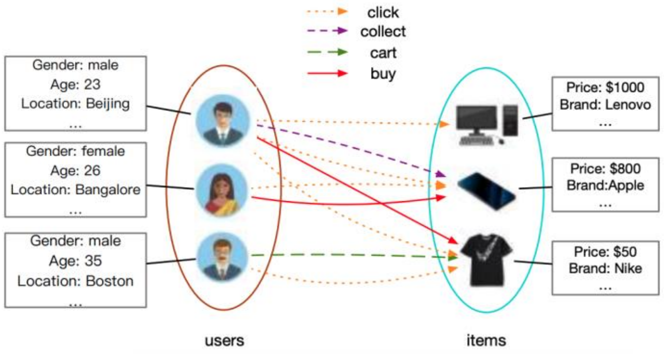
FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems

Yuwei Hu[1], Zihao Ye[2], Minjie Wang[2], Jiali Yu[2], Da Zheng[2], Mu Li[2], Zheng Zhang[2], Zhiru Zhang[1], Yida Wang[2]

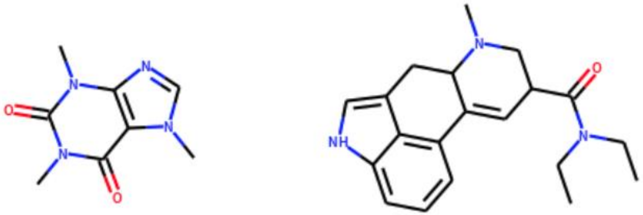
[1] Cornell University

[2] Amazon Web Services

Graph Neural Networks (GNNs) Are Getting Popular



[1]



[2]

Diverse Applications

DeepGraphLibrary [3]

PyTorch geometric [4]

Paddle Graph Learning [5]

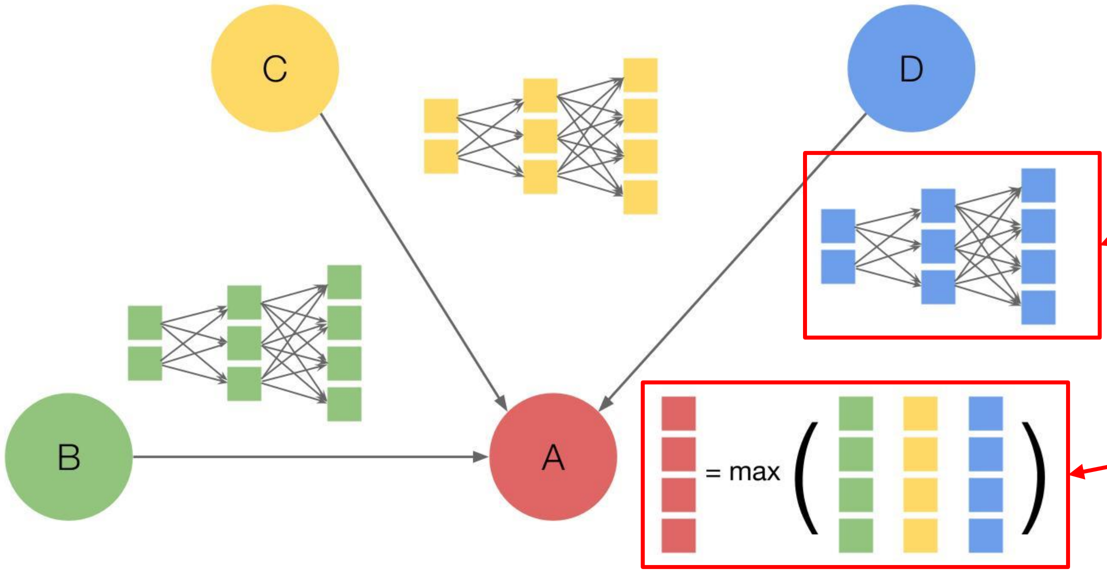
Emerging Systems

1. AliGraph: A Comprehensive Graph Neural Network Platform
2. Interpolate between two molecules with pre-trained JTNN

3. https://www.dgl.ai
4. https://pytorch-geometric.readthedocs.io
5. https://github.com/PaddlePaddle/PGL

Key Building Block of GNNs — Message Aggregation

- **Message function** calculates a message from the feature of each source vertex
- **Aggregation function** aggregates the messages as the new feature of the destination vertex



