上海科技大学
ShanghaiTech University

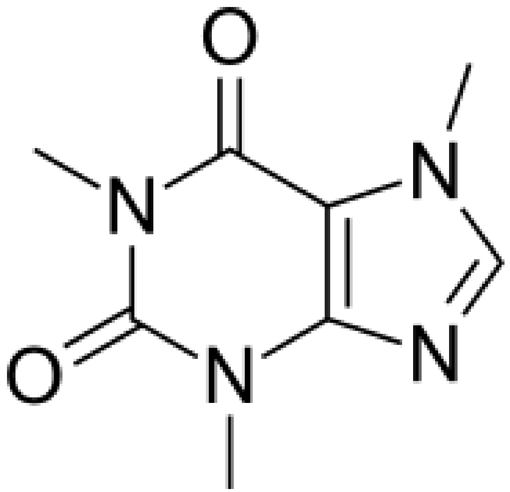
aws

Graphiler: Optimizing Graph Neural Networks with Message Passing Data Flow Graph

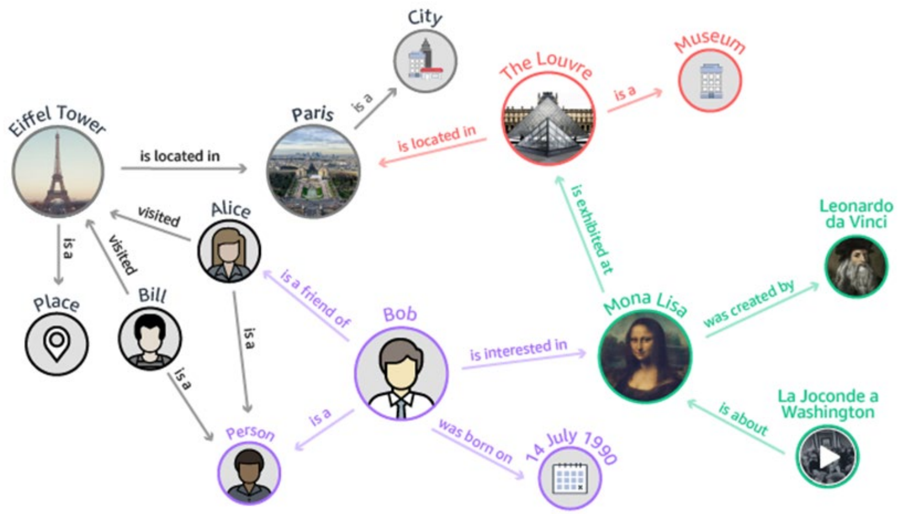
Zhiqiang Xie†*
Joint work with
Minjie Wang*, Zihao Ye*, Zheng Zhang* and Rui Fan†
† ShanghaiTech University * AWS

