The MASS CUDA Translator (CSS600 Term Report)

Haobo Peng
penghb@uw.edu

August 25, 2025

1 Introduction

The Multi-Agent Spatial Simulation (MASS) is a library designed for high-performance Agent-based modeling and simulation. It supports high parallelism by splitting the computation tasks over multiple threads on multiple computing nodes. The standard MASS Java and MASS C++ library runs on CPU. Although modern CPUs contains multi-cores to support higher parallel performance, the number of threads that can be "physically" executed at the same time is still limited. To further improve the parallelism capability, the MASS CUDA library is introduced. It is the CUDA GPU accelerated version of MASS that aims at providding similar APIs and programming experience to MASS C++ while delivering better performance. Compared to CPUs, a GPU usually has slower cores, but thousands of them to support thousands of threads running at the same time. As a result, the MASS CUDA library is able to reach a much higher parallelism performance than the standard version [1].

Originally MASS CUDA used Array of Structure (AoS) for storing custom Place and Agent instances on the device memory just like the MASS C++ to maintain similar APIs. However, AoS can cause threads in the same wrap to access uncontinuous memory when accessing attributes of Place / Agent, violating the requirements for coalesced memory access. As a result, the latency of memory access can significantly affect the overall performance. To address this issue, the MASS CUDA library has been updated to use Structure of Array (SoA) instead, which allows coalesced memory access when the threads are accessing attributes of Place / Agent in the device memory and leads to improved performance [2].

Although the performance is improved by adopting SoA, it also leads to significant changes in the API requirements for declaring custom Place and Agent, making the codes less intuitive and more error-prone. To mitigate this issue and improve the programmability of the library, a translator is implemented and introduced in this report to allow developers to write Place / Agent declaration in a cleaner and safer way while the translator will automatically generate additional codes to adopt API requirements of MASS CUDA.

2 Motivation

The original version of MASS CUDA does have relatively intuitive API requirements for declaring Place and Agent by declaring the attributes inside a dedicated state class while the custom Place or Agent hold a pointer to an instance of that class as a member field `states`. With that, attributes can be accessed through `this->states->attr`. Listing 1 shows an example.

Listing 1: Original MASS CUDA Place Declaration Example

```
class MyPlaceState {        // the state class
    float attr1 = 1.0f;
    int attr2[3] = { 0, 0, 0 };
};

class MyPlace : public mass::Place {
public:
    MASS_FUNCTION MyPlace(int index): mass::Place(index) {}
    MASS_FUNCTION ~MyPlace() {}
    MyPlaceState* states;       // pointer to the attributes
    __device__ virtual void callMethod(int functionId, void* arg);
};
```

Now with the SoA adopted in MASS CUDA, the requirement for declaring a custom Place / Agent becomes less intuitive. Attributes can no longer be declared as a member of the state class. Instead, developers needs to invoke the `setAttribute` method on an instance of `Places` or `Agents` with the type, id, length and initial value of the attribute, which will then allocate an array on GPU to store values of that attribute for all the instances of that Place / Agent. To access the attribute, developers should invoke the `getAttribute` method on an instance of `Place` or `Agent` with the id and length of that attribute. For better readability, it is encouraged to declare an `ATTRIBUTE` enum to hold the id of each attribute instead of using integer literal. Listing 2 shows an example.

Listing 2: MASS CUDA with SoA Place Declaration Example

```
class MyPlace : public mass::Place {
public:
    enum ATTRIBUTE {
        ATTR1,
        ATTR2
    };
    MASS_FUNCTION MyPlace(int index): mass::Place(index) {}
    MASS_FUNCTION ~MyPlace() {}
    __device__ virtual void callMethod(int functionId, void* arg);
    __device__ inline void method() {
        float* attr1 = this->getAttribute<float>(
            /* id = */ ATTRIBUTE::ATTR1, /* length = */ 1);
        int* attr2 = this->getAttribute<int>(
            /* id = */ ATTRIBUTE::ATTR2, /* length = */ 3);
    }
};

// in some other file
mass::Places* myPlaces
    = mass::Mass::createPlaces<MyPlace>(/* args */);
myPlaces->setAttribute<float>(
    /* id = */ MyPlace::ATTRIBUTE::ATTR1,
    /* length = */ 1, /* initial = */ 1.0f);
myPlaces->setAttribute<int>(
    /* id = */ MyPlace::ATTRIBUTE::ATTR2,
    /* length = */ 3, /* initial = */ 0);
myPlaces->finalizeAttributes();
```

Such design has several drawbacks listed as follows:

1. The code becomes more verbose.

2. Comparing to declaring attributes as members of state classes using familiar C++ field declaration syntax, the new version is less intuitive, especially for new comers.

