Computational Building Blocks for Machine Learning on Graphs

Ariful Azad

Assistant Professor

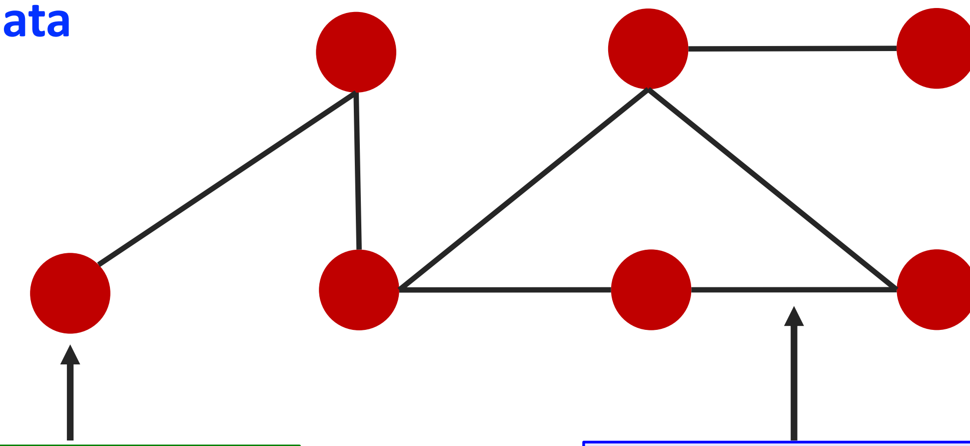
Indiana University, Bloomington

azad@iu.edu

MIT Fast Code Seminar 01/25/2021

What is Graph and why do we care?

Graph: A representation of data



	Vertex (entity)	Edge (relationship)
Biology:	Proteins	Interactions
Social media:	Persons	Friendship
Brain:	Neurons	Synapses

Example: Social networks

What to learn?

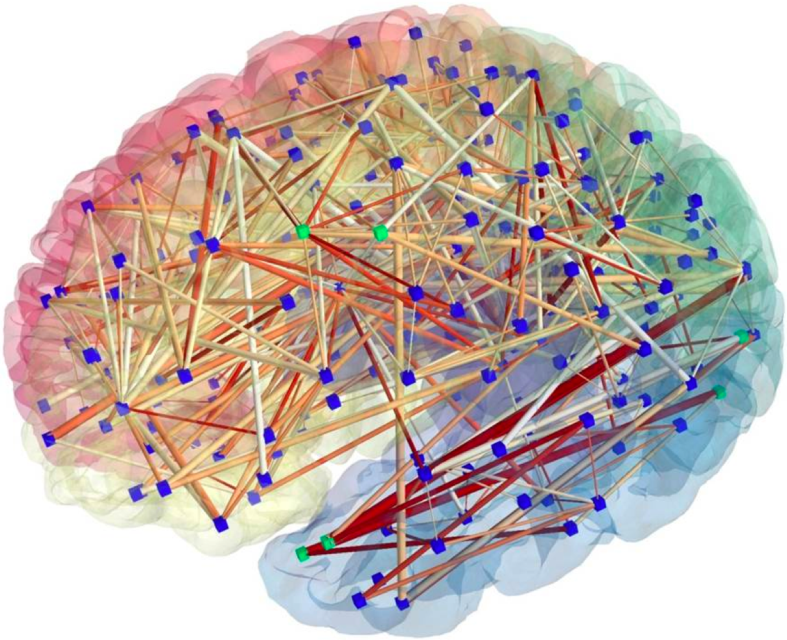
- Information spread
- Recommend friend
- Advertisement

Facebook: > 2.2 billion active users

Brain network

What to learn?