Graph and Graph Neural Networks

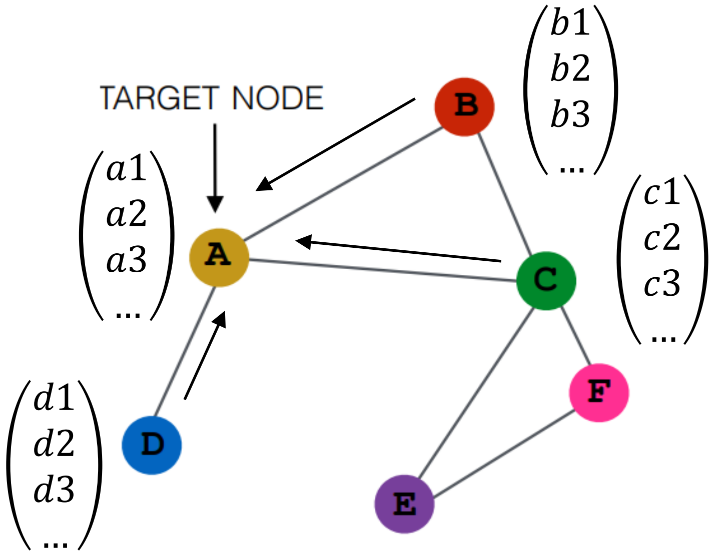
Social Network



Molecular



Knowledge Graph

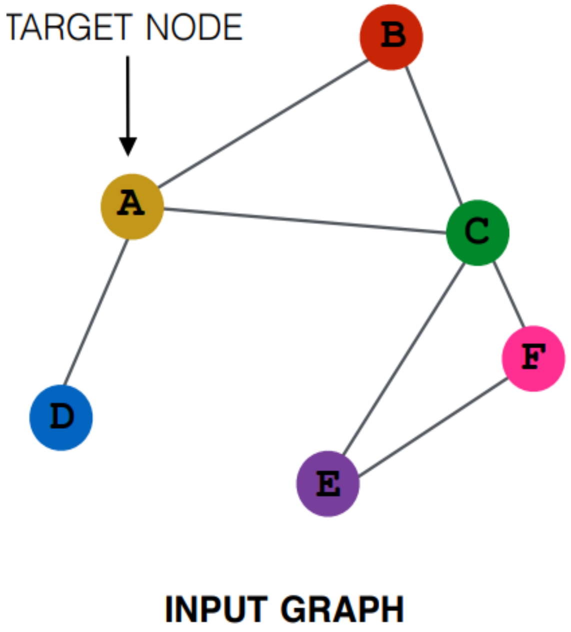


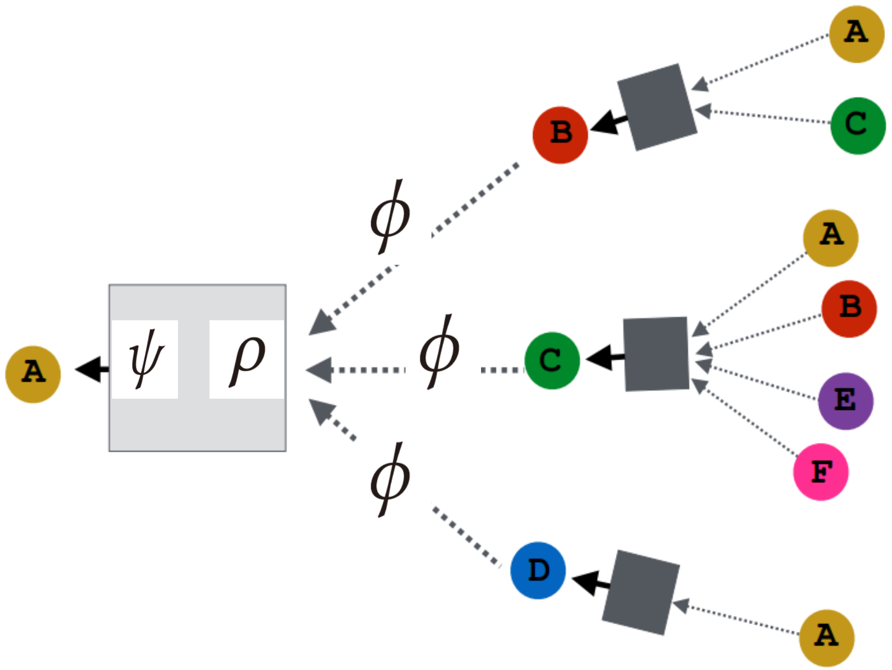
Graphs are **ubiquitous** in real world!

Graph neural networks **combine** graph data structures and neural networks

Image source: AWS Partner Network Blog, Stanford CS224W notes

Message Passing Paradigm





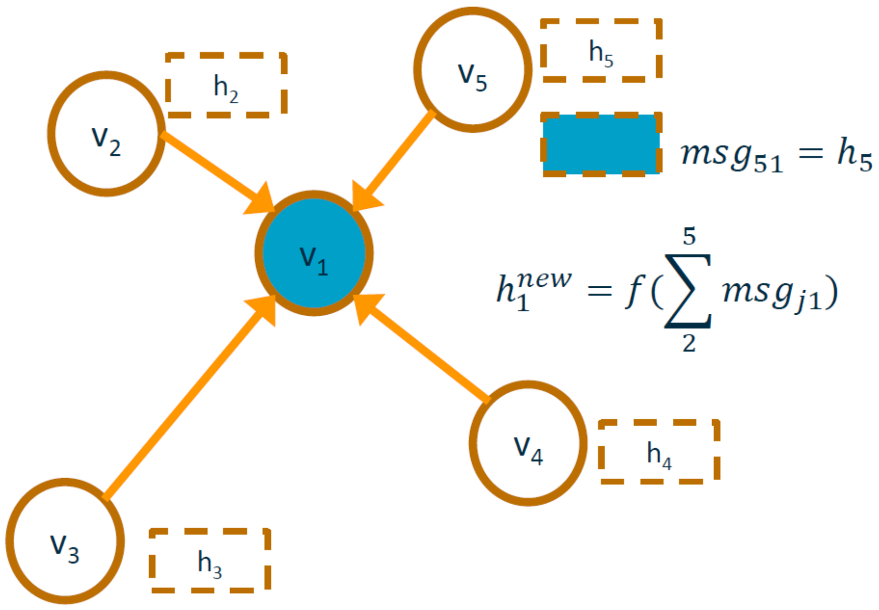
Message passing in three stages

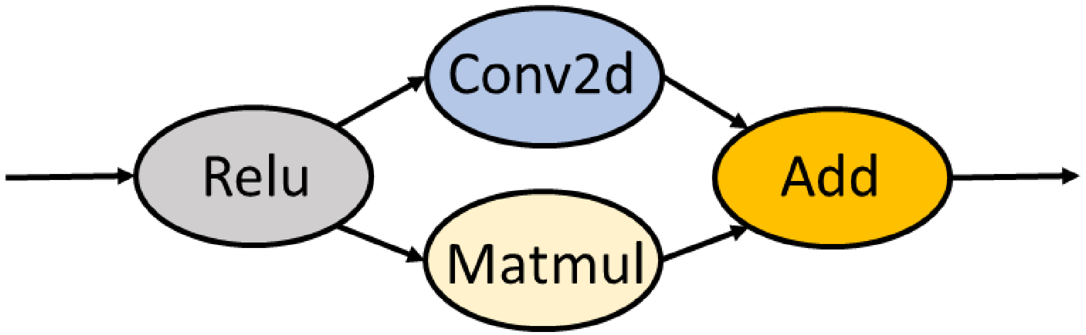
Message creation: $m_e = \phi\left(x_u, x_v, w_e\right), (u, e, v) \in \mathcal{E},$

Message aggregation: $h_v = \rho\left(\{m_e : (u, e, v) \in \mathcal{E}\}\right),$

Feature update: $x_v^{new} = \psi\left(x_v, h_v\right), v \in \mathcal{V}.$

Gap between GNNs and DNN frameworks





Message passing paradigm defines **fine-grained** graph computation

- **Edge-wise:** how to send message
- **Node-wise:** how to use message

DL frameworks provides **coarse-grained** tensor computation

- **Operators**: how to transform tensors

User-Define Function (UDF)

Simplified GCN:

$$m_e = x_u W, (u,e,v) \in \mathcal{E}$$

$$h_v = \sum_{(u,e,v)\in\mathcal{E}} (m_e)$$

$$x_v^{new} = \sigma(h_v)$$

Implementation in DGL-UDF

```
def message_func(edges):
    return {'m': torch.mm(edges.src['h'], Weight)}

def reduce_func(nodes):
    return {'h': torch.sum(nodes.mailbox['m'], dim=1)}

def update_func(nodes):
    return {'x': torch.relu(nodes.data['h'])}
```

- **Intuitive** and straightforward translation from math formula to code
- **Less efficient** owing to implicit conversion from irregular graph computation to fixed-shape dense tensor computation by duplicating, sharding, etc.
- Good for fast prototyping