3. Attribute declarations are separated from the Place / Agent declaration, leading to less readability.

4. For each attribute, the length and type for invoking `getAttribute` must match those used for invoking `setAttribute`. This must be ensured manually and cannot be checked by the compiler, making the code error-prone.

Besides the difficulties introduced in the new version, the API requirements in original MASS CUDA also include lots of boilerplate codes listed as follows:

1. The `callMethod` method, which is required for routing the method calls. Its implemention is always a big switch-case statement that map the specified function id to the corresponding member method.
2. The constructor and destructor. Becuase they must be annotated with `__device__` and `__host__` (or `MASS_FUNCTION` as a shorthand), they cannot be automatically generated by the compiler even though the body is usually empty.

Such API requirements for declaring Place and Agent is the result of the SoA design and cannot be altered easily. So instead of trying to fully re-design the API, we decided to implement a translator that takes in Place / Agent declaration written in a more intuitive and safer way and automatically generate additional codes to make that Place / Agent to meet the new API requirements. The format and requirement of Place / Agent declaration used for the translation input is discussed in the Design section and the implementation details of the translator is discussed in the Implementation section.

3 Design

As mentioned in the Motivation section, the translator takes in Place / Agent declaration written in a different way that should be safer and more intuitive. The new declaration requirements and formats requires a detailed design with two basic restrictions. First, The declaration should still be a C++ class / struct that inherits from `mass::Place` or `mass::Agent`. In other words, the new design is not a DSL. Second, the declaration should not contains direct attributes as members, which will violate the SoA design.

3.1 Identifying Agent / Place Classes

For the translator to correctly analyze and generate codes, the first step is to find all the Place / Agent classes declaration in the codebase. One possible approach is to find all the classes that inherits from `mass::Place` or `mass::Agent`. However, it is also important to leave the developers a chance do the implementation by themselves for specific Place / Agent classes without being interferred by the translator. As a result, the final design decision is to use C++ annotations, where the translator will only analyze and generate codes for classes annotated with `__MASS_AUTOGEN__` while inherits from `mass::Place` or `mass::Agent`. This way, if the developers decide to fully implement a Place / Agent class all by themselves, they can simply not annotate that class so that the translator will ignore it.

3.2 Attribute Declaration Design

To achieve better readability, the attributes of a Place / Agent should be declared in-place within the declaration body of the Place / Agent class. However, one of the basic design restrictions requires that the attributes must not be direct members of the Place / Agent class. This difficulty is addressed by using a nested states struct inside the declaration body of the Place / Agent class where the attributes will be declared as members of that nested struct. Attributes declared in this way are not direct members of the outer Place / Agent class. So they do not violate the SoA design. In addition, since the states struct is nested, the attributes can still be considered as being declared in-place, providing improved readability compared to declaring them through calling `setAttribute` in separated files. To access the attributes, the translator will automatically generate getters for each of them, which will be discussed latter in the Implementation section.

In the new MASS CUDA that adopts SoA, the `setAttribute` method takes in the initial value of the each attribute. In the new design with the translator, this is supported by specifying a inline initializer for each attribute. For example, for attribute declared as `float attr;` and requires an initial value of `1.0f`, the declaration can be written as `float attr = 1.0f;`.

