Marius Script: Message Passing Interface for the MariusGNN framework

Venkata Saikiranpatnaik Balivada Agam Dwivedi

July 8, 2023

Abstract

There is a steep learning curve to writing new layers in *MariusGNN* [12], the SoTA Graph Neural Networks (GNNs) framework. To that end, we introduce a new pythonic scripting interface called `Marius Script` to ease the development of new GNN Modules on *MariusGNN*. Moreover, we build a compiler tool, `mpic`, so that users can develop their algorithms using the intuitive Message Passing abstraction and target the *MariusGNN* backend. As a result, we hope that `Marius Script` increases user adoption of the *MariusGNN* framework by not only reducing the surface of bugs but also flattening the learning curve by keeping the interface similar to that of the popular Amazon's *Deep Graph Library (DGL)* [13]. Finally, we evaluate the performance and accuracy of `Marius Script` against the vanilla *MariusGNN* framework to prove code equivalence. Our *GraphSage*[6] implementation demonstrates that it is much easier to develop modules using `Marius Script` than otherwise. `mpic` is open-sourced as a *MariusGNN* tool.

1 Introduction

Graph Neural Networks (GNNs) are a recent, emerging incarnation of Neural Networks for graph-structured data. GNNs have applications in numerous tasks such as protein classification [6], content recommendation [7], and inferring emergent properties of new molecular structures [4]. Graphs are unique in that they not only represent the individuals but also the relationships between them. Graphs naturally occur in many real-world scenarios. One such example is a social network where individuals are represented as nodes, and the connections between them are represented as edges. Likewise, in the biological domain, individual proteins are represented as nodes and the interactions between the proteins are represented as edges[1]. GNNs have State-of-The-Art performance when it comes to such graph-based predictions.

Unlike traditional NNs, GNNs store huge amounts of data. It is not uncommon to find graphs that have millions of nodes and billions of edges. On top of this, target vertices of GNNs sample multi-hop neighborhoods[2], exacerbating the memory requirements. To combat these heavy resource requirements, popular frameworks such as DGL and PyG deploy multiple GPUs[3] to execute their models. Such an approach is resource inefficient and produces sub-linear speedups. Instead, *MariusGNN* a SoTA GNN framework, uses SSD storage for out-of-core GPU pipeline processing. In summary, *MariusGNN* uses SSDs to store node features and predictively loads subgraphs to overlap computation with communication more effectively. The pipelining approach in *MariusGNN* leads to extremely high GPU utilization thus drastically reducing resource costs. Additionally, *MariusGNN* stores its sub-graphs in a compact representation called *DENSE* for a more efficient model execution on the GPU, thus further reducing[4] resource costs.

New users struggle to write new layers in Marius for various reasons. First, is the programming language barrier. Many researchers are more comfortable with python than C++ as can be seen in figure 1. Python is less verbose, easier to use, and more popular amongst researchers. However, *MariusGNN* only provides a C++ interface for developing new layers. Next, the *DENSE* representation of Marius, while it produces a more efficient execution of the model, is not intuitive to new users.

On the other hand, popular frameworks such as DGL employ the Message Passing interface to abstract away low-level details about graph operations.

[1]One such example is kinetic interactions

[2]An n-hop neighborhood is the set of all nodes that are n hops away

[3]These multiple GPUs can also be deployed on the same machine

[4]The reduction is multiplicative

In this project, we aim to combine the benefits of both the DGL and the *MariusGNN* frameworks to provide a simple Message Passing abstraction to users while also preserving the resource and run-time efficiency of *MariusGNN*. To that end, we develop a compiler, `mpic`, that translates DGLesque code to C++ source files that *MariusGNN* understands. In this report, we,

- Discuss the preliminaries on GNNs, Message Passing Abstraction, and Marius in section 2
- Discuss the interface provided by `Marius Script` in section 3
- Discuss the high-level architecture of `mpic` in section 4
- Demonstrate compiler correctness in section 5
- Propose potential work for the future in section 6