Specialized Primitives

DGL-Primitives of GAT:

```
def gat_dgl_primitives(graph, h):
    # equation (1)
    z_src = z_dst = h @ W
    # equation (2)
    el = z_src @ W_att_l
    er = z_dst @ W_att_r
    graph.srcdata.update({'m': z_src, 'el': el})
    graph.dstdata.update({'er': er})
    graph.apply_edges(dgl.u_add_v('el', 'er', 'e'))
    e = leaky_relu(graph.edata.pop('e'))
    # equation (6)
    e_max = dgl.copy_e_max(graph, e)
    e = exp(dgl.e_sub_v(graph, e, e_max))
    e_sum = dgl.copy_e_sum(graph, e)
    graph.edata['alpha'] = dgl.e_div_v(graph, e, e_sum)
    # equation (7)
    graph.update_all(dgl.u_mul_e('m', 'alpha', 'm'),
                     dgl.sum('m', 'r'))
    return graph.dstdata['r']
```

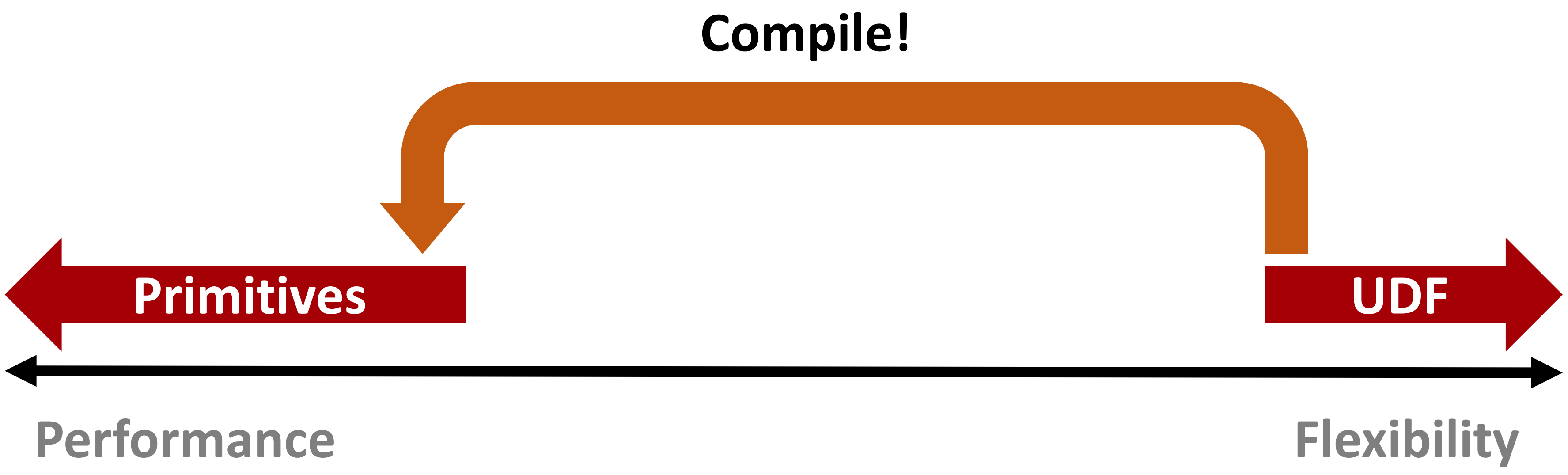
DGL-UDF of GAT:

```
def message_gat(edges):
    # equation (1)
    z_src, z_dst = edges.src['h'] @ W, edges.dst['h'] @ W
    # equation (2)
    e = leaky_relu(concat(z_src, z_dst, dim=1) @ W_att)
    return {'m': z_src, 'e': e}

def aggregate_func(nodes):
    # equation (6)
    alpha = softmax(nodes.mailbox['e'], dim=1)
    # equation (7)
    r = sum(alpha * nodes.mailbox['m'], dim=1)
    return {'r': r}
```

- Efficient sparse computation primitives, **10x – 100x faster** than UDFs!
- Hard to use!
- Suitable for performance critical scenarios

Goal: Bridge the Gap!



Key idea:

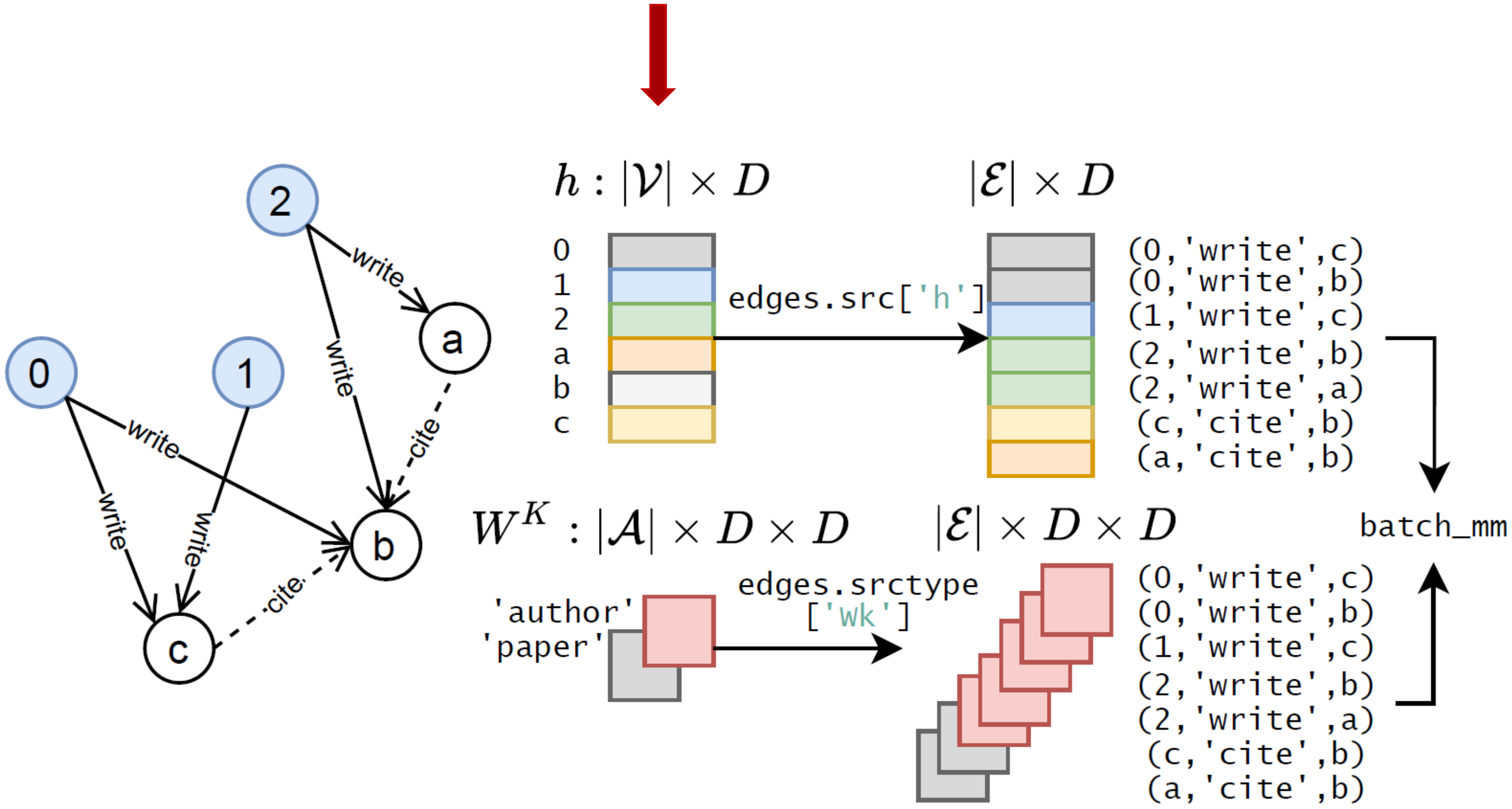
Build a compiler stack to bridge the gap between UDFs and primitives