An attribute may not always be a single value, it can also be an array. Array attributes are also supported in the new design with the translator by declaring a C-style static length array with its length explicitly specified. For example, an array attribute with float elements and length of 10 can be declared as `float arr[10];`. Initial value is also supported with standard array initializer list. However, due to the current limitation of MASS CUDA APIs, it is only possible to specify the same initial value for all the elements of the array. For example, to give an initial value of `1.0f` to all elements of the array attribute `arr` above, use `float arr[10] = { 1.0f };`. It is not an error to write `float arr[10] = { 1.0f, 2.0f }`, but only the first element in the initializer list will be considered.

There is no guarantee that a Place / Agent class will contains no additional nested struct other than the one used for declaring attributes. The translator will need some help to identify the one actually used for attribute declaration. C++ annotation is a good sulution to this concern. The developers are required to annotate the nested state struct used for declaring attributes with `__MASS_ATTRIBUTE__` and when the translator is processing this Place / Agent class, it will focus only on this nested struct to scan for attributes declarations.

Becuase the translator will not alter existed MASS CUDA API requirements for declaring Place and Agent, the `setAttribute` must eventually be invoked to allocate memory on the GPU. For that, all the required invocations to the `setAttribute` method will be generated and wrapped in a static method of the Place / Agent class named `registerToContainer`. The initial values declared for each attributes will be used here while it also provides an overload to accept alternative initial values for each attributes. More details about this generated `registerToContainer` method will be discussed in the Implementation section.

3.3 Method Declaration Design

Method declarations in MASS CUDA with SoA adopted has no difference compared to the original version. However, the `callMethod` method remains to be a heavy boilerplate code, especially when there are many member methods to be exposed. Since the implementation of this method is trivial, it is possible to have the translator to automatically generate it. The detail of generating the implementation will be discussed in the Implementation section, but there remains one problem at design level. That is to identify member methods that needs to be exposed since some of them are just helper methods and are not intended to be exposed by the `callMethod` method. This problem is again addressed by C++ annotations where member methods intended to be exposed should be annotated by `__MASS_METHOD__`. As a result, the translator will only search for member methods with this annotation and include them in the generated `callMethod` method.

3.4 Constructor and Destructor Design

MASS CUDA requires each Place / Agent to have a constructor that accepts a single `int` argument and a destructor, where both of them should be annotated with `__device__` and `__host__` (or `MASS_FUNCTION` as a shorthand). In the new design with the translator, developers are allowed to not defining them. In such case, the translator will automatically generate them.

3.5 Example

Consider a Place class declaration as an example. With the new design, it can be written as presented in Listing 3.

Listing 3: MASS CUDA Place Declaration Example with New Design

```
class __MASS_AUTOGEN__ MyPlace : public mass::Place {
    struct __MASS_ATTRIBUTE__ States {
        float attr = 1.0f;
        float arr[10] = { 1.0f };
    };
    __MASS_METHOD__ void exportedMethod();
    __device__ void helperMethod();
};
```

4 Implementation

The translator is implemented as a external program that takes in multiple C++ source files containing Place / Agent declarations in the new design and outputs generated codes to a specified output directory while inserting some codes to the input source files. Details of the implemention are discussed in the following sub-sections.

4.1 Library Used for Handling C++ Codes and AST

Parsing and handling C++ source codes by hand is not feasible due to the complexity. Therefore, a third-party library is used to help with this task. The one chosen for this project is Clang LibTooling, which is a sub library within the LLVM project. This library provides a set of APIs to parse C++ source codes into Abstract Syntax Tree (AST) and allows developers to analyze the AST for code inspection or code generation, which fits perfectly with the requirements of the trasnlator [3].

4.2 Parsing and Matching Place / Agent Declarations

The translator starts by parsing the input C++ source files into C++ AST. This is done automatically by adopting the `clang::tooling::ClangTool`, `clang::ASTFrontendAction` and `clang::ASTConsumer`. `ClangTool` is the entry point that handles command line arguments and create instances of `ASTFrontendAction` to parse the source codes. `ASTFrontendAction` provides callbacks to handle different life cycle phases when processing each source file. By subclassing it, the trasnlator can initializes and frees certain resources at appropriate time, such as the `Rewriter` and the `DiagnosticsEngine`. Finally the `ASTConsumer` is responsible for receiving the parsed AST and further process it, which require creating a custom subclass and overriding the `HandleTranslationUnit` method.