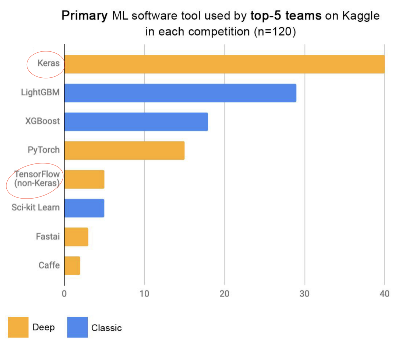


Figure 1: Popularity comparison between frameworks

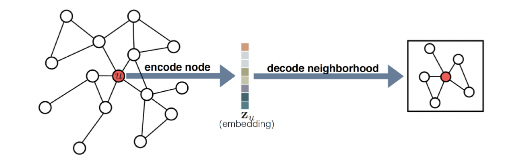


Figure 3.2: Overview of the encoder-decoder approach. The encoder maps the node u to a low-dimensional embedding $\mathbf{z}_u$. The decoder then uses $\mathbf{z}_u$ to reconstruct u's local neighborhood information.

Figure 2: Illustration of the Encoder/Decoder Model

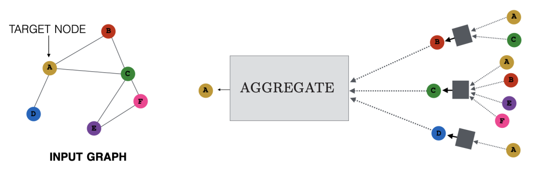


Figure 5.1: Overview of how a single node aggregates messages from its local neighborhood. The model aggregates messages from `A`'s local graph neighbors (i.e., `B`, `C`, and `D`), and in turn, the messages coming from these neighbors are based on information aggregated from their respective neighborhoods, and so on. This visualization shows a two-layer version of a message-passing model. Notice that the computation graph of the GNN forms a tree structure by unfolding the neighborhood around the target node.

Figure 3: Illustration of Message Passing

2 Preliminaries

2.1 Comparison with traditional approaches

GNNs differ from traditional approaches in two major ways [5]. First, GNNs operate on graph-structured data. Traditional neural networks have always operated on simpler structures of data. Take the Convolutional Neural Networks (CNNs) for example. CNNs became popular for their use in Computer Vision tasks. Images are often structured in a grid-like format and there is almost no asymmetry between various pixels in terms of their neighborhood structure. This means that a simple window scan of the image is often sufficient to feed into the network. On the other hand, graph vertices (unlike pixels) can vary a lot in their neighborhood structure. This structure makes the problem fundamentally different from grid-like data, hence capturing more real-world problems where the iid(Independent Identically Distributed) assumption does not hold. Next, classical graph algorithms employ carefully engineered algorithms and attempt to provide classical guarantees on the correctness of algorithms, deterministic or otherwise. Such an approach is often impractical in the modern age of Big Data where the target functions are often too complex to be designed by a human. Learning methods, on the other hand, can infer these functions for us, thus providing extraordinary flexibility. This combination is what makes GNNs so powerful and popular.

So, how do GNNs work and what are they useful for? The first class of applications is node classification tasks. For example, consider Facebook, a popular social media website. Say, we want to identify bot (faux) users on the website. To achieve this task, we can construct a graph where users are nodes of the graph and the connections or interactions are the edges. Given a few human-labeled bot users, we can train our graph to compute the model weights of the GNN. These weights are computed in a back-propagation step on a network built using a sampled multi-hop neighborhood of the target nodes. This model can in turn be used to predict bot users in a separate inference step. Similarly, GNNs can also be used for link predictions. Say, Facebook wants to push friend recommendations to users. We can encode existing nodes (users) to a low-dimensional vector embedding. Proximal nodes in the said embedding are more likely to become friends, thus identified in a decoding step, see figure 2. The embeddings are trained by defining a loss functions on the existing positive and negative edges.