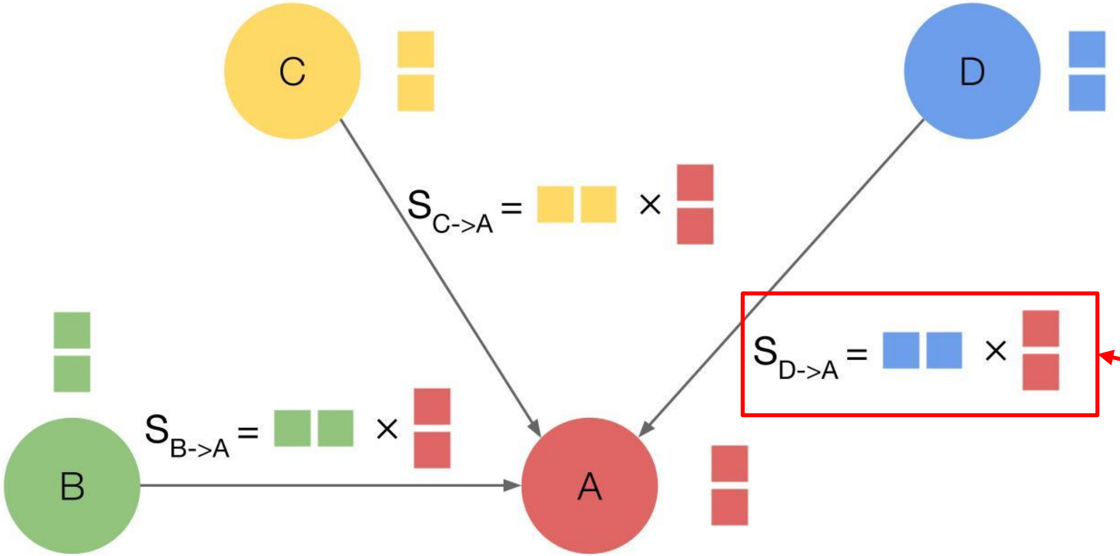
The message function is multi-layer perceptron (MLP)

The aggregation function is taking the max

Message function and aggregation function are customizable

Key Building Block of GNNs — Attention Calculation

- **Edge function** calculates an attention score for each edge



The edge function is dot product

Edge function is customizable

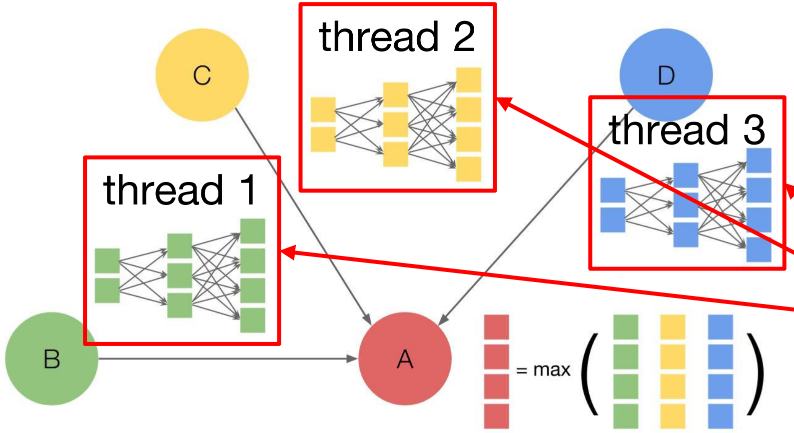
GNN Systems Lack a Flexible and Efficient Backend

Deep learning frameworks as backend (e.g., PyTorch in PyG):

- Lack of support for computation on graph (highly sparse) ✗

Graph processing frameworks as backend (e.g., Minigun in DGL):

- Can flexibly express computation on graph ✔
- Missing optimizations in the feature dimension ✗

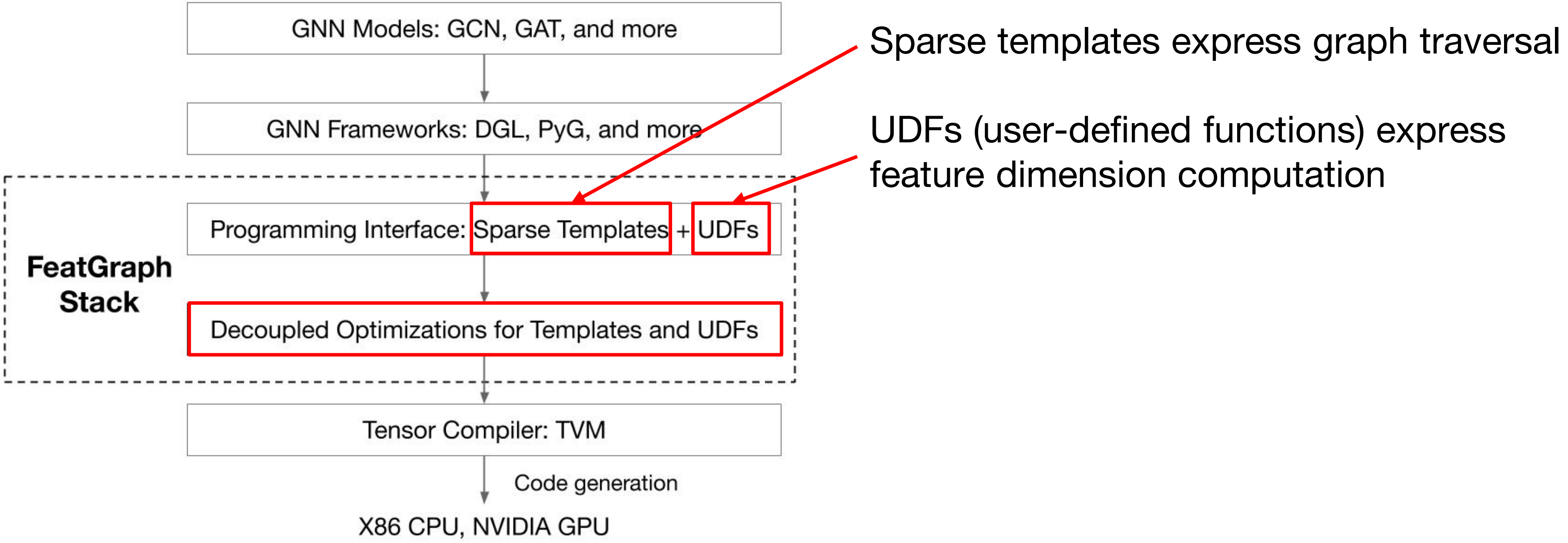


Minigun exploits edge parallelism by scheduling the computation on one edge to one thread