By adopting the APIs provided by LibTooling above, the translator receives the parsed AST as an instance of `clang::ASTContext`. Such an AST can be very complicated while the translator only needs declarations of Place / Agent annotated with `__MASS_AUTOGEN__`. To find the desired AST nodes, the `clang::ast_matchers::MatchFinder` is used. This class allows specifying a set of rules to match AST nodes and invoke a specified callback to handle the matched nodes. The callback is declared by subclasing `clang::ast_matchers::MatchFinder::MatchCallback` and overriding the `run` method. For this translator, the following four rules are used for matching AST nodes and the actual codes are presented in Listing 4.

- `cxxRecordDecl`: match a C++ class / struct / enum declaration
- `isDefinition`: the declaration must has a body attached
- `isExpansionInMainFile`: the declaration must be defined in the current file
- `hasAttr(Annotate)`: the declaration must has annotations, which does not required to be `__MASS_AUTOGEN__` due to the limitation of `MatchFinder`

Listing 4: AST Matching Rules for Matching Place / Agent Declarations

```cpp
using namespace clang::ast_matchers;
using namespace clang::attr;
matchFinder.addMatcher(
    cxxRecordDecl(
        isDefinition(), isExpansionInMainFile(), hasAttr(Annotate)
    ).bind("class"),
    &processor      // the callback instance
);
```

In the `run` method of the `MatchCallback`, an argument of type `MatchResult` is received, whose `Nodes` field contains the matched AST node. Since the rules used for the translator match only C++ class / struct / enum declarations, it is safe to extract the AST node as a `clang::CXXRecordDecl` for further analysis.

4.3 Declaration Analysis

C++ class / struct / enum declarations are modeled as `clang::CXXRecordDecl` in the AST, which provide plenty of helper methods for extracting information as discussed in the following sub-sections.

4.3.1 Extracting Declaration Name

The name of the declaration can be extracted by invoking the `getNameAsString` method on the `clang::CXXRecordDecl` instance, which is only the unqualified name without any namespaces. To get the fully qualified name, use the `getQualifiedNameAsString` method.

4.3.2 Extracting Annotations

`Decl`, `CXXRecordDecl`, `CXXMethodDecl`, `CXXConstructorDecl` and `CXXDestructorDecl` all provide a `attrs` method to access all the attributes attached to them as a list of `clang::Attr` instances. Not all of them are annotations, so the correct way is to invoke `llvm::dyn_cast` function to try to cast it to a `clang::AnnotateAttr`, then invoke the `getAnnotation` method to get the annotation string, which is used for checking whether the declaration has certain annotation attached.

4.3.3 Extracting Declaration Type (Place or Agent)

The easiest way to decide whether the current declaration is a Place or an Agnet is through the inherited classes. Such classes can be extracted through the `bases` method, which returns a list of `clang::CXXBaseSpecifier` instances.

4.3.4 Extracting The Nested State Structs

There is no API to directly get all the nested structs, so the only way is to invoke the `decls` methods to traverse all nested declarations, try to cast each of them to `clang::CXXRecordDecl` and check whether it includes the `__MASS_ATTRIBUTE__` annotation. If so, it is considered as the nested state struct used for attributes declaration.

To avoid potential conflicts when the developers defines multiple qualified nested structs, the translator will report an error if more than one match is found.

4.3.5 Extracting Member Fields

The translator will only try to find member fields inside the nested state struct annotated with `__MASS_ATTRIBUTE__` since this is where attribute declarations are supposed to be made. The `fields` methods gives all the member fields of that struct as a list of `clang::FieldDecl`. For each of them, the translator will extract the following information:

- The name, through the `getNameAsString` method.
- The type, through the `getType` method.
- The initializer, if any, through the `getInClassInitializer` method
- Whether it is an array, through the `isArrayType` method in `QualType`
- The element type, if it is an array, through casting the type to `ArrayType` and invoke the `getElementType` method.
- The length, if it is an array, through casting the type to `ConstantArrayType` and invoke `getSize` or casting it to `DependentSizedArrayType` and invoke `getSizeExpr`. If it is an array without length (both castings fail), the translator will report an error.