2.2 Message Passing Abstraction

Consider the problem where we want to find the minimum node value of a graph. In a message passing approach [4], each vertex sends its current minimum to each of its neighbors in a message step. Then, in a reduce step, each vertex collects messages from its neighbors and stores the minimum. This process repeats until all nodes reach the fixed point. This particular abstraction is widely employed in contemporary graph algorithms because of the immutable nature of messages. The abstraction also makes it easier to think about graph algorithms in a distributed setting. So, how does MPI (message passing interface) fit into GNNs? As mentioned previously, GNNs sample a multi-hop neighborhood of its target nodes to gather and reduce information from its neighbors. MPI is famously used to specify these vertex interactions, see figure 3. A typical message passing phase is as follows [5]

$$m^k_{\mathcal{N}(u)} = \text{AGGREGATE}^k(\{\mathbf{h}^k_v, \forall v \in \mathcal{N}(u)\}) \quad (1)$$
$$\mathbf{h}^{k+1}_u = \text{UPDATE}^k(\mathbf{h}^k_u, m^k_{\mathcal{N}(u)}) \quad (2)$$

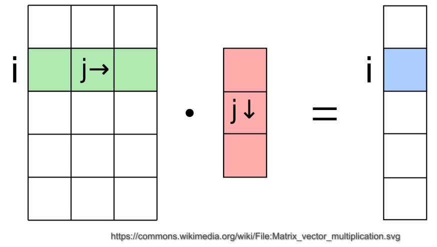
For example, in a page rank step, the AGGREGATE function sums all the values from its neighbors. While the UPDATE function first computes $0.15 + 0.85 * \mathbf{h}$[5] and then we divides it by the number of outgoing neighbors.

2.3 DGL Example - Graph Sage

In this subsection, we discuss the implementation of a simple graph convolution layer called GraphSage [6]. The AGGREGATE function is simply the average of all of its neighbors, while the UPDATE function applies an activation function after a linear transformation over both the current node features and the aggregated neighbor features, see 4.

$$m^k_{\mathcal{N}(u)} = \text{AVERAGE}^k(\{\mathbf{h}^k_v, \forall v \in \mathcal{N}(u)\}) \quad (3)$$
$$\mathbf{h}^{k+1}_u = \sigma(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}^k_u, m^k_{\mathcal{N}(u)})) \quad (4)$$

Now, let us dive into the DGL code. Every DGL module inherits from torch's $Module$ class either directly or indirectly, see line number 3 in 1. Next, we need to register learnable weight parameters for $\mathbf{W^k}$ in 4. We do this by registering a torch *Linear* module, see line number 6[6]. Line number 11 specifies the aggregation function using the graph's *update_all* API. Finally, the update happens in line number 16 where we perform the said linear transformation[7].



i
j→
j↓
i
https://commons.wikimedia.org/wiki/File:Matrix_vector_multiplication.svg

Figure 4: Sum all the vertex neighbors

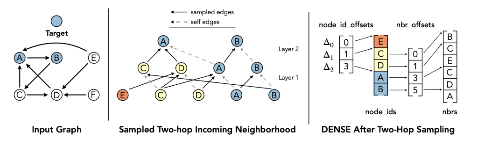


Figure 3. Example two-hop neighborhood sample for target nodes $\{A, B\}$ and the corresponding DENSE data structure.

Figure 5: DENSE Representation

2.4 Marius Graph Representation

Before we dive into how the same layer can be implemented in *MariusGNN*, we discuss some prerequisite details about how *MariusGNN* represents its graph. Recall that *MariusGNN* uses a compact representation for its sampled sub-graphs for a more

[5]Here we assume that α is 0.85

[6]We need twice the number of input parameters here because we transform both the node features as well as the aggregated features

[7]Note that we skip the activation function here

```python
import dgl.function as fn

class SAGEConv(nn.Module):
    def __init__(self, in_feat, out_feat):
        super(SAGEConv, self).__init__()
        self.linear = nn.Linear(in_feat * 2, out_feat)

    def forward(self, g, h):
        with g.local_scope():
            g.ndata['h'] = h
            g.update_all(
                message_func=fn.copy_u('h', 'm'),
                reduce_func=fn.mean('m', 'h_N'))
            h_N = g.ndata['h_N']
            h_total = torch.cat([h, h_N], dim=1)
            return self.linear(h_total)
```