We want to exploit parallelism in the feature dimension as well

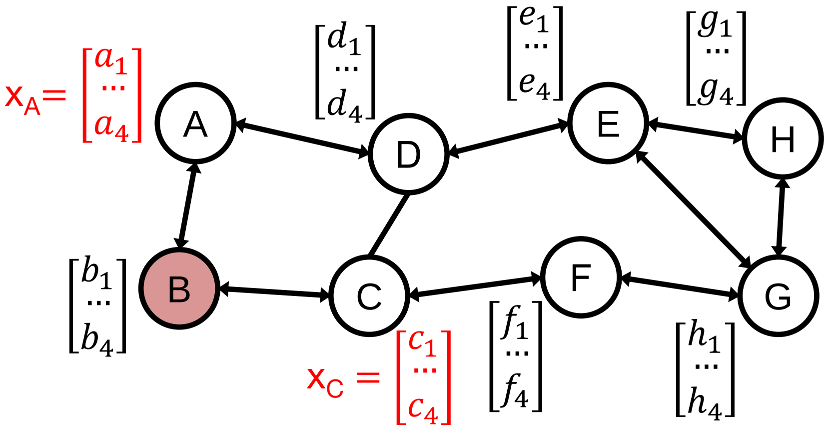
Our Solution: FeatGraph

- FeatGraph co-optimizes graph traversal and feature dimension computation
- FeatGraph accelerates GNN training and inference by 32× on CPU, 7× on GPU



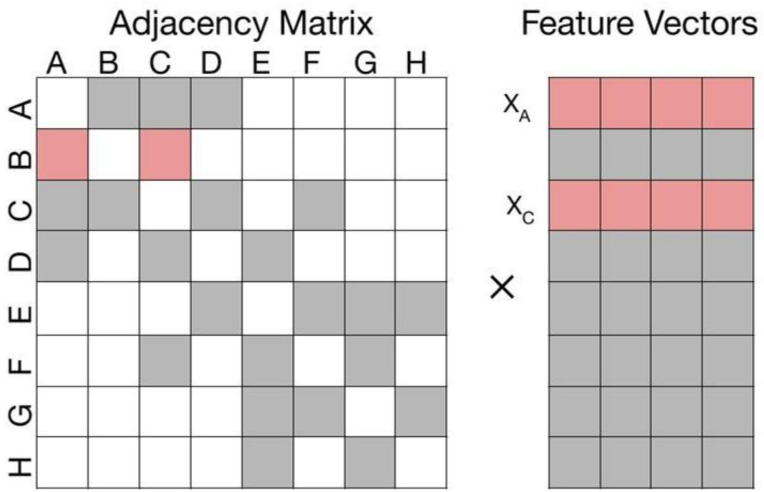
Mapping Graph Computations to Sparse Kernels

Graph View



X_B^{new} = sum(copy(X_A), copy(X_C))

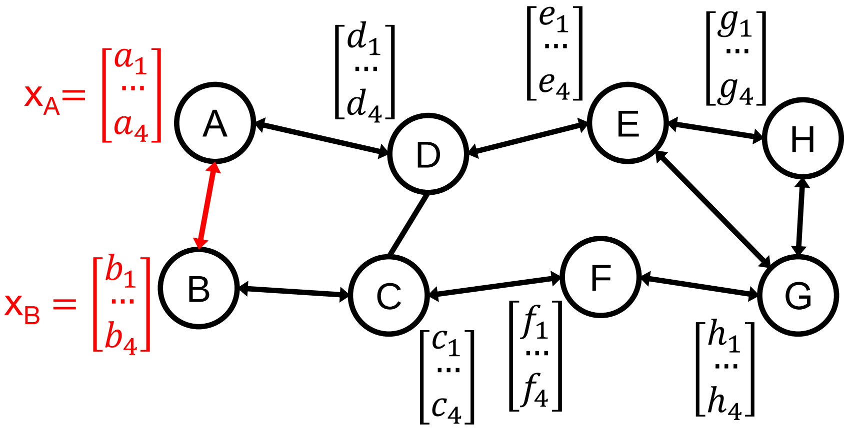
Matrix View



Message aggregation is mapped to generalized SpMM
(sparse-dense matrix multiplication)