- Brain functions
- Diagnosis and treatments



Brain network: 100B neurons and 100T synapses

Common learning objectives on graphs

- Classify nodes/subgraphs/graphs: protein classification, topic classification
- Predict links: Are x and y friends? Friend suggestions on Facebook
- Identify communities: protein families
- Visualize graphs

This talk discusses common computations needed in all these tasks and how to map them to sparse linear algebra

Outline

1. **Learning on graphs**: All we need is **graph embedding**
   - Shallow embedding/deep embedding

2. **Computational patterns in graph embedding:** All we need is **passing messages among nodes**

3. **Central computations in message passing**: All we need is **sparse-dense (SpMM)** and **sampled dense-dense matrix multiplications (SDDMM)**
   - Example: graph drawing, graph embedding, graph neural networks

4. **Efficient computations**: All we need is **optimizing memory utilization**

5. **Portable implementations:** All we need is an **auto-tuning framework**

1

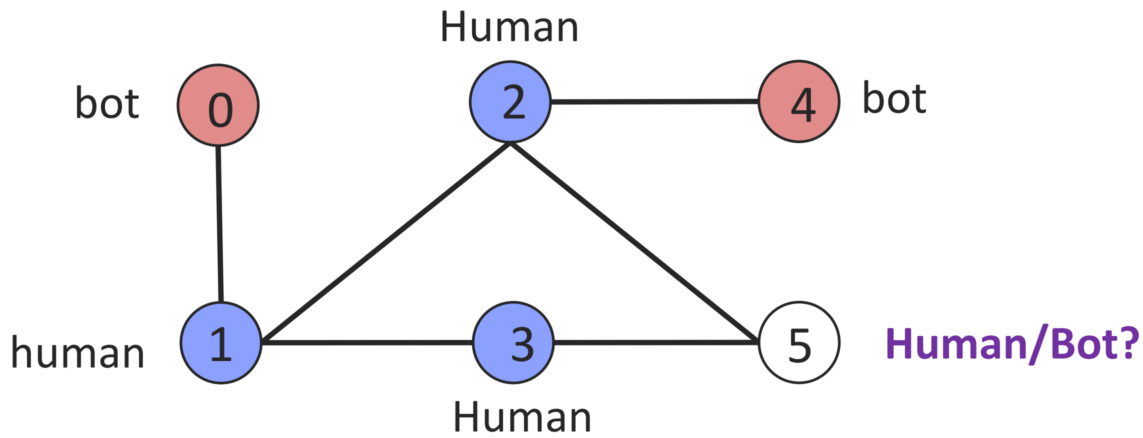
Learning on graphs:

All we need is

graph embedding

An example of learning on a graph

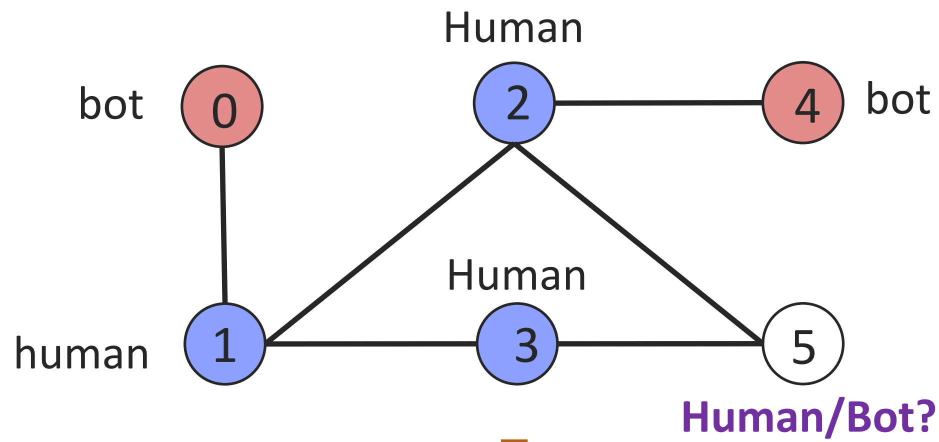
- Consider a node classification task



- A binary classification problem.
- Can we apply a traditional machine learning approach such as the **logistic regression**?
- In theory, yes! **It may not work well in practice. Why not?**

