Sanajeh: A DSL for GPGPU programming with Python objects

Jizhe Chenxin   Hidehiko Masuhara   Matthias Springer   Youyou Cong

GPGPU (general purpose computing on graphics processing units) is one of the economical methods of parallel programming. However, in order to obtain high performance, the programmers must write code in a low-level programming language such as C and pay attention to memory allocation. We propose Sanajeh, a Python DSL (domain-specific language) that compiles object-oriented programs into GPGPU code. It is a language which is based on the Single-Method Multiple-Objects (SMMO) model. Sanijeh compiles parallel Python code into C++/CUDA code and utilizes DynaSOAr for efficient GPU memory allocation.

1 Introduction

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

Object-oriented programming (OOP) is a widely used programming paradigm but it is seen as too inefficient for high-performance computing (HPC). Recent work has shown that efficient OOP is feasible on GPUs when following the *Single-Method Multiple-Objects* (SMMO) [11] programming model. In SMMO, programs are designed to run one method on all objects of a class. It fits well with parallel Single-Instruction Multiple-Data (SIMD) architectures. SMMO model allows programmers to write efficient parallel programs in OOP style. The performance of many SMMO applications depends on efficient memory

(de)allocation.

In this paper, we propose a Domain Specific Language (DSL) called **Sanajeh**. Sanajeh provides API (section 4) for programmers to write SMMO applications with high performance in normal Python syntax. Programmers do not have to bother with GPU memory allocation, which is difficult implement efficiently. Also, Sanajeh allows users to use Python libraries in HPC programs. We evaluated Sanajeh's performance with an N-Body simulation (section 7).

2 Background

2.1 DynaSOAr

DynaSOAr [12] is an efficient GPU memory allocator for SMMO applications. It stores data in a Structure of Arrays (SOA) memory layout to improve the efficiency of memory access. However, programmers must adhere to a certain coding style and follow certain coding conventions when using DynaSOAr.

- **Pre-declarations for fields**:
  DynaSOAr requires user to pre-declare all the fields of a class that resides on the device (we call these classes "device classes") through a special syntax (Listing 1). Such code is different from ordinary C++ (Listing 2) and harder to read. Also, all device classes have to be child classes of "`AllocatorT::Base`".
- **Specification for device data**:

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

Listing 1 Field pre-declarations (DynaSOAr) in an N-Body simulation

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

Listing 2 Ordinary C++ field declarations (equiv. to Fig. 1)

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

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

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

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

Listing 3 "__device__" keywords before function declarations in an N-Body simulation.

Since DynaSOAr is a CUDA framework, programmers must annotate functions and certain fields with "__device__" and/or "__host__" modifiers, indicating whether code/data should run/reside on the GPU or on the CPU.

We use DynaSOAr as our device memory allocator.

2.2 Host and device

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

In CUDA programs, users have to write "__device__" keywords (Listing 3). These keywords marked as red are required for data on the device.

2.3 Foreign function interface

Foreign function interface is a mechanism used to call functions written in another language. Sanajeh uses CFFI [9], which is a Python library, to call C++ function from Python. We choose the "in-line", "ABI mode" provided by the CFFI library. Listing 4 shows a example of using CFFI. Declarations like function prototypes are written in "C − likedeclarations", and the path of the compiled shared library (in Sanajeh, a ".so" file) is written in "libpath". By this, users can call C++ function directly through "lib.function_name".