Listing 1: *SAGEConv* DGL Example

efficient model execution. This is problematic because code written using the Message Passing abstraction never executes by actually sending messages. Doing that would be too inefficient. Instead, they are transformed into efficient matrix operations on the GPU. For instance, say we want to sum all the vertex neighbors in the 'aggregate' phase. If the graph is represented using an adjacency matrix, we can exploit GPU parallelism better by multiplying the adjacency matrix with the node feature vector, see 4. Clearly, we need a translation layer to convert code written for the Message Passing interface into operations on the *DENSE* representation.

At a high-level, *DENSE* minimizes memory requirements by reusing already sampled nodes, see figure 5. In summary, the *node_ids* data structure points to an array of nodes at that particular layer, and the *nbr_offsets* array points to the list of neighbors. Observe that node_ids are reused across layers. See [12] for more information.

2.5 Marius Example - Graph Sage

In this section, we highlight the important deviations from the DGL implementation of the SAGEConv layer. First, we used Python to implement SAGEConv in DGL, however, we have to use C++ for *MariusGNN*. Second, DGL averages features only from the incoming neighbors but we average features from both the incoming as well as the outgoing neighbors in *MariusGNN* for better performance. Last but most importantly, to sum our neighbors, we *index_select* the node features using the *nbrs* array to get features for each element in the array. Then, we sum the segments made by the *nbr_offsets*, resulting in the required sum. Finally, we use *index_select* and *segment_sum* in particular since they have efficient GPU implementations to exploit parallelism.

3 Marius Script Interface

To tackle the issues mentioned in the previous section, we designed a `Marius Script` interface. We carefully design a minimal viable interface since developing a full-blown compiler is out of the question given the scope of our project. We list the full grammar of `Marius Script` at 1. See 2 for our implementation of the SAGEConv layer. In this section, we provide a brief explanation of the implementation.

First, observe that there are no import statements. Currently, `Marius Script` only supports builtins from the 'mpi' package which is imported implicitly. Hence, we do not support any import statements at the moment. Next, every GNN module must import from mpi's *Module*, see line number 1. Moreover, `Marius Script` requires type annotations on function arguments and nowhere else, see line numbers 2 and 10. There are three important functions that `Marius Script` recognzies, namely *__init__*, *reset_parameters* and *forward*. The type signatures of the reset_parameters and the forward functions are fixed while the __init__ function accepts additional arguments to be specified which count as additional configuration options. The __init__ function also requires that the user pass the input and output dimensions of the module to the base Module class, see line number 3.

```
module MariusScript {
    mod = Module(stmt* body)
    stmt = FunctionDef(identifier name,
             arguments args,
             stmt* body,
             expr returns)
        | ClassDef(identifier name,
             expr base,
             stmt* body)
        | Return(expr? value)
        | Assign(expr target, expr value)
        | Expr(expr value)
        | Pass
    expr = BinOp(expr left, operator op, expr right)
        | Call(expr func, expr* args, keyword* keywords)
        | Constant(constant value)
        | Attribute(expr value, identifier attr)
        | Subscript(expr value, constant slice)
    operator = Mult
    arguments = (arg* args)
    arg = (identifier argument, expr annotation)
    keyword = (identifier argument, expr value)
    withitem = (expr context_expr)
}
```

Listing 1: Marius Script Abstract Grammar

```python
class GraphSageLayer(mpi.Module):
 def __init__(self, input_dim: int, output_dim: int):
    super(mpi.Module, self).__init__(input_dim, output_dim)
    self.linear = mpi.Linear(input_dim * 2, output_dim)
    self.reset_parameters()

 def reset_parameters(self):
    self.linear.reset_parameters()

 def forward(self, graph: mpi.DENSEGraph, h: mpi.Tensor)->mpi.Tensor:
    with graph.local_scope():
        graph.ndata["h"] = h
        graph.update_all(
            message_func=mpi.copy_u("h", "m"),
            reduce_func=mpi.mean("m", "h_N")
        )
        h_N = graph.ndata["h_N"]
        h_total = mpi.cat(h, h_N, dim=1)
        return self.linear(h_total)
```