4.3.6 Extracting Member Methods

Information of the member methods, which can be accessed through `methods` as a list of `clang::CXXMethodDecl`, is required for generating the `callMethod` method. From each of them, the following information is extracted:

- The annotations, through `attrs` method. This is used to filter member methods with the `__MASS_METHOD__` annotation, which indicates that this method should be exposed.
- The name, through `getNameAsString` method.
- The parameters (a list of `clang::ParmVarDecl`), through `parameters` method.

In addition, the translator will check whether each exposed method has no parameter or only a single one with type of `void*`. If that is not the case, an error will be reported. Reason for this will be discussed in the Code Generation section.

4.3.7 Extracting Constructors and Destructors

The translator needs to check whether developers has already provided the constructor and the destructor to decide whether to generate them. For the constructor, it can be done by traversing all the constructors returned by the `ctors` method and check whether there is one that accepts a single `int` argument. For the destructor, simply invoke `getDestructor` and check whether it is `nullptr`.

In addition, for the found constructor, the translator will also check whether it is annotated with `mass_autogen_constructor`. Similar for the found destructor, but check for `mass_autogen_destructor` annotation. If so, the translator will still generate them. More details about this will be discussed in the Code Generation section.

4.4 Code Generation

With all the information extracted from the declaration discussed in the previous section, the translator can now generate codes to make the declaration adopt to the API requirements of MASS CUDA. To avoid inserting too many codes into the original source file, most of the generated codes will be written to separated files while only an include statement will be inserted into the body of the original declaration.

4.4.1 Naming Generated Files

For each processed Place / Agent declaration `NS::A` in source file `F.h`, generated declarations will be written into `NS--A.gen.h` (colons are replaced by dashes due to limitation of Makefile) and generated implementations will be written into `F.h.gen.cu`. If the declaration is templated, then both the generated declarations and implementations will be written into `NS--A.gen.h` since templated classes require inline definition.

4.4.2 Generate For Attribute Access

The translator will generate two getter overloads for each attribute declared in the nested state struct, where one takes no arguments for accessing attribute of the current Place / Agent, while the other takes an `int` for accessing attribute of other Place / Agent. Like the `getAttribute` method, it returns a pointer, so that this getter can also be used for updating the attribute value.

Before generating the getters themselves, the translator will first generate an `ATTRIBUTE` enum nested inside the Place / Agent declaration to hold the id of each attribute. The case name for each attribute will be the underscore-separated name of the attribute in uppercase.

Then for each attribute, inline definitions of the getters will be generated with the following format. Using inline definition instead of separating declaration and definition is for better performance.

```
__device__ inline <type>* get<name>() const {
    return getAttribute<type>(
        ATTRIBUTE::<enum_case_name>, getIndex(), <length>);
}
__device__ inline <type>* get<name>(int index) const {
    return getAttribute<type>(
        ATTRIBUTE::<enum_case_name>, index, <length>);
}
```

Codes in angle brackets are placeholders where:

- <type>: the data type of the attribute or the element type if it is an array
- <name>: the name of the attribute in big camel case
- <enum_case_name>: the corresponding name in the `ATTRIBUTE` enum
- <length>: the length of the attribute, which is the length of the array for array attributes or 1 for non-array attributes

4.4.3 Generate *callMethod* Method

Similar to attributes, the translator will first generate a `FUNCTION` enum nested inside the Place / Agent declaration to hold the id of each exposed member method. The case name for each method will be the underscore-separated method name in uppercase.