Can existing DNN compilers help?

What Makes UDF Slow: Redundancy

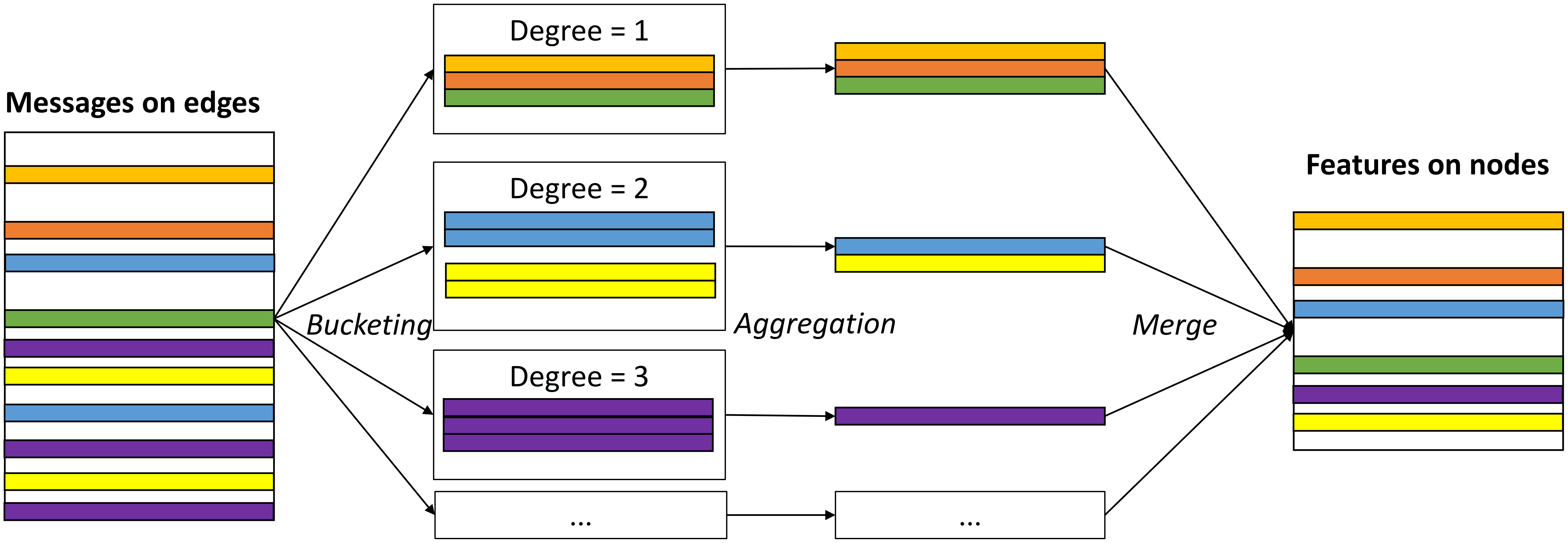
From Heterogeneous Graph Transformer (HGT):

```
k = batch_mm(edges.src['h'], edges.srctype['Wk'])
```



- High memory consumption
  - Excessive intermediate data materialization
- Redundant computation and memory access