Sanajeh first compiles Python device code into C++/CUDA code, then the code is called through CFFI. Details for using CFFI are hidden by Sanajeh API (section 4) so users does not need to write such code.

```
import cffi

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

**Listing 4　Example of using `CFFI`, "in-line".
"ABI mode"**

3　Overview

In this section, we discuss two challenges in Sanajeh.

3.1　Distinct host code and device code

In Sanajeh, code also includes host code and device code (2.2). Host code in Sanajeh is executed directly by the Python interpreter, but device code needs to be compiled into C++/CUDA (3.2). Sanajeh is designed to free users from explicitly specifying device data by writing code with extra syntax like the "__*device*__" keywords in Listing 3. The recognition of device code is an issue in Sanajeh.

We use `CallGraphAnalyzer` to solve this issue. `CallGraphAnalyzer` tracks all code related to device classes and marks them as device code. It only requires the specifications of device classes. `CallGraphAnalyzer` will be described in section 5.

3.2　Compile device code

Python is a highly abstracted language. In order to gain high performance in SMMO applications, we have to use a low-level programming language.

In Sanajeh, device code is compiled into C++/CUDA by Ahead-Of-Time compilation. The compiled code includes DynaSOAr syntax for calling DynaSOAr API to do memory allocation. As shown in Listing 1 and Listing 2, DynaSOAr's syntax is different from those in usual programming style. Fields in device classes need to be pre-declared through DynaSOAr syntax, which is defined using C++ templates.

It is obvious that DynaSOAr's syntax makes programs more complex. Sanajeh users can write device code in normal programming style in Python. We uses a compiler called `Py2Cpp` to do the compilation from Python to C++/CUDA. It is described in section 6. We then compiles those C++/CUDA code generated by `Py2Cpp` into a shard library using the Nvidia CUDA Compiler (NVCC), and calls them through FFI (section 2.3)

4　Sanajeh's API

Sanajeh provides API separately for device code and host code.

4.1　Device API

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

- `device_do(cls, func, *args)`: Run function `func` on all objects of class `cls` sequentially with the arguments of `func`. Here `func` has to be a function declared in the device classes.
- **Random API**: Random API is provided for random number generation, for example the `rand_init` function and the `rand_uniform` function. Corresponding to the cuRAND library.
- **Math API**: Math API is provided for Math computation, for example the `sqrt` function. Corresponding to the CUDA Math API.

4.2　Host API

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

- `initialize()`: Load the above shared library to Python FFI module.
- `device_class(*clss)`: Specify the device classes through variadic arguments `*clss`.
- `parallel_do(cls, func, *args)`: Run function `func` on all objects of class `cls` parallelly with the arguments of `func`. Here `func` has to be a function declared in the device classes. Notice that there is no `parallel_do` API in device API (section 4.1), since device code is executed in units of one thread and parallelism cannot be applied further more.
- `parallel_new(cls, object_num)`: Create objects of a class `cls` on device memory with a number of `object_num`.
- `do_all(cls, func)`: Run function `func` on copies of all objects of class `cls` sequentially.

Here `func` is a function which receives an object as an argument. The object is a copy of an object of class `cls`. Functions like `print` can be used in `func`.

5　CallGraphAnalyzer

`CallGraphAnalyzer` is designed to free users of Sanajeh from explicitly specifying all host code and device code. It uses the abstract syntax tree (AST) of Python source code as input, tracks calling relationships between functions and using of variables, then creates a `CallGraph` data structure (section 5.2). After user specifies the device classes, by analyzing the `CallGraph`, all device code will be marked.

5.1　Definition of Sanajeh's device code

Here are the definitions of *Sanajeh device code*:

1. Declarations of the device classes is *device code*. These are specified by users.
2. Declarations for the functions and variables called/used in those classes are *device code*.
3. Declarations for the functions and variables called/used in above functions are *device code*. This is applied recursively.

5.2　CallGraph data structure

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

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

Fig.1 showed an example of the `CallGraph` structure for the code in Listing 5. For illustration

```
global1 : int = 5
global2 : int = 10
global3 : int = 15

class HostClass :
    def HostFunction ( self ) -> None :
        Function3 ( )

class DeviceClass :
    def DeviceFunction ( self ) -> None :
        self . field1 = global1 + 1
        self . field2 = Function1 ( )

def Function1 ( ) -> int :
    Function2 ( )
    return 0

def Function2 ( ) -> None :
    global global3
    local1 = 0
    global3 += local1