The generated `callMethod` method always has the same signature as follows:

```
__device__ virtual void callMethod(int functionId, void* arg);
```

As for the definition, it will be a big switch-case statement on `functionId`, where each case matches one exposed member method and invokes it within the case body. If the method require arguments, then the `arg` parameter will be passed to it directly without any type casting. That means an exposed method must accept no argument or a single `void*` argument, which is checked by the translator when extracting information of member methods.

```
__device__
void <class_name>::callMethod(int functionId, void* arg) {
    switch (functionId) {
        case FUNCTION::<enum_case_name>:
            <name>();   // or <name>(arg) if it accepts arguments
            break;
        default: break;
    }
}
```

Codes in angle brackets are placeholders where:

- <class_name>: the fully qualified name of the Place / Agent class
- <name>: the original name of the method
- <enum_case_name>: the corresponding name in the `FUNCTION` enum

4.4.4 Generate For Declaring Attributes

Although attribute declarations are now done by declaring member fields in the nested state struct, the `setAttribute` must still be invoked somewhere to actually allocate memory on the GPU. Without updating implemention in MASS CUDA, such invokations still need to be done manually by the developers. To make it easier and safer, the translator will generate a static method named `registerToContainer` for each processed Place / Agent, which will invoke `setAttribute` for each attribute declared. This static method has two overloads with the following signatures:

```
static void registerToContainer(<container_type>* const container);
static void registerToContainer(
    <container_type>* const container,
    const <state_struct>& defaultValues);
```

Codes in angle brackets are placeholders where:

- <container_type>: either `mass::Place` or `mass::Agent` depending on the type of the current declaration
- <state_struct>: the name of the nested state struct for attribute declarations

The second overload accepts an instance of the nested state struct, which is used to provide alternative initial values for each attribute.

The implementation for this method will invoke `setAttribute` for each attribute in the following format:

```
void <class_name>::registerToContainer(
    <container_type>* const container
) {
    setAttribute<type>(
        ATTRIBUTE::<enum_case_name>, <length>, <initial_value>);
}
void <class_name>::registerToContainer(
    <container_type>* const container,
    const <state_struct>& defaultValues
) {
    setAttribute<type>(
        ATTRIBUTE::<enum_case_name>, <length>,
        defaultValues.<name>);
}
```

Codes in angle brackets are placeholders where:

- <type>: the data type of the attribute or the element type if it is an array

- <class_name>: the fully qualified name of the Place / Agent class
- <name>: the original name of the attribute
- <enum_case_name>: the corresponding name in the `ATTRIBUTE` enum
- <length>: the length of the attribute, which is the length of the array for array attributes or 1 for non-array attributes
- <initial_value>: the initial value to set for the attribute

4.4.5 Generate Constructor and Destructor

If the developers do not provide the required constructor and destructor, the translator will generate them automatically with an empty body. Because they are relatively trivial, they are generated as inline methods without separating the definition.

One concern here is that when the translator is executed again after the constructor and destructor have been generated, it is not able to decide whether the found constructor / destructor is user-defined or generated. To address that, the translator will annotate the generated constructor with `mass_autogen_constructor` and annotate the generated destructor with `mass_autogen_destructor`.

```
__host__ __device__
__attribute__((annotate("mass_autogen_constructor")))
<class_name>(int index): <super>(index) {}

__host__ __device__
__attribute__((annotate("mass_autogen_destructor")))
~<class_name>() {}
```

Codes in angle brackets are placeholders where:

- <class_name>: the fully qualified name of the Place / Agent class
- <super>: either `mass::Place` or `mass::Agent` depending on the type of the current declaration

4.4.6 Inserting Include Statement

Since all the generated codes are written into separated files, they must be included into the body of the original Place / Agent declaration to make them parts of the members. This can be done by inserting an include statement at the end of the declaration body. To avoid inserting the include statement multiple times, a special marker method will also be inserted, so that the translator will skip the insertion when it finds this method.

```
class <class_name> : mass:Place {
    // other declarations
private:
    [[deprecated("an empty marker method, don't invoke it")]]
    inline static void ___mass_autogen_import_marker___() {}
public:
    #include "<class_name>.gen.h"
};
```

4.5 The Annotation Header

All the annotations used in the new design are defined as macros in this `Annotations.h` header. The marker method used for telling the translator to skip inserting the include statement also has a macro defined to make the inserted code cleaner.

```
#define __MASS_AUTOGEN__ __attribute__((annotate("mass_autogen")))
#define __MASS_ATTRIBUTE__ \
    __attribute__((annotate("mass_attributes")))
#define __MASS_METHOD__ \
    __attribute__((annotate("mass_method"))) __device__
#define __MASS_AUTOGEN_MARKER__ \
    private: \
    [[deprecated("an empty marker method, don't invoke it")]] \
    inline static void ___mass_autogen_import_marker___() {}
```

5 Usage

The source codes of the translator can be found in the translator folder in the feature/translator branch of the mass_cuda_core repository. Once merged, it should also be found in the develop branch.

The translator binary needs to be executed in the following format:

```
translator <input-files> --gen-dir=<output-dir> --
   -I<annotation-header-dir> <other-args>
```

Codes in angle brackets are placeholders where:

- <input-files>: the list of source files to process
- <output-dir>: path to the directory to output the generated files
- <annotation-header-dir>: the include search path of the annotation header
- <other-args>: any other arguments to pass to the clang compiler

Detailed instructions of how to build and integrate the translator into projects using MASS CUDA can be found in the README file within the translator folder. A small example is also hosted in translator/example folder demonstrating how to use the translator as an individual package with both CMake and Makefile.

6 Evaluation

The new design with the translator has three categories of metrics for evaluation: programmability, performance and compilation time. Programmability ensures that the new design does result in codes of higher quality, performance ensures that the new design will not hurt the execution performance, and compilation time measures the time taken to compile the whole application.

Evaluation are done with the Heat2D, GameOfLife and SugarScape applications, by modifying the current codes for declaring Place and Agent to use the new design. The platform for running all the tests is the UW Bothell's Juno Linux server equipped with a AMD EPYC 7232P 8-Core Processor and two NVIDIA RTX A5000 GPUs (24G of device memory each).

6.1 Programmability

Programmability is evaluated on all the source files being modified to use the new design with the following two metrics.

- Lines of code (LOC) (lower the better)
- Average Cyclomatic complexity per function (AvgCCN), which indicates the complexity of the code by measuring the number of linearly independent paths through the code (lower the better).

The tool used for measuring these two metrics is `Lizard`, which automatically analyzes source files and counts total lines of codes and average cyclomatic complexity for each function [4].

Results evaluated on the three applications are presented in Table 1, 2 and 3. Note that the LOC for the new design includes the include statement being inserted, so the codes that developers actually write is 3 lines less than the LOC shown in the tables.

According to the results, the new design has significantly reduced the LOC and slightly reduced the AvgCCN. Theoretically, the new design will not change the executaion logic of the codes and should have no effect on AvgCCN. However, the translator remove the need of manually implementing the `callMethod` method, which contains a big switch-case statement that leads to relatively high Cyclomatic complexity. As a result, the AvgCCN is slightly reduced.

Table 1: Programmability Evaluation Results with Heat2D

	Metal.h		Metal.cu		Heat2D.cu	
	old	new	old	new	old	new
LOC	23	20	103	87	157	154
AvgCCN	1.0	0.0	4.0	4.0	4.2	4.2

Table 2: Programmability Evaluation Results with GameOfLife

	Life.h		Life.cu		GameOfLife.cu	
	old	new	old	new	old	new
LOC	19	14	70	51	72	71
AvgCCN	1.0	0.0	3.5	3.3	3.7	3.7

Table 3: Programmability Evaluation Results with SugarScape

	Ant.h		Ant.cu		SugarPlace.h		SugarPlace.cu		SugarScape.h	
	old	new	old	new	old	new	old	new	old	new
LOC	27	18	56	36	39	27	154	117	189	180
AvgCCN	1.0	0.0	2.2	1.5	1.0	0.0	2.9	2.4	12.0	12.0

6.2 Performance

Performance is evaluated by running the three applications in different problem sizes with both the original design and the new design, and measuring the execution time using the timer inside each application. For each problem instance, the application will be executed five times and the average execution time will be recorded.

The problem sizes and other configuration chosen for each application are presented in Table 4 and the measurement results are presented in Table 5, 6 and 7.

According to the results, performance of applications adopting the new design and the translator shows no significant difference compared to the original MASS CUDA, indicating that the new design with the translator will not hurt runtime performance.

Table 4: Problem Sizes and Configurations for Performance Evaluation

	Sizes	Other Configurations
Heat2D	100, 500, 1000, 3000	heat_time=2700, max_time=3000, interval=0
SugarScape	100, 500, 1000, 3000	generations=100
GameOfLife	100, 500, 1000, 3000	seed=-1, step=100, interval=0, timer=true

Table 5: Performance Evaluation Result with Heat2D

	size=100	size=500	size=1000	size=3000
old	5108.4ms	6128ms	9392.8ms	43440.8ms
new	5105.6ms	6121.8ms	9256.8ms	41822.6ms

Table 6: Performance Evaluation Result with GameOfLife

	size=100	size=500	size=1000	size=3000
old	4922ms	4968.4ms	4968.2ms	5383.6ms
new	4950.2ms	4969.8ms	5008.2ms	5383.2ms

6.3 Compilation Time

The compilation time is measured for each application project by running `make build` through the `time` command and recording the `real` time. Note that the compilation time for the new design includes the time taken by the translator to analyze and generate codes.

Results are presented in Table 8, showing that the compilation time for the new design is significantly longer than the old design. This is expected since it takes time for the translator to analyze and generate codes. Besides, the generated source files also increase the compilation time. Although developers have to wait longer for building the project, the safer and higher-quality codes enforced by the new design can help reduce time spent on debugging and maintenance.

7 Conclusion and Future Work

The new design of the MASS CUDA API requirements for declaring custom Place / Agent together with the translator that support it has been shown to help improving the code quality while not affecting the runtime performance, allowing developers to write cleaner and safer

Table 7: Performance Evaluation Result with SugarScape

	size=100	size=500	size=1000	size=3000
old	5078ms	5728.2ms	7924.6ms	31117.4ms
new	5187.6ms	5727.2ms	7935.2ms	31053ms

Table 8: Compilation Time

	Heat2D	SugarScape	GameOfLife
old	28.518s	24.738s	25.391s
new	60.247s	50.966s	52.626s

codes. Although it requires longer time for compiling the application, the higher-quality codes can reduce the time spent on debugging and maintenance, ultimately leading to increased productivity and efficiency.

Currently the translator still have certain limitations that can be addressed in the future. First, it does not support indirect inheritance to `mass::Place` or `mass::Agent`. In other words, if a class declaration inherits from another Place / Agent, the translator is not currently able to identify it as a Place / Agent. This can potentially be addressed by recursively checking the inheritance list of all the base classes in the inheritance hierarchy. Second, if developers change the name of a Place / Agent declaration when the include statement has already been inserted, the translator will not be able to update the include statement accordingly. In this case, the developers will have to manually remove the include statement and run the translator again. Finally, since the translator is implemented as an external program that will generate additional source files, developers have to manually integrate the translator into their build script and include the generated files into the build commands, making the project setup process more complex. This limitation is very tricky to address with the current "external program + annotation" implementation. A potential solution is to switch to the "Clang plugin with custom `#pragma`" implementation, but it requires switching the compilor used for MASS CUDA from GCC to Clang, which may require more effort.

References

[1] L. Kosiachenko, N. Hart, and M. Fukuda, "MASS CUDA: A General GPU Parallelization Framework for Agent-Based Models," en, in *Advances in Practical Applications of Survivable Agents and Multi-Agent Systems: The PAAMS Collection*, Y. Demazeau, E. Matson, J. M. Corchado, and F. De la Prieta, Eds., Cham: Springer International Publishing, 2019, pp. 139–152, ISBN: 978-3-030-24209-1. DOI: 10.1007/978-3-030-24209-1_12

[2] W. Liu, M. Fukuda, K. Sung, and C. Olson, "Programmability and Performance Enhancement of MASS CUDA," *University of Washington, Bothell, Capstone Report, Jun*, 2024.

[3] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665 Accessed: Aug. 17, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/1281665

[4] T. Yin, *Lizard: An extensible Cyclomatic Complexity Analyzer*, Published: Astrophysics Source Code Library, record ascl:1906.011, Jun. 2019. Accessed: Aug. 19, 2025. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2019ascl.soft06011Y