Listing 2: ***graph_sage_layer*** implementation using Marius Script

Next up is the mpi's *Linear* module. This has a similar interface to the torch's Linear module and is used to transform inputs linearly. Notice that no variable declarations or type annotations were required to register 'linear' as a member variable. This is possible despite C++ requiring static variable scopes and static types because `mpic` uses type inference to register new local and instance variables. However, `mpic` rejects variables whose type or scope cannot be inferred. Also, users cannot initialize new member variables outside the __init__ function.

Finally, we discuss the main details of the *forward* function. The top-level *with* statement contains any modifications to the graph to the local scope. `Marius Script` requires that all graph modifications be done in the local scope. The UPDATE step of Message Passing happens using the return value of the forward function. Next, we discuss the update_all call. We support builtins for message and reduce functions and leave supporting user-defined functions for future work. mpi's *copy_u* and *mean* functions have semantics similar to DGL's builtin functions. The rest is similar to the DGL code to keep learning overhead to a minimum.

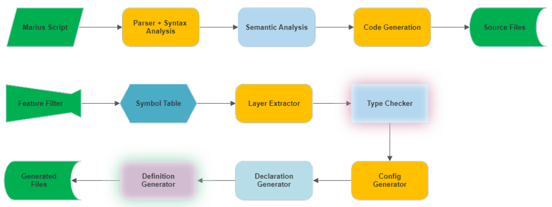


Figure 6: MPIC Architecture

4 Marius Script Compiler

In this section, we lay out the high-level details of our compiler implementation. We use a standard compiler architecture pipeline starting with a syntax analysis phase, followed by a semantic analysis phase and finally, a code generation phase, see 6. We use python's builtin *AST* [1] module to parse `Marius Script` code. As with most compilers, `mpic` runs multiple passes over the generated AST as detailed in figure 6. These passes leverage the visitor pattern to recursively process the generated AST. Now, we describe the important parts of the pipeline.

Since `Marius Script` exposes a minimal interface, proper error reporting is essential. In the syntax analysis phase, we run a *feature filter* pass that filters out extraneous python features such as generators, coroutines, lambdas, decorators, etc. In the semantic analysis phase, we have a type checker that is responsible for static type checking as well as type inference to determine the scope and type of newly initialized variables. This is essential for translation to C++ since the latter is not dynamically scoped or typed. In the code generation phase, we extract configuration options from the __init__ function signature to generate the layer-specific Options struct, generate class and function declarations, and finally, the function body. We use the Jinja2 [3] third-party tool to lay out the code skeleton for generated code, simplifying code generation.

Generating the function body is one of the core aspects of the compiler and here we discuss a couple of implementation details. First is the generation of the Linear module. We manually inline the generation of mpi's *Linear* module instead of using torch's *Linear* module directly. This approach not only simplifies code generation by avoiding generating shared_ptr variables but also paves way for future instance variable elision when the input and output dimensions are a direct function of the config. In the SAGEConv example, the input dimension of the initialized *Linear* module is simply twice the input dimension, and the output dimension is unchanged. In such scenarios, it is wasteful to store additional instance variables for the dimensions of the Linear module. Finally, we use C++ templates to translate the update_all message passing function to Marius code. We have a template for each builtin aggregation function, namely sum, mean and max. We invoke the update_all function with the appropriate template at the call site.

Finally, we were able to write the complete compiler code in less than 2k lines of python code. The code is open source as part of the *MariusGNN* framework

5 Evaluation

5.1 Dataset and Machine Configuration

We evaluated the MariusGNN [12] on FB15k_237 dataset [9]. It is a link prediction dataset with 272,115 edges, 14541 edges and 237 relations. We have trained the MariusGNN entirely on CPU instance (with 8 GB RAM) provided by CloudLab [8].

