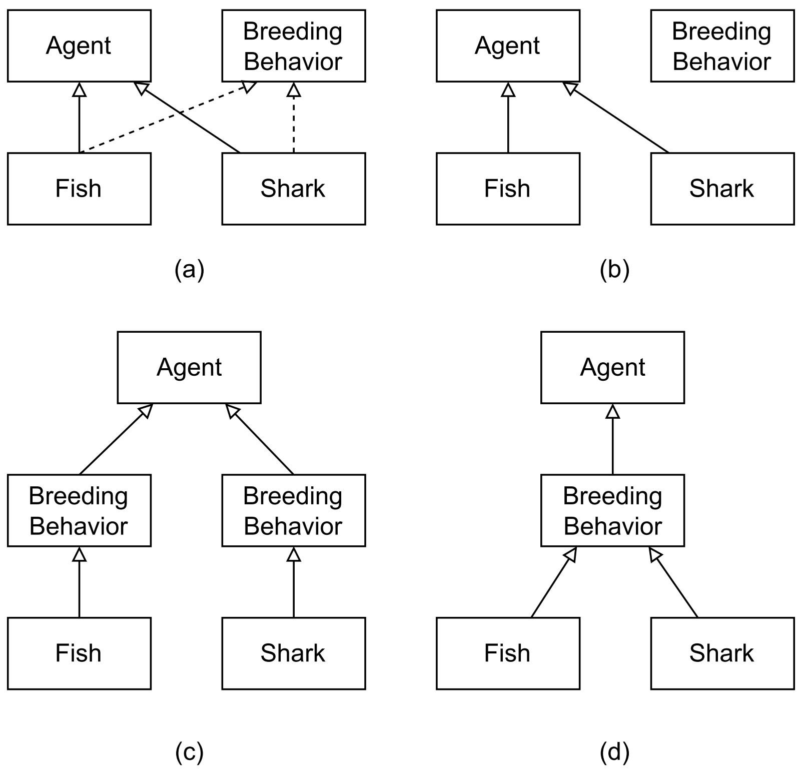

Figure 5 Hierarchy transformation algorithm for a simple multiple inheritance hierarchy.

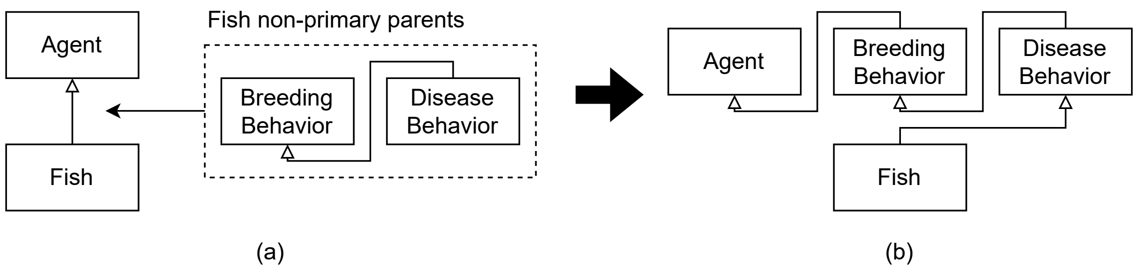

Figure 6 Hierarchy transformation where a class (Fish) has more than one non-primary parents (BreedBehavior and DiseaseBehavior).

cated classes that are not used as variable type, we can leave them as is (figure 7b).

If the duplicated class is used as variable type, this is resolved by using common ancestor method (figure 7c). First, create a new super class named SanajehBaseClass, and make this class as the parent of every class that does not have a parent yet. Next, replace variable type of all duplicated classes (BreedBehavior) into SanajehBaseClass. Finally, set up wrapper functions and up-castings for variable assignments or accesses to dynamically dispatch into its proper type.

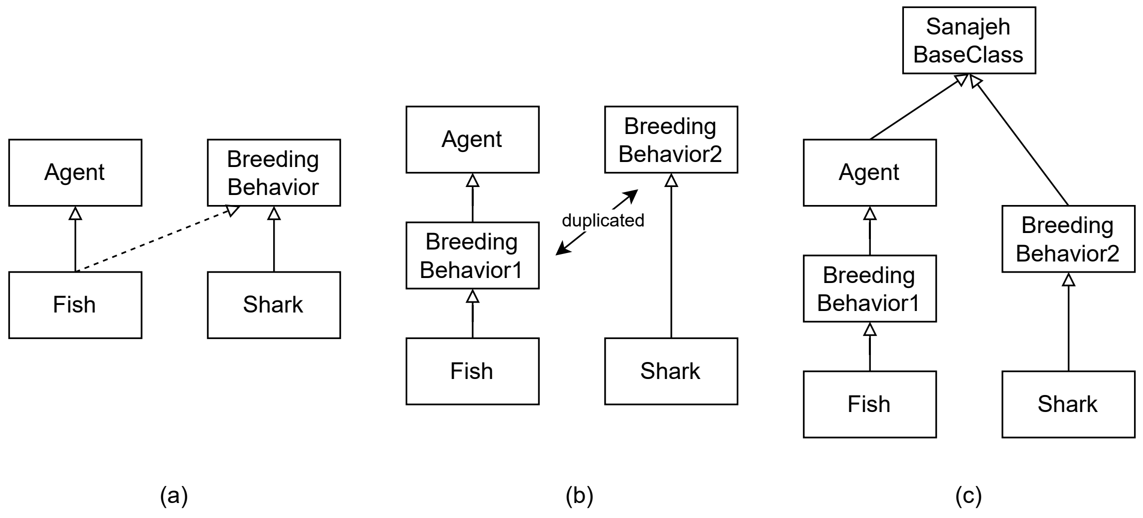

Figure 7 Hierarchy transformation where a duplicated class (BreedBehavior) cannot be merged.