Graphs represent very high-dimensional data

- **What is the dimension of this graph?**
- Can be n (# of vertices)
  according to Erdős, Harary, Tutte
  - One row represents the connectivity features of a vertex
- **What is the problem?**
  - Curse of dimensionality
  - Need enormous training data
  - Standard machine learning methods do not work well

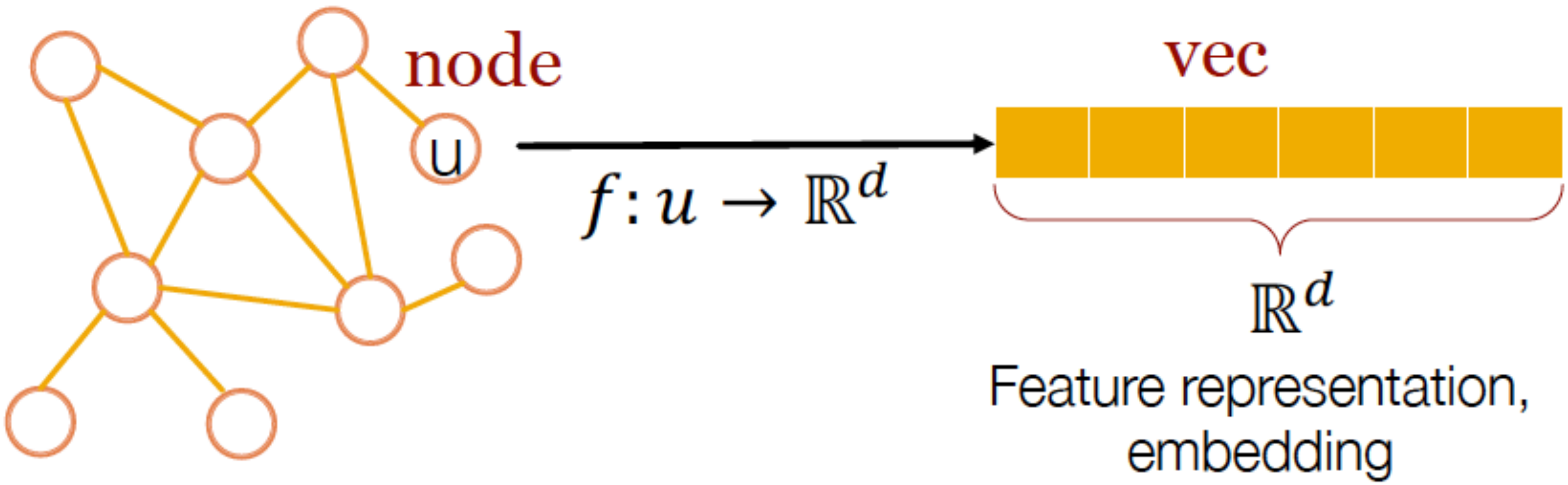


0	0	1	0	0	0	0
1	1	0	1	1	0	0
2	0	1	0	0	1	1
3	0	1	0	0	0	1
4	0	0	1	0	0	0
5	0	0	1	1	0	0