5.2 Experiments

MariusGNN model was trained for 10 epochs. As seen in the Table 1, Marius Script compiler reduces the complexity of *graph_sage_layer* implementation

	Marius(without Script Compiler)	**Marius(with Script Compiler)**
Code Length (*graph_sage_layer)*	114	23

Table 1: Code Length comparison

	Marius(without Script Compiler)	**Marius(with Script Compiler)**
MRR	0.26	2601
Hits@1	0.174	0.174
Hits@5	0.348	0.349
Hits@100	0.706	0.705
Execution time per epoch(ms)	3231.9	3254.9

Table 2: MRR, Hits@K and Execution Time Comparison

- with 79.8% reduction in the number of lines of code.

Also, to confirm the correctness of the Marius Script Compiler, we tested the MariusGNN on MRR (Mean Reciprocal Rank) and Hits@N (N = 1, 5, 100) metrics. We found out that there is no decrease in both of these metrics - either with or without using our script compiler. Further, we also calculated the execution time of average training epoch. It was observed that there is an insignificant difference (∼23 ms) when using Marius Script Compiler. This means our script compiler integration in MariusGNN decreases the code complexity without compromising the execution time. The results are displayed in Table 2. The simplicity of the *graph_sage_code* can be seen in the code listing 2.

6 Future Work

The scope of any compiler is ever-expanding, and a major reason for that is the endless opportunities for optimizations. In this section, we lay out potential future opportunities for the project.

6.1 User defined functions - GAT layer

Graph Attention Network (GAT for short) [11] is a popular Graph Neural Network architecture that introduces the self-attention [10] mechanism in learning graph-structured data. More importantly, GAT differs from Graph Sage in that the *message* and *reduce* functions are *user-defined.* Therefore, we need to support user-defined functions in `mpic` to compile the GAT layer.

6.2 Dataflow optimizations - Graphiler

While popular frameworks such as DGL and Pytorch Geometric[2] support user-defined functions, they are not very optimized. In particular, users are prone to writing unoptimized message-passing functions, making the execution significantly slower. In response, Graphiler[14] proposes numerous dataflow-based optimizations to tackle this problem. Similarly, the `mpic` tool can in turn adopt these optimizations.

6.3 Just-in-time compilation - DGL integration

Currently, users run *MariusGNN* in multiple phases if they want to use the Message Passing Interface. First, they need to generate cC++ files from `Marius Script` code. Then, they need to integrate these files into *MariusGNN* in order to run their layer. This approach slows down development. Instead, can users use *MariusGNN* as a DGL backend? Such support will increase code reusability and improves developer productivity significantly. However, achieving such compatibility requires just-in-compilation making this an interesting future project.

7 Conclusion

As part of this project, we

- Designed a minimal viable interface in `Marius Script`
- Developed a compiler tool to improve the usability of the *MariusGNN* framework

- Compared performance and accuracy of generated code against vanilla Graph Sage implementation of the framework to verify compiler correctness

References

[1] ast — abstract syntax trees. `https://docs.python.org/3/library/ast.html`.

[2] Pyg documentation. `https://pytorch-geometric.readthedocs.io/en/latest/`.

[3] Template designer documentation. `https://jinja.palletsprojects.com/en/3.1.x/templates/`.

[4] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry, 2017.

[5] W. L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159.

[6] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs, 2017.

[7] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks, 2016.

[8] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), Dec. 2014.

[9] K. Toutanova and D. Chen. FB15K-237 Dataset. `https://msropendata.com/datasets/2b11f0e1-a9b2-4983-8c60-a9eb92950c79`, 2015.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2017.

[11] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks, 2017.

[12] R. Waleffe, J. Mohoney, T. Rekatsinas, and S. Venkataraman. Mariusgnn: Resource-efficient out-of-core training of graph neural networks. In *Eighteenth European Conference on Computer Systems (EuroSys' 23)*, 2023.

[13] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks, 2019.

[14] Z. Xie, M. Wang, Z. Ye, Z. Zhang, and R. Fan. Graphiler: Optimizing graph neural networks with message passing data flow graph. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 515–528, 2022.