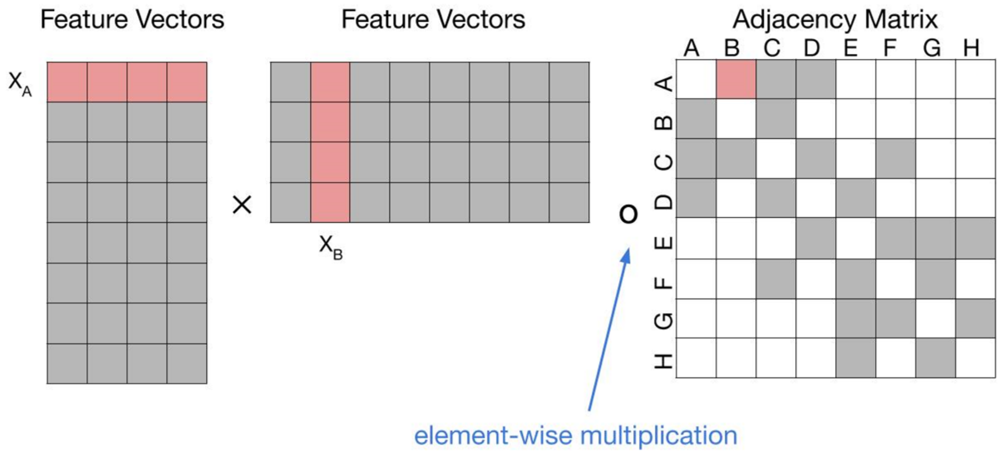
Mapping Graph Computations to Sparse Kernels

Graph View



$S_{AB} = S_{BA} = \mathrm{dot}(X_A, X_B)$

Matrix View



Attention calculation is mapped to generalized SDDMM (sampled dense-dense matrix multiplication)

Programming Interface

```
featgraph.spmm(Adj, MessageF, AggregationF, target, FDS)
```

adjacency matrix of the graph

user-defined message function

user-defined aggregation function

CPU or GPU

feature dimension schedule

```
featgraph.sddmm(Adj, EdgeF, target, FDS)
```

adjacency matrix of the graph

user-defined edge function

CPU or GPU

feature dimension schedule

Expressing GCN[1] Message Aggregation

```
import featgraph, tvm
Adj = featgraph.spmat(shape=(n, n), nnz=m)
VertexFeat = tvm.placeholder(shape=(n, d))
```

```
def MessageF(src, dst, eid):
    out = tvm.compute(shape=(d,),
        lambda i: VertexFeat[src, i])
    return out
```

The message function copies the feature vector of the source vertex

```
AggregationF = tvm.sum
```

The aggregation function is sum

```
Result = featgraph.spmm(Adj, MessageF, AggregationF)
```

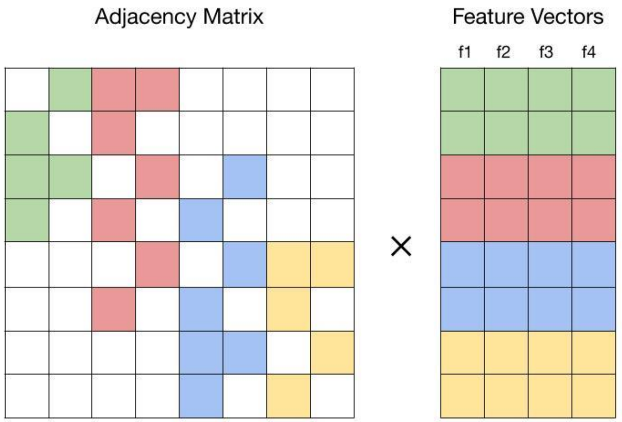
Trigger the SpMM template

[1] T. N. Kipf and M. Welling. “Semi-supervised classification with graph convolutional networks.” ICLR 2017

Optimizing GCN Message Aggregation on CPUs

- Graph partitioning to improve cache utilization

Assume cache capacity is $2 \times L$, L is feature length ⟶ 4 source vertex partitions



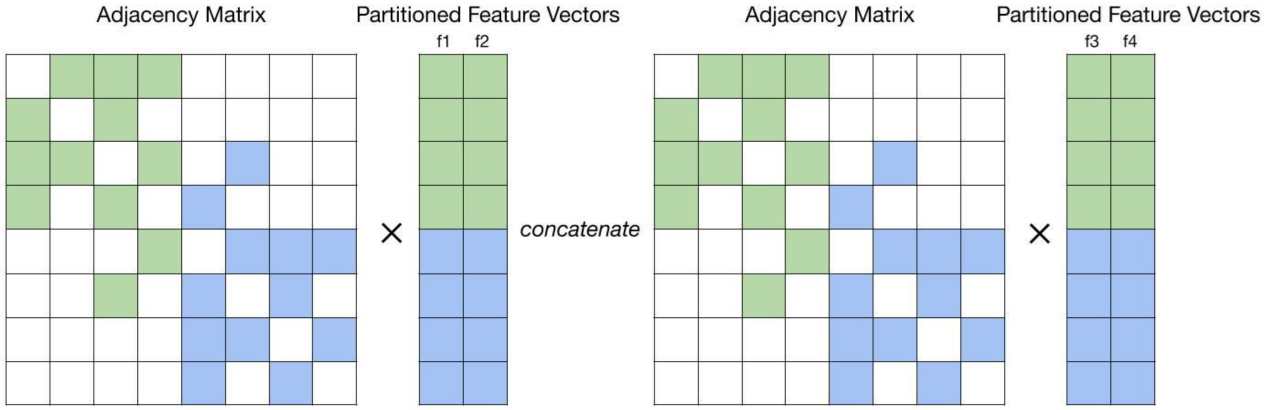
Improved read locality within each partition

Need to merge intermediate results from 4 partitions

Optimizing GCN Message Aggregation on CPUs

- Combining graph partitioning with feature tiling

2 source vertex partitions, 2 feature partitions



Lower merge/write cost

Need to traverse the adjacency matrix twice

Feature tiling enables the tradeoff between accesses to graph topological data and accesses to feature data