Erdős, Harary, and Tutte,, On the dimension of a graph, Mathematika, 1965

How to address: Node embedding

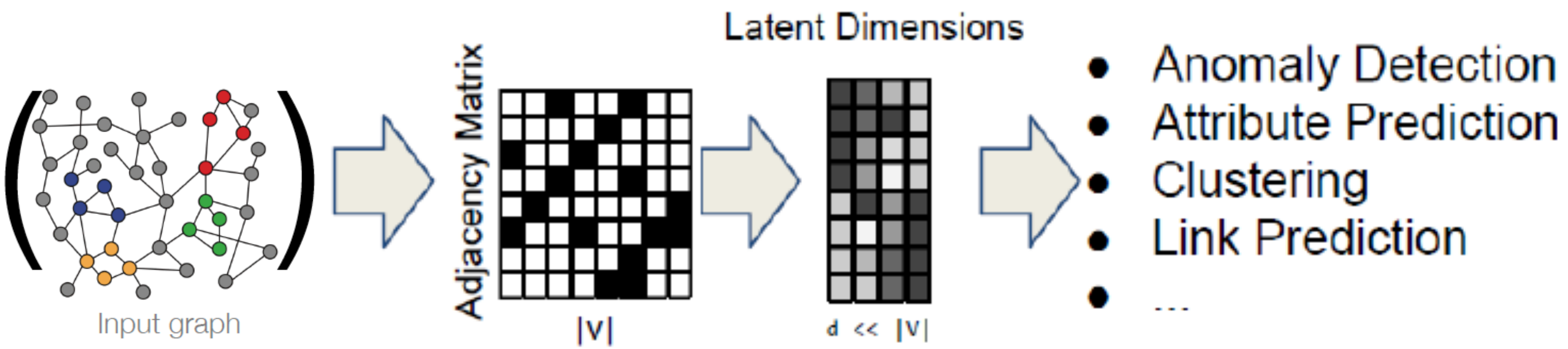
- Represent nodes by low dimensional vectors



d << n

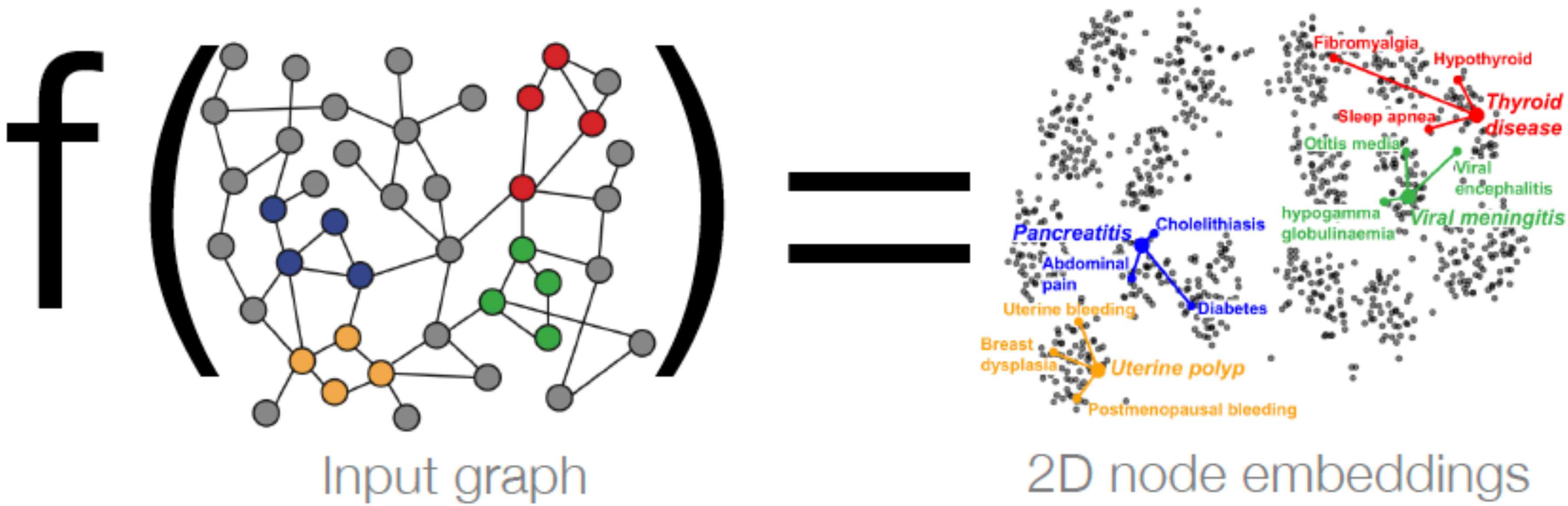
Node embedding

- Node embedding => dimensionality reductions



Credit: Jure Leskovec