4. 4 Make super-calls explicit

The final step adjusts super calls to its appropriate functions. A class (e.g. Fish) in the modified hierarchy (figure 5d) may have different parent compared to the original hierarchy (figure 5a). For each class that has its parent changed, any super call functions will be replaced into its direct function reference.

5 Results & Discussion

Aspect	Single Inheritance	Multiple Inheritance
Execution time (seconds)	17.644	18.018
Source code size (lines of code)	309	330

Table 1 Execution results.

We tested our implementation as follows. First, we write a benchmark application for DynaSOAr named Wa-Tor in both single inheritance and multiple inheritance version. Next, we run both versions in render mode to check its visual output. After that, we re-run the execution in no-render mode to record their execution times. Finally, we compare both versions in terms of execution time and code length.

We run the simulation using a machine with 12GB memory NVIDIA TITAN Xp GPU. The Wa-Tor simulation parameters are 100x100 cell size and 100 steps for each run.

Our test results are as presented in table 1. The render mode execution showed that both version yields expected result. The no-render test showed that the multiple inheritance version is 2% slower compared to single inheritance version. Source code inspection showed that the multiple inheritance version has more line of code compared to single inheritance version.

The multiple inheritance version is unexpectedly slower than the single inheritance version. We take a look at the intermediary code, and we suspect that the dynamic dispatch increases the execution time. Dynamic dispatch introduces branching, which is discouraged in GPGPU.

The longer source code is within our expectation. Since Wa-Tor has a small hierarchy, the new classes introduces code overhead yet is not being reused optimally. We estimated that if we add more types of agent in this simulation, those new agents will benefit from code reuse, thus will have smaller code length compared to single inheritance version.

6 Related works

There are several projects for expanding GPGPU into high-level languages that supports OOP. Ikra [5] is a GPGPU extension for Ruby, in which another study provides object-support for it [7].

In Python, CUPY [6] and PyCUDA [4] allows GPU utilization. These two libraries uses standard CUDA implementation, as opposed to Sanajeh which uses DynaSOAr, an optimized CUDA framework.

Concord [1] is a heterogeneous C++ programming framework for processors with integrated GPUs designed for general purpose object-oriented programming. It enables GPU execution by using C++, which includes multiple inheritance support. Concord uses native memory allocation which is not optimized for better GPU execution.

Modular Class-based Reuse Mechanisms [9] defines a modular meta-level runtime architecture that converts several code reuse mechanism into single inheritance environment. However, this work is tailored for targeting a Virtual Machine.

7 Conclusion

We presented an implementation of multiple inheritance in Sanajeh, a Python DSL for GPGPU through hierarchy transformation algorithm. This support extends Sanajeh's capabilities for multiple inheritance flexibilities, such as better code reuse and maintainability. However, there is a trade-off between those advantages and execution times.

One area of improvement is the expected output of our algorithm. We designed the transformation algorithm according to mixin linearization. However, in some complex hierarchies, the modified hierarchy may be different than the output of Python's own C3 linearization. Further studies may improve this algorithm by utilizing Python's built-in linearization to determine the parents of a class.

References

[1] Barik, R., Kaleem, R., Majeti, D., Lewis, B. T., Shpeisman, T., Hu, C., Ni, Y., and Adl-Tabatabai, A.-R.: Efficient mapping of irregular C++ applications to integrated GPUs, *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014, pp. 33–43.

[2] Bracha, G. and Cook, W.: Mixin-based inheritance, *ACM Sigplan Notices*, Vol. 25, No. 10(1990), pp. 303–311.

[3] Jizhe, C., Springer, M., Masuhara, H., and Cong, Y.: Sanajeh: A DSL for GPGPU programming with Python objects.

[4] Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., and Fasih, A.: PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation, *Parallel Computing*, Vol. 38, No. 3(2012), pp. 157–174.

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

[6] Nishino, R. and Loomis, S. H. C.: CuPy: A NumPy-compatible Library for NVIDIA GPU Calculations, *31st confernce on neural information processing systems*, Vol. 151(2017).

[7] Springer, M. and Masuhara, H.: Object support in an array-based GPGPU extension for Ruby, *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 2016, pp. 25–31.

[8] Springer, M. and Masuhara, H.: DynaSOAr: a parallel memory allocator for object-oriented programming on GPUs with efficient memory access, *arXiv preprint arXiv:1810.11765*, (2018).

[9] Tesone, P., Polito, G., Fabresse, L., Bouraqadi, N., and Ducasse, S.: Implementing modular class-based reuse mechanisms on top of a single inheritance VM, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 1030–1037.