Applying CPU Optimizations in FeatGraph

Decoupled, two-level optimizations:

- Incorporating graph partitioning into the sparse templates
- Specifying feature tiling with an FDS (feature dimension schedule)

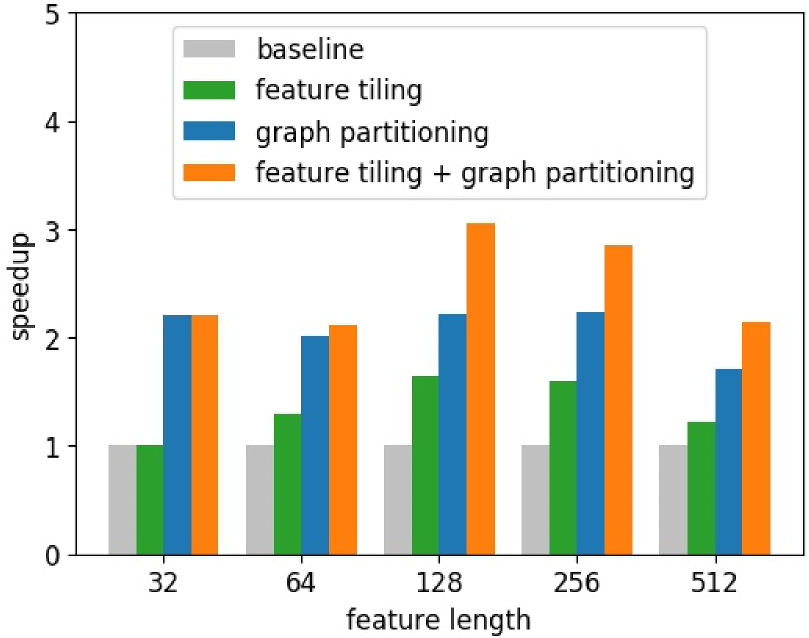
```
def FDS(out):
    s = tvm.create_schedule(out)
    s[out].split(out.axis[0], factor=8)
    return s
```

```
Result = featgraph.spmm(Adj, MessageF, AggregationF, ‘cpu’, FDS)
```

More complex UDFs that compute on multi-dimensional feature tensors require a multi-level tiling scheme, which can also be expressed by an FDS

Effect of Graph Partitioning and Feature Tiling

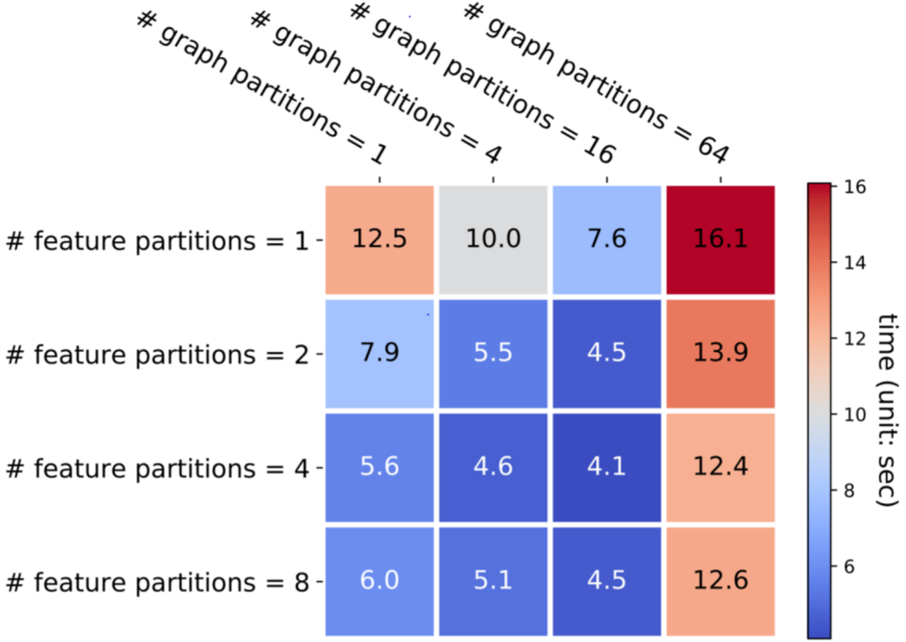
GCN message aggregation, *reddit* dataset:



- Combining graph partitioning and feature tiling effectively boosts the performance