def Function3 ( ) -> None :
    local2 = 1
    global global2
    global2 += local2
```

Listing 5　A Sanajeh sample program

purpose we do not mark all calling/using relationships between nodes but only those between device nodes.

5.3　Marking of device data

After the `CallGraph` structure is created, devices code will be marked by tracking the declaration, calling and using relationships between the nodes.

As shown in Fig.1, since `B` is a device class, by tracking the declarations in class B: B_F, `field1` and `field1` are marked. During the marking, functions called and variables used in those nodes are also marked. In this example, `Function1` is called in B_F so it was marked. Recursively, `Function2` is marked. Then since `local1` is declared and used in `Function2`, global variable `global3` is used in `Function2` , they are marked as well. `global1` is also marked because it was used in B_F.

6　Py2Cpp

`Py2Cpp` is a compiler which compiles Python device code to C++/CUDA code. It uses the check result of `CallGraphAnalyzer` (section 5) to trans-
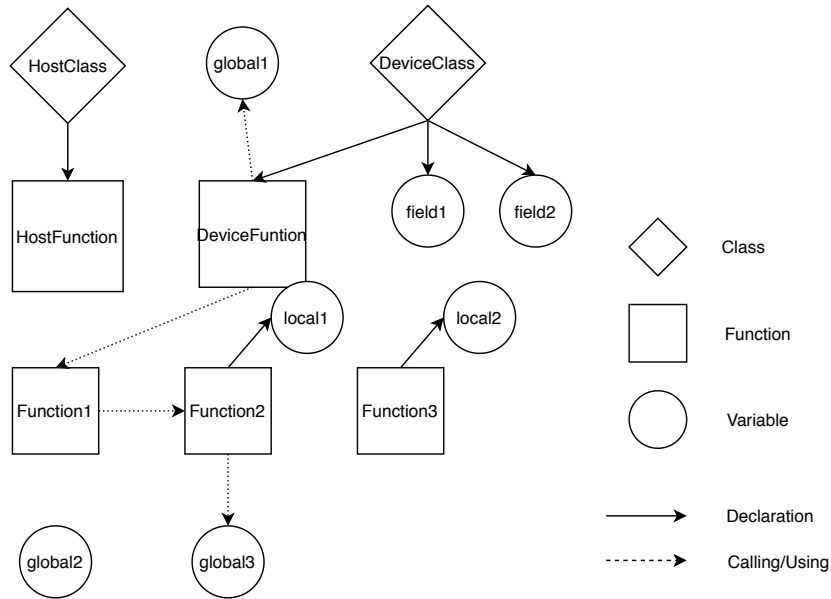
Fig. 1 `CallGraph` **of Listing 5**

form the Python AST of device code to C++ AST code (section 6.1). Then C++ code will be builded from C++ AST (section 6.2)

6.1 Transform Python AST to C++ AST

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

Data types is an important issue during the transformation. Type annotations are supported from Python 3.6. Although Python runtime does not enforce function and variable type annotations, since type information is important for memory allocation, Sanajeh requires users to explicitly write type hints for device data (as shown in Listing 5). There is a type converter in `Py2Cpp`. For now it only supports `bool`, `int` and `float` data types, but we are going to extend it to support other data types in the future, for example arrays.

6.2 Build C++ code

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

- **Pre-declarations of fields**:
  Pre-declarations of fields are generated from class AST nodes in C++ AST. code in Listing 1 can be generated automatically.

- **code for API callings**:
  API calling in DynaSOAr uses C++ templates to pass argument like class and function. Although Python has mechanism to pass class and function as an argument, templates cannot be called through FFI (section 2.3) except for functions. We generate a function for each class. The function includes C++ template, which is used to call DynaSOAr API. For example, Listing 6 shows this the function for calling `parallel_new` on class `Body`.

- **code for callback**:
  In Sanajeh's API (section 4), there is a `do_all` function. This function receives a Python function, say `PF`, while `PF` receives an Python object as a parameter. The constructor function of the class, say `CF`, will be included together

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

Listing 6   Function for calling `parallel_new` on class `Body`

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

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

Listing 7   `_do` and `do_all` function of class `Body`

with `PF` in such an lambda expression:

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

$\{*args\}$ is the fields of the device class. This lambda expression, say `lexp`, is passed to C++ as a callback function. In c++, there is a `do_all` function for every class that receives `lexp`, and passes it to another function `_do` using DynaSOAr API. In `_do` function, all fields of the class is passed to `lexp`. As a result, `lexp` is called multiple times in C++ territory until copies of all device objects are created by `CF` and passed to `F`. Finally, the fields of those copy objects are accessed in `F`. Example code for `_do` and `do_all` is shown in Listing 7. `Py2Cpp` collects the information of fields of device classes then generates such code automatically.

7   Performance Evaluation

To see how calling device code through FFI module affects overall performance, we evaluated the performance of Sanajeh compared to DynaSOAr by an N-Body simulation program.

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

We measured the overall execution time of N-Body simulation with 100 iterations for different number of objects in Sanajeh and DynaSOAr on NVIDIA TITAN Xp GPU (12 GB device memory). We compiled the device code with NVCC (-O3) from the CUDA Toolkit 10.1 on Ubuntu 18.04.4. Host code was run with Python 3.7.4. Fig.2 shows the result of the execution time. The vertical axis is execution time in seconds, and the horizontal axis is the number of bodies.

In conclusion, the execution time of each iteration in Sanajeh and DynaSOAr has no much difference. The increase percentage of execution time from Sanajeh's to DynaSOAr's is no more than 1.5%. It is conceivable that this extra time is the execution time of FFI module. It does not influence much on overall performance.

We measured the compile time spent by `Py2Cpp`. `Py2Cpp` spent an average time of 0.006s to compile N-Body's Python code into C++/CUDA ones. It will take up less and less up proportion of overall execution time when body objects increases.

8   Related work

There have been high-level DSLs for GPGPU programming. Such as *SPOC* [2], which uses stream processing with Ocaml, and *Ikra* [5], which is a data-parallel extension to Ruby. In Python, Klöckner et al. designed *PyCUDA* and *PyOpenCL* [4], two Python libraries for GPU computing based on CUDA. They are designed for experienced CUDA users and require users to write code in CUDA syntax as a string. This is not friendly to a Python programmer who doe not understand C++/CUDA syntax. Another library for GPU computing is called *CUPY* [6]. Its syntax is close to *NumPy* [7], a widely-used Python library for data computing. It provides the same functionality as
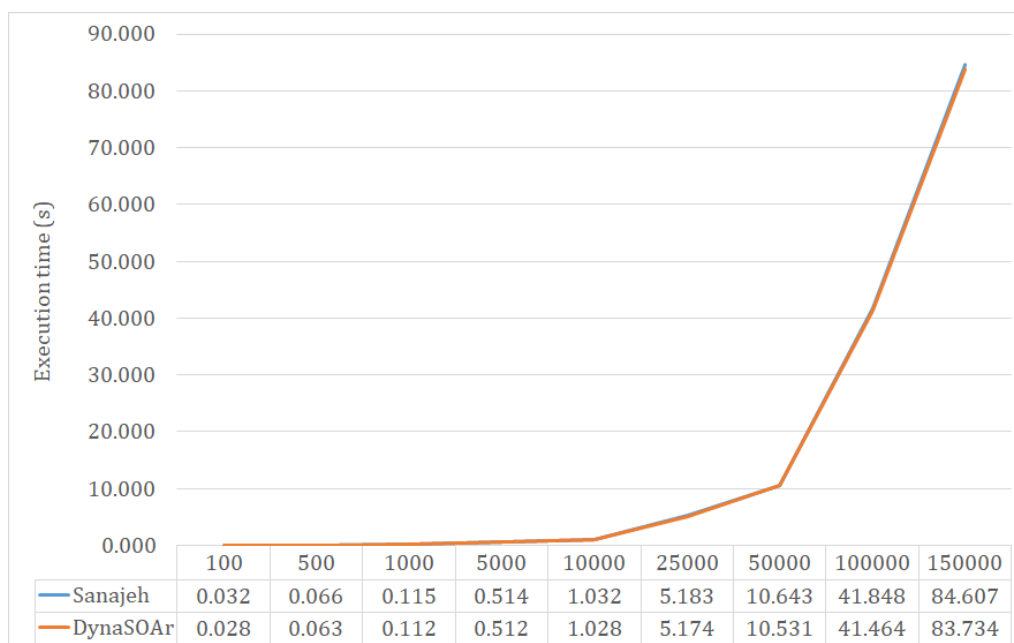
	100	500	1000	5000	10000	25000	50000	100000	150000
Sanajeh	0.032	0.066	0.115	0.514	1.032	5.183	10.643	41.848	84.607
DynaSOAr	0.028	0.063	0.112	0.512	1.028	5.174	10.531	41.464	83.734

Fig. 2　Execution time of N-Body simulation with 100 iterations

NumPy, but executes on a GPU.

Although the above libraries have good performance in most of the cases in GPGPU programming by Python, they are not designed and optimized for OOP. In Python, a list of objects is stored as an Array of Structures (AOS) [3][8]. Previous works [1][10][13] have shown that switching to a Structure of Arrays (SOA) data layout can speed up HPC applications by several factors compared to a traditional AOS layout. Therefore, Python's implementation of a list of objects is not a good choice for GPU programs. Our work uses DynaSOAr [12] as memory allocator, which allocates memory dynamically with SoA performance characteristics.

9　Conclusion

We proposed a DSL for GPGPU programming with Python objects called Sanajeh. Sanajeh provides API for users to write SMMO applications. By compiling device code to C++/CUDA code automatically, Sanajeh users can write GPGPU in OOP style using a high abstracted language, Python, in the meanwhile without losing performance.

References

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

[2] Bourgoin, M., Chailloux, E., and Lamotte, J.-L.: SPOC: GPGPU programming through stream processing with OCaml, *Parallel Processing Letters*, Vol. 22, No. 02(2012), pp. 1240007.

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

[4] Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., and Fasih, A.: PyCUDA and Py-OpenCL: A scripting-based approach to GPU runtime 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*, (2017), pp. 151.

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

[8] Phillips, D.: *Python 3 object-oriented program-*

ming, Packt Publishing Ltd, 2015.

[9] Rigo, A. and Fijalkowski, M.: CFFI documentation, 2012.

[10] Springer, M. and Masuhara, H.: Ikra-Cpp: A C++/CUDA DSL for Object-Oriented Programming with Structure-of-Arrays Layout (WPMVP' 18), *ACM, Article*, Vol. 6, No. 9(2018).

[11] Springer, M.: Memory-Efficient Object-Oriented Programming on GPUs, *arXiv preprint arXiv:1908.05845*, (2019).

[12] 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).

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