What Makes UDF Slow: Fragmentation



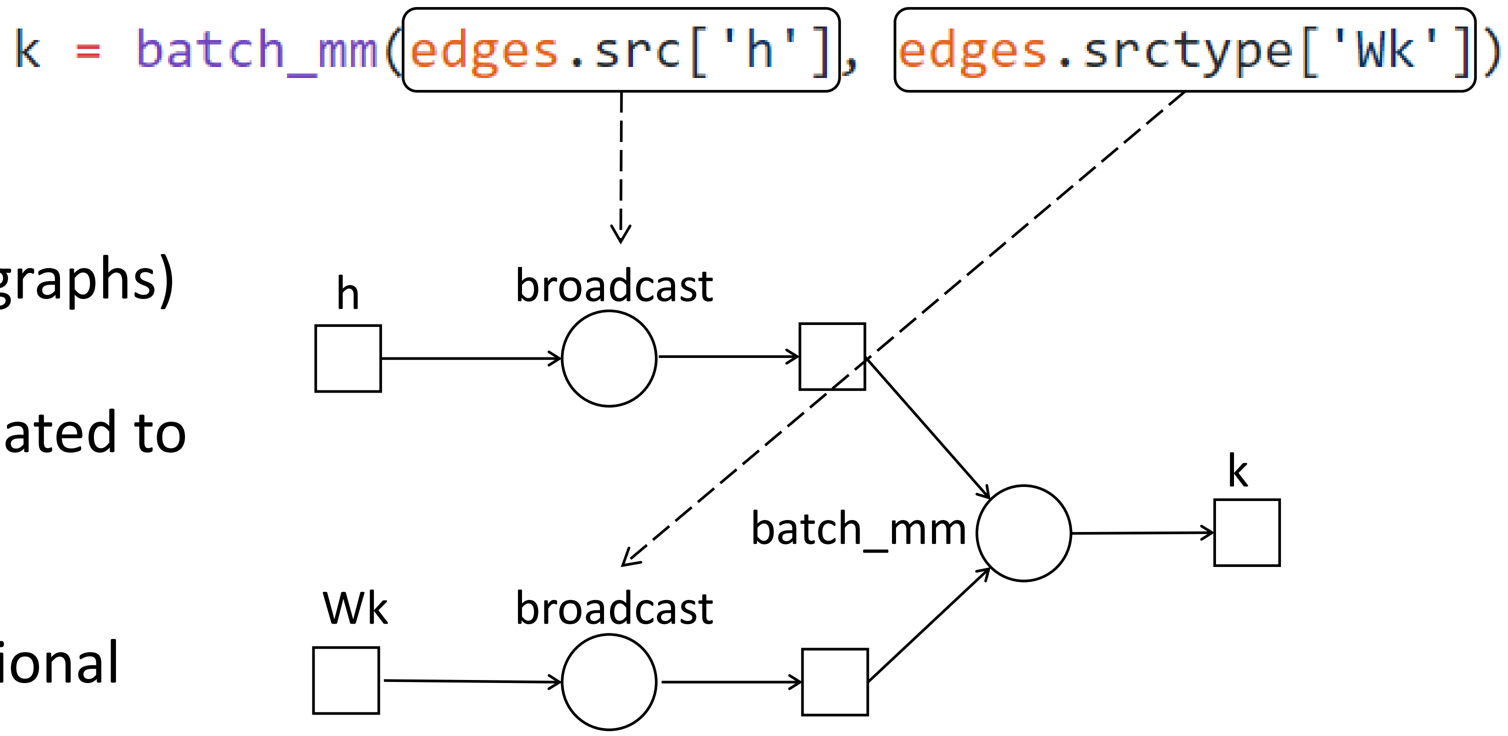
- Significant function call overhead
- Low hardware utilization
- Extra memory traffic

Can Existing DNN Compilers help?

Yes! Many components are reusable
(e.g., parsing programs to build data flow graphs)

No! Message passing operations are translated to opaque operators

- Infeasible to describe certain computational patterns
- Prohibit further specific optimizations



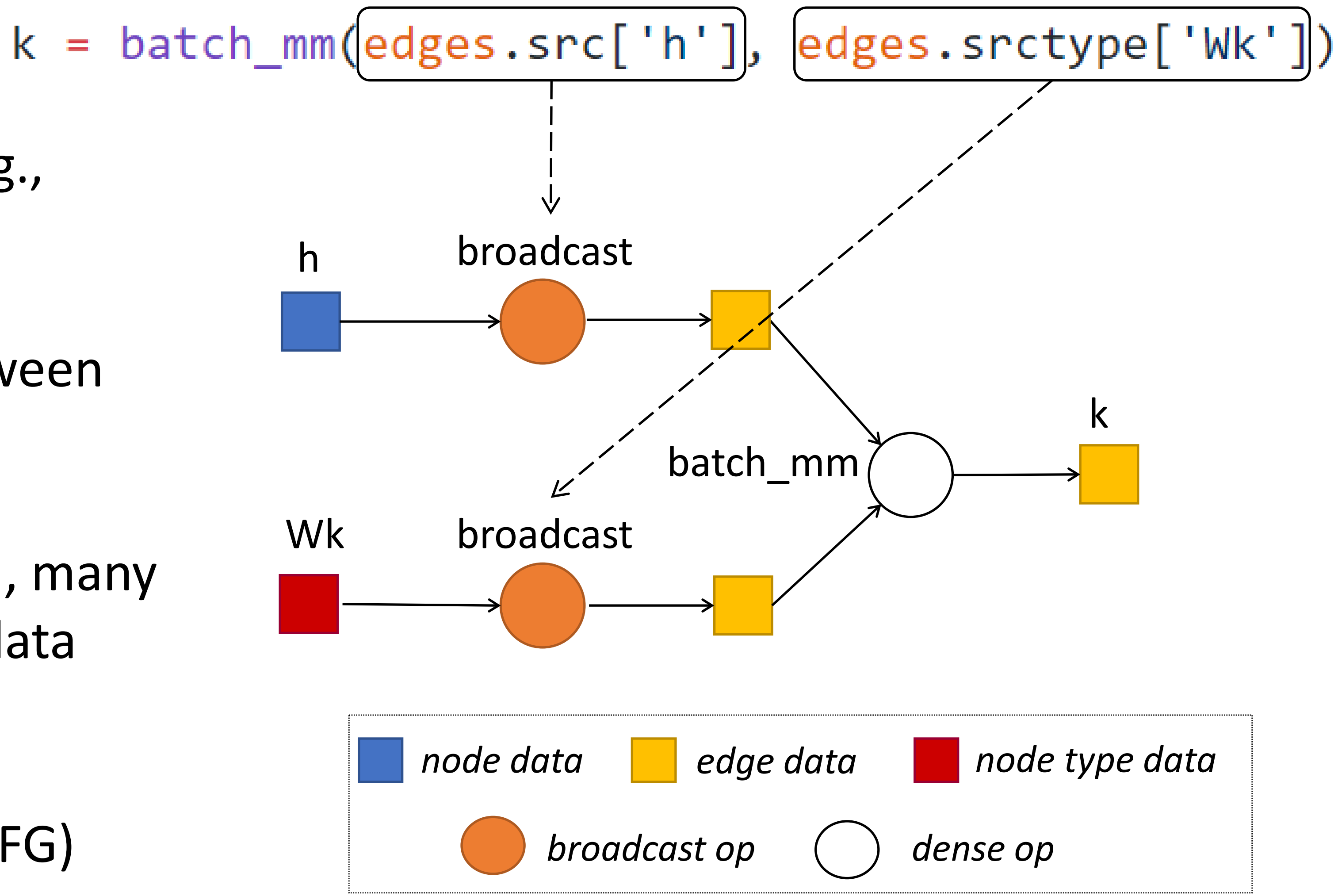
Observation: Data Residency and Movement

Data **residency**: where do data reside (e.g., nodes, edges, types)?

Data **movement**: how do data move between entities in a graph?

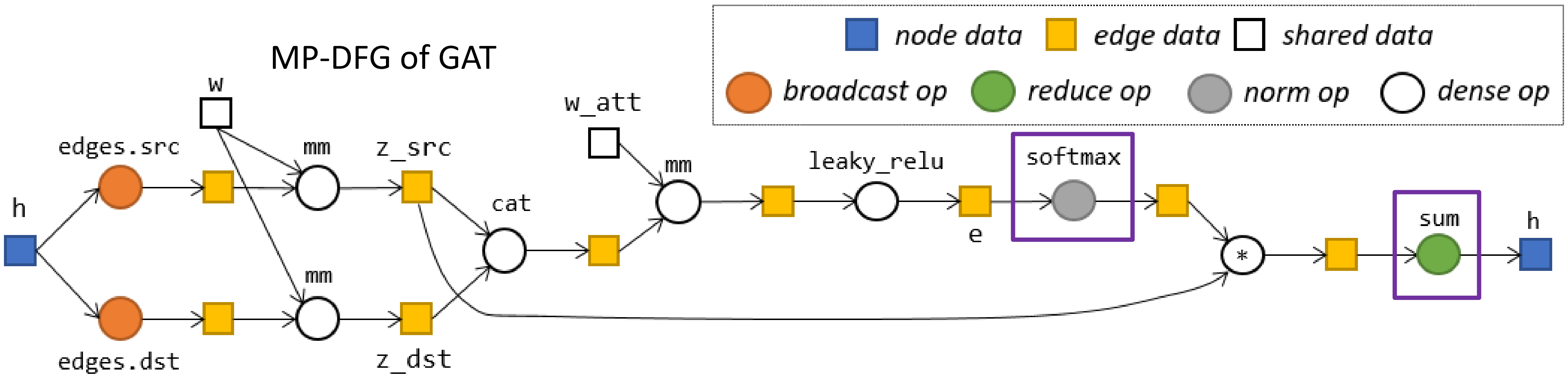
Insight: In the message passing paradigm, many computation-heavy patterns are tied to data residency changes

Message Passing Data Flow Graph (MP-DFG)



MP-DFG Builder

MP-DFG of GAT




```python
def message_gat(edges):
    # equation (1)
    z_src, z_dst = edges.src['h'] @ W, edges.dst['h'] @ W
    # equation (2)
    e = leaky_relu(concat(z_src, z_dst, dim=1) @ W_att)
    return {'m': z_src, 'e': e}

def aggregate_func(nodes):
    # equation (6)
    alpha = softmax(nodes.mailbox['e'], dim=1)
    # equation (7)
    r = sum(alpha * nodes.mailbox['m'], dim=1)
    return {'r': r}
```

GAT in DGL-UDF

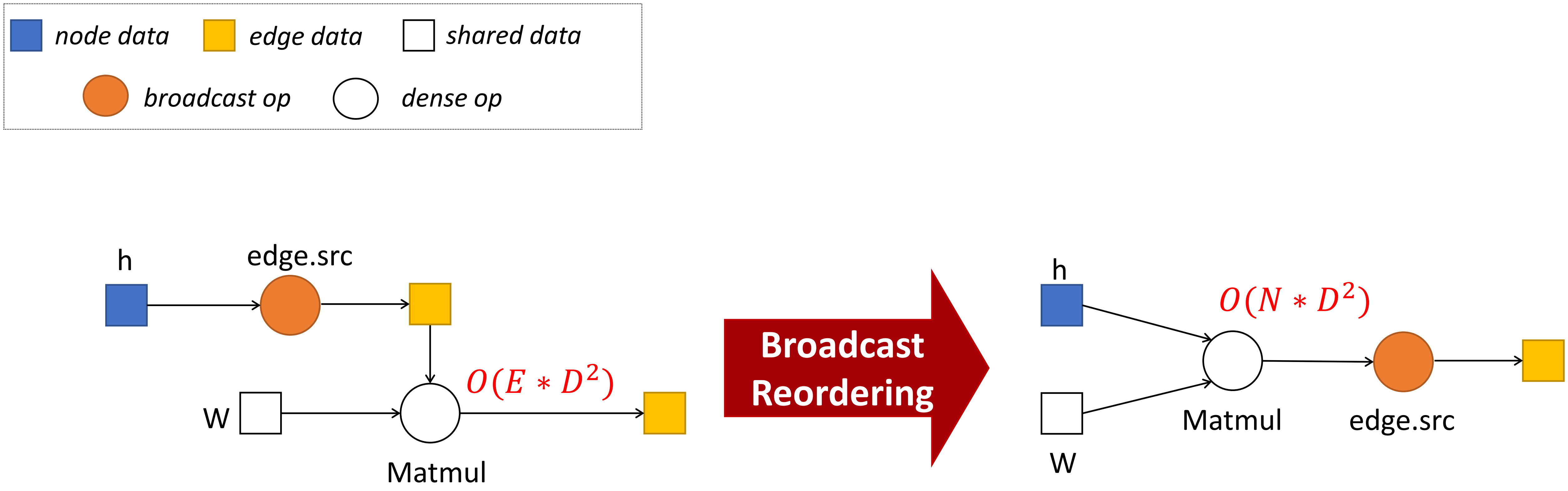
Type Inference & Annotation

- Annotation propagation
- Automatically replace fragmented message aggregation with efficient primitives

…

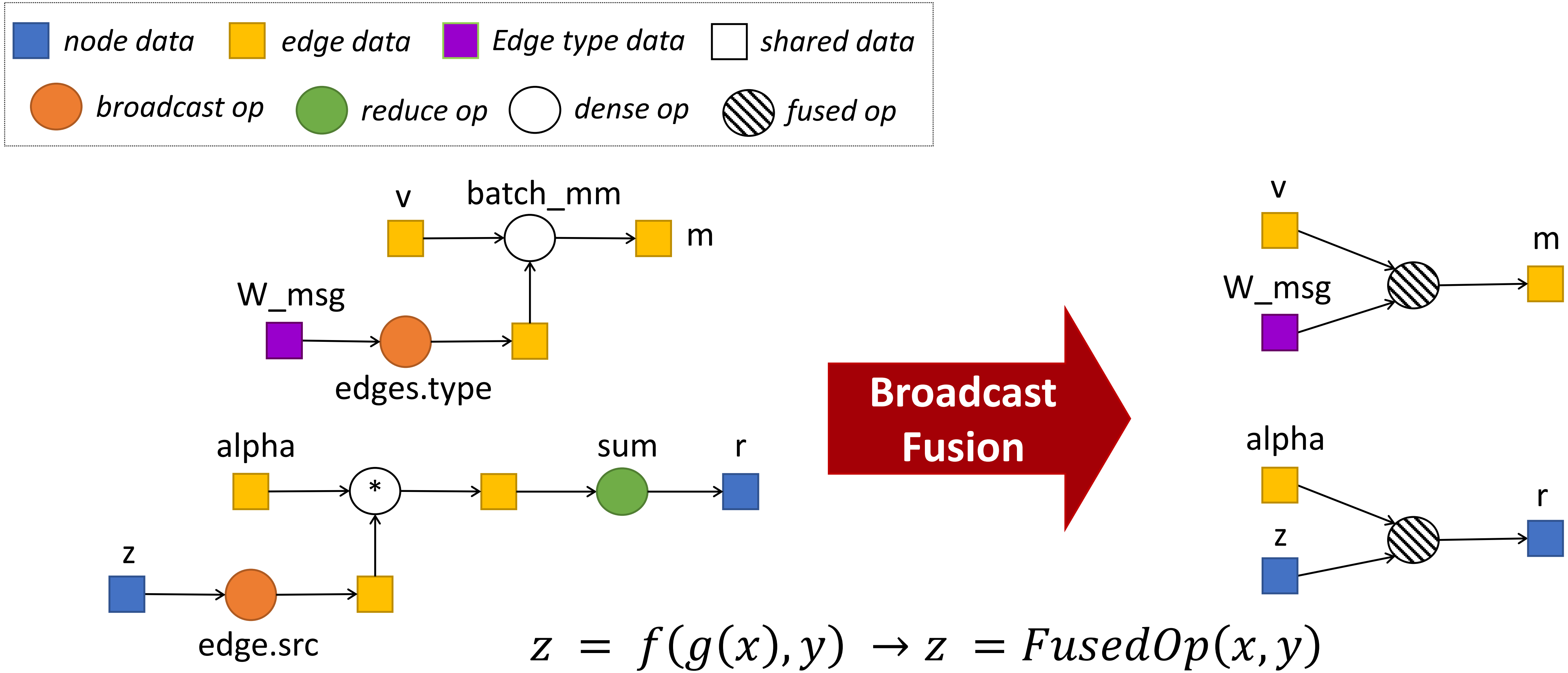
Check out paper for more details!

Optimization enabled: Broadcast Reordering



$$y = f(g(x)) \rightarrow y = g(f(x))$$

Optimization enabled: Broadcast Fusion

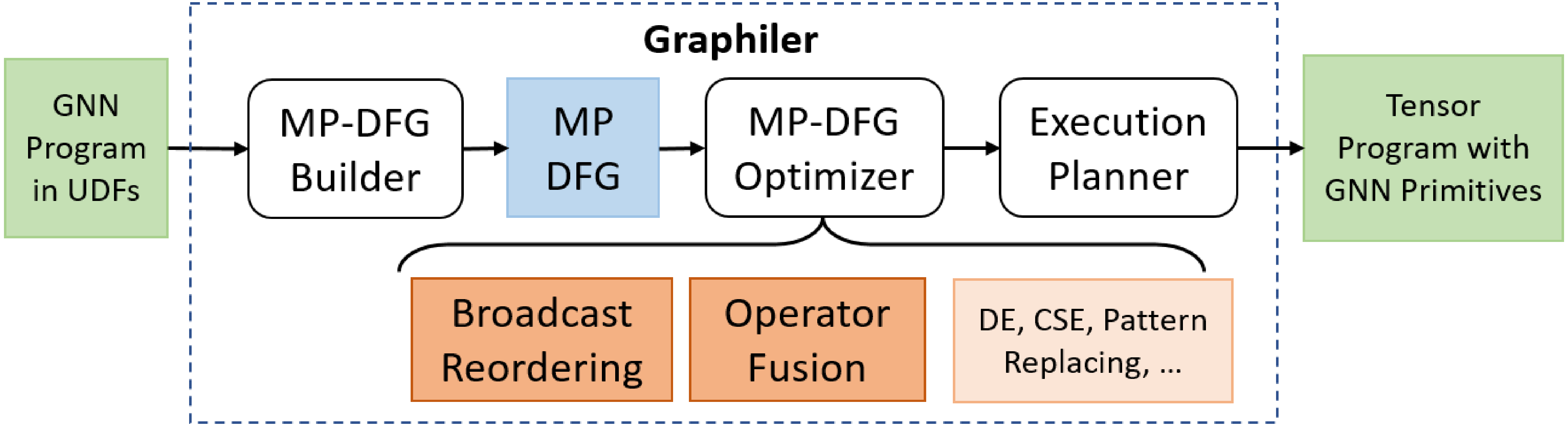


$$z = f(g(x), y) \rightarrow z = FusedOp(x, y)$$

$$z = \rho(f(g(x), y)) \rightarrow z = FusedOp(x, y)$$

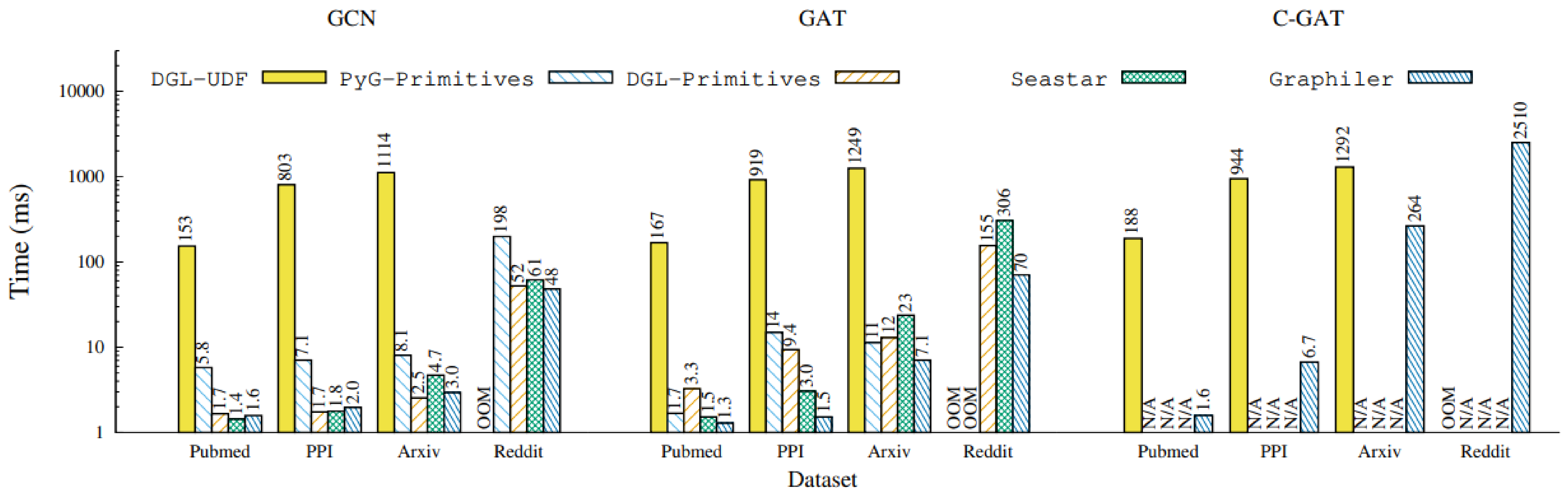
...

Graphiler as a Compiler Stack



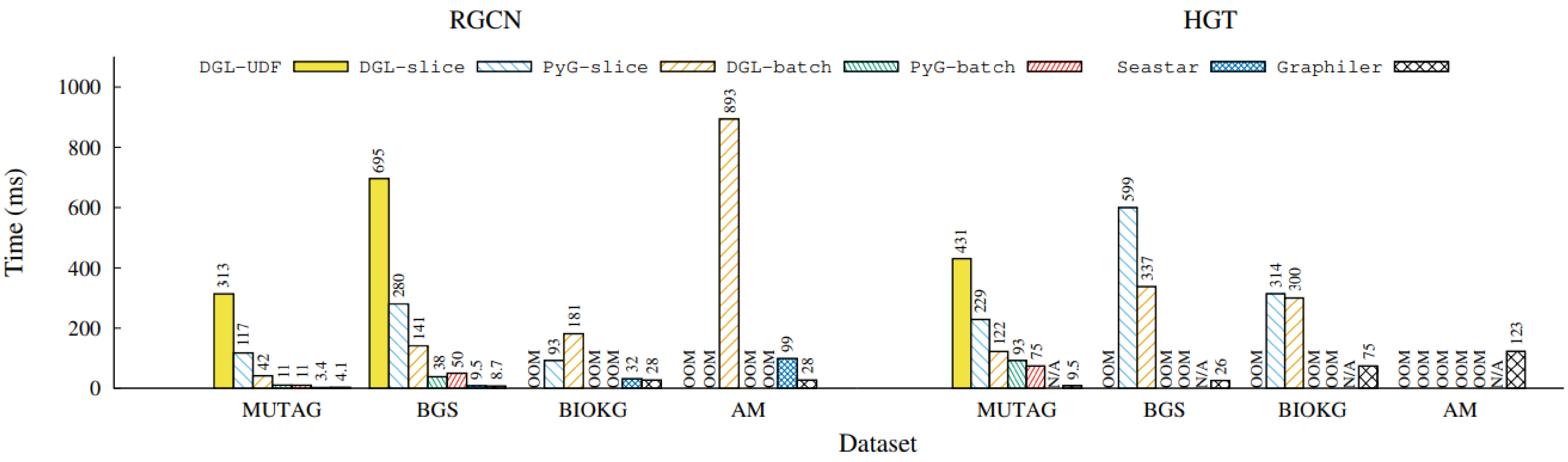
- Check out the paper and code for more details!
  https://github.com/xiezhq-hermann/graphiler

End-to-end Performance of Homogeneous GNNs



- **231× (GCN), 243× (GAT) and 80× (C-GAT)** faster on average over all the datasets compared with DGL-UDF baselines
- Comparable performance and often faster than DGL-primitives, PyG-primitives and Seastar
- Significant memory saving

End-to-end Performance of Heterogenous GNNs



- **78× (DGL-UDF), 21× (DGL-slice), 16× (PyG-slice), 3.6× (DGL-batch) and 4.2× (PyG-batch)** faster on average across all benchmarks for R-GCN.
- Enables models running in large datasets by substantial memory saving

Key Takeaways & Future Work

Programming GNNs faces a performance and flexibility trade-off

Graphiler achieves the best of both worlds using a **compiler** approach

GNNs introduce unique computational patterns

A **tailored abstraction** MP-DFG is needed to enable better performance

It is possible to unify computational abstraction for homogeneous and heterogenous GNNs

How about user interfaces? How about abstraction for kernel generation?

DGL team is integrating Graphiler into its official release.

More GNN compilation projects from AWS are to come!

DEEP GRAPH LIBRARY

Questions or comments?