Sensitivity to Partitioning Factors

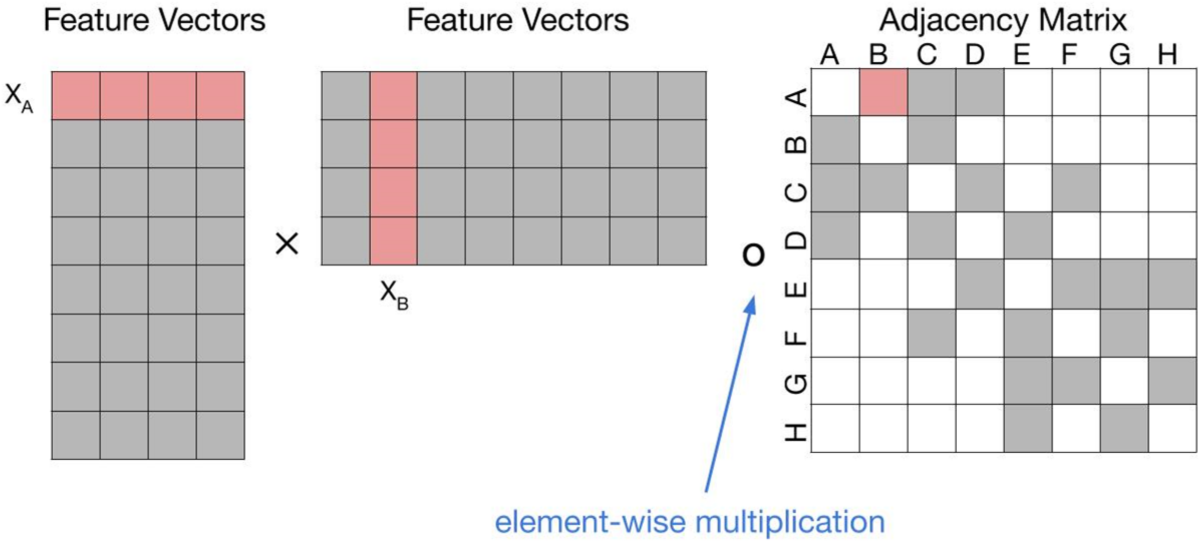
GCN message aggregation, *reddit* dataset, feature length 128:

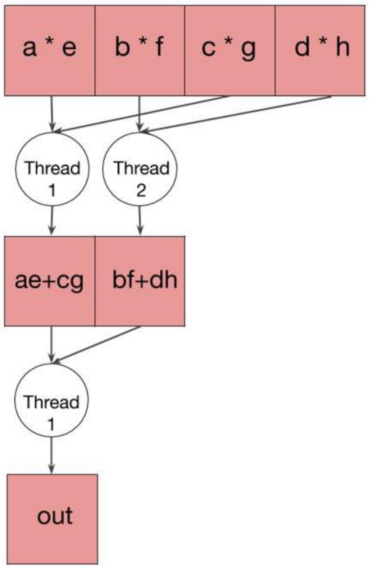


- The best performance is achieved with 16 graph partitions and 4 feature partitions
- FeatGraph uses naive grid search; using intelligent tuners is left for future work

Optimizing Dot-Product Attention on GPUs

- Effective parallelization is the key to achieving high performance on GPU
- FeatGraph exploits parallelism in the feature dimension
  - Threads collectively process one edge using tree reduction
  - In comparison, Gunrock's parallelization strategy: one thread processes one edge





Applying GPU Optimizations in FeatGraph

Decoupled, two-level optimizations:

- Incorporating vertex/edge parallelization into the sparse templates
- Specifying feature parallelization with an FDS

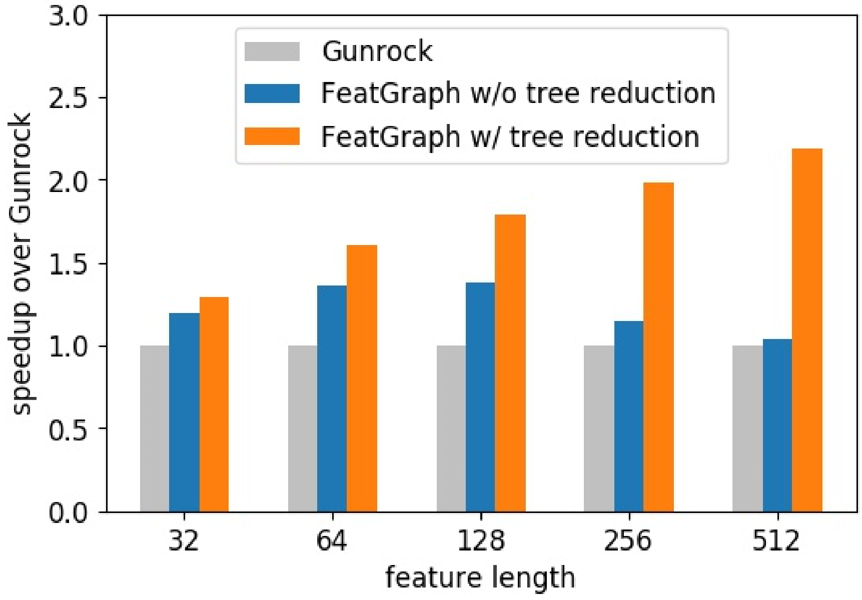
```
def FDS(out):
    s = tvm.create_schedule(out)
    s[out].tree_reduce(out.reduce_axis[0], 'thread.x')
    return s

Result = featgraph.sddmm(Adj, EdgeF, ‘gpu’, FDS)
```

More complex UDFs that compute on multi-dimensional feature tensors require a multi-level parallelization scheme, which can also be expressed by an FDS

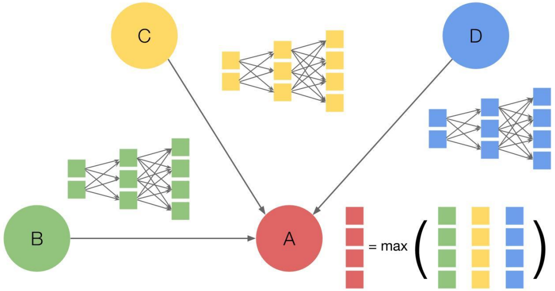
Effect of Feature Dimension Tree Reduction

Dot-product attention, *rand-100K* dataset:



- Tree reduction is especially efficient when the feature length is large

MLP Message Aggregation




```
import featgraph, tvm
Adj = featgraph.spmat(shape=(n, n), nnz=m)
# message function: ReLU((src feature + dst feature) * W)
XV = tvm.placeholder(shape=(n,d1))
W = tvm.placeholder(shape=(d1,d2))
def MessageF(src, dst, eid):
    k = tvm.reduce_axis((0, d1))
    out = tvm.compute((d2,), lambda i:
        tvm.max(tvm.sum((XV[src, k] + XV[dst, k]) * W[k,i ])), 0)
    return out
# aggregation function: max
AggregationF = tvm.max
# CPU FDS: tile multiple dimensions
def FDS(out):
  s = tvm.create_schedule(out)
  s[out].split(out.axis[0], factor=8)
  s[out].split(out.reduce_axis[0], factor=8)
  return s
# GPU FDS: parallelize multiple dimensions
def FDS(out):
  s = tvm.create_schedule(out)
  s[out].bind(out.axis[0], 'block.x')
  s[out].tree_reduce(out.reduce_axis[0], 'thread.x')
  return s
```

Evaluation Setup

Environment

- CPU evaluation is on Amazon c5.9xlarge instance, which is a one socket 18-core 3.0 GHz Intel Xeon Platinum 8124M machine with 25 MB LLC
- GPU evaluation is on Amazon p3.2xlarge instance, which has a Tesla V100

Kernels

- GCN message aggregation (vanilla SpMM)
- MLP message aggregation (generalized SpMM)
- Dot-product attention (vanilla SDDMM)

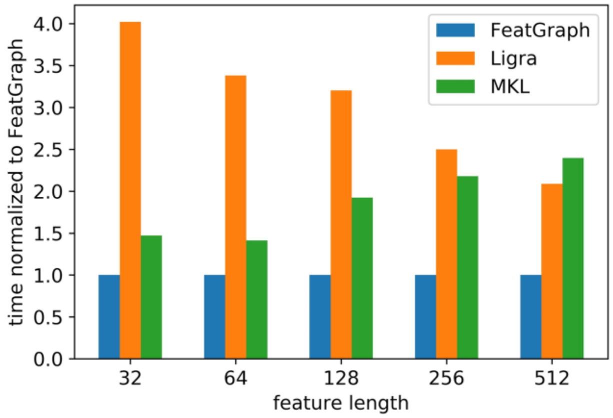
Baselines

- Vendor-provided sparse libraries: MKL on CPU, cuSPARSE on GPU
- Graph processing frameworks: Ligra on CPU, Gunrock on GPU

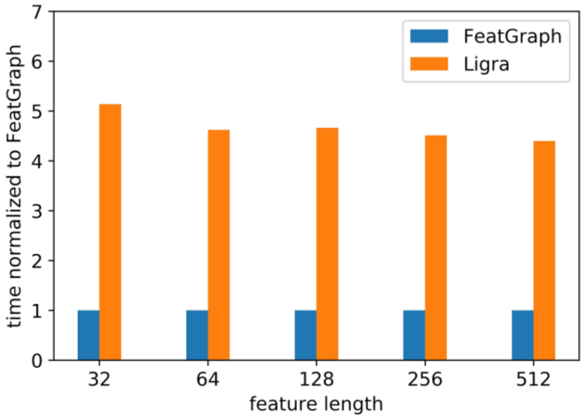
Single-Threaded CPU Kernel Performance

On *reddit* dataset:

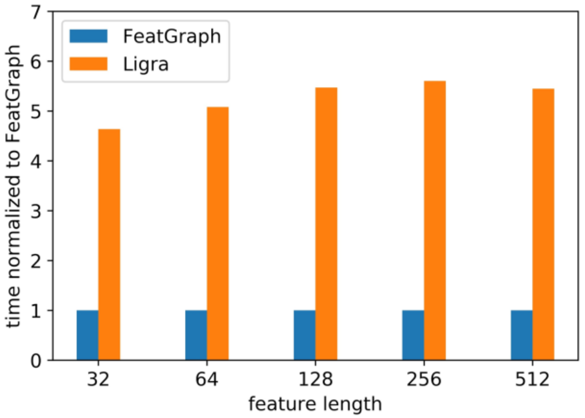
GCN message aggregation



MLP message aggregation



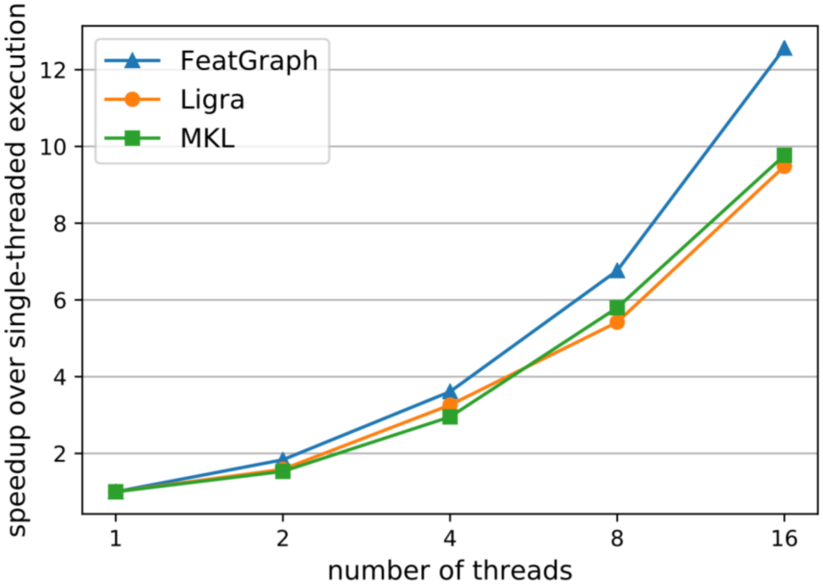
Dot-product attention



- FeatGraph outperforms both Ligra and MKL; MKL does not support MLP message aggregation and dot-product attention
- FeatGraph achieves similar speedup on other tested datasets

Multi-Threaded CPU Kernel Performance

GCN message aggregation kernel, *reddit* dataset, feature length 512:

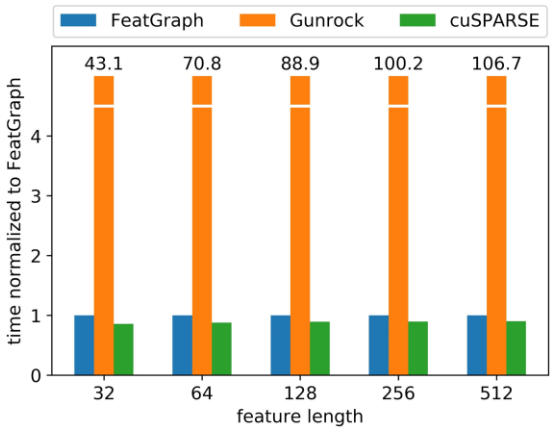


- FeatGraph scales well because of two reasons:
  - Avoiding LLC contention by assigning multiple threads to work on one graph partition at a time
  - The thread pool in TVM runtime is lightweight and efficient

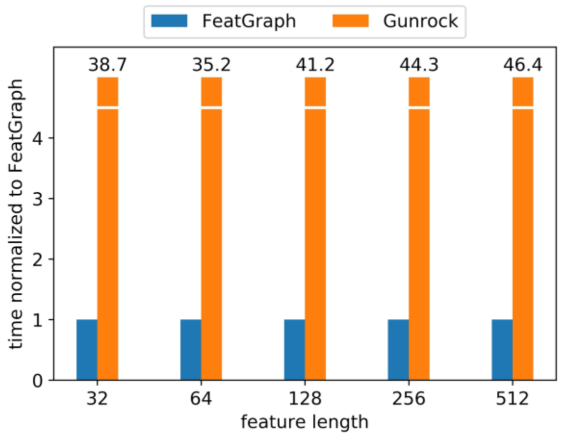
GPU Kernel Performance

On *reddit* dataset:

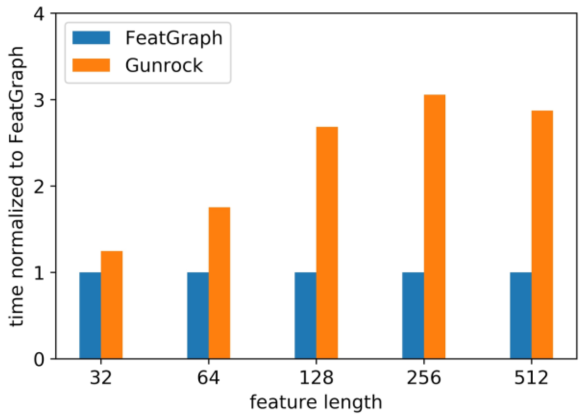
GCN message aggregation



MLP message aggregation



Dot-product attention



- FeatGraph outperforms Gunrock; FeatGraph is on par with cuSPARSE on GCN message aggregation; cuSPARSE does not support the other two kernels
- Gunrock is extremely slow on message aggregation kernels because of two reasons:
  - Its edge parallelization incurs a huge overhead of atomic operations for vertex-wise reductions
  - It does not exploit parallelism in feature dimension computation

End-to-End GNN Training and Inference

We integrated FeatGraph into DGL (version 0.4.1)
The original backend of DGL is Minigun, a “mini-version” of Gunrock

reddit dataset		DGL w/o FeatGraph (unit: sec)	DGL w/ FeatGraph (unit: sec)	Speedup
CPU training	GCN	2447.1	114.5	21.4×
	GraphSage	1269.6	57.8	21.9×
	GAT	5763.9	179.3	32.2×
CPU inference	GCN	1176.9	55.3	21.3×
	GraphSage	602.4	29.8	20.2×
	GAT	1580.9	71.5	22.1×
GPU training	GCN	6.3	2.2	2.9×
	GraphSage	3.1	1.5	2.1×
	GAT	*N/A	1.64	*N/A
GPU inference	GCN	3.1	1.5	2.1×
	GraphSage	1.5	1.1	1.4×
	GAT	8.1	1.1	7.1×

FeatGraph accelerates end-to-end GNN training and inference by up to 32× on CPU and 7× on GPU

FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems

https://github.com/dglai/FeatGraph

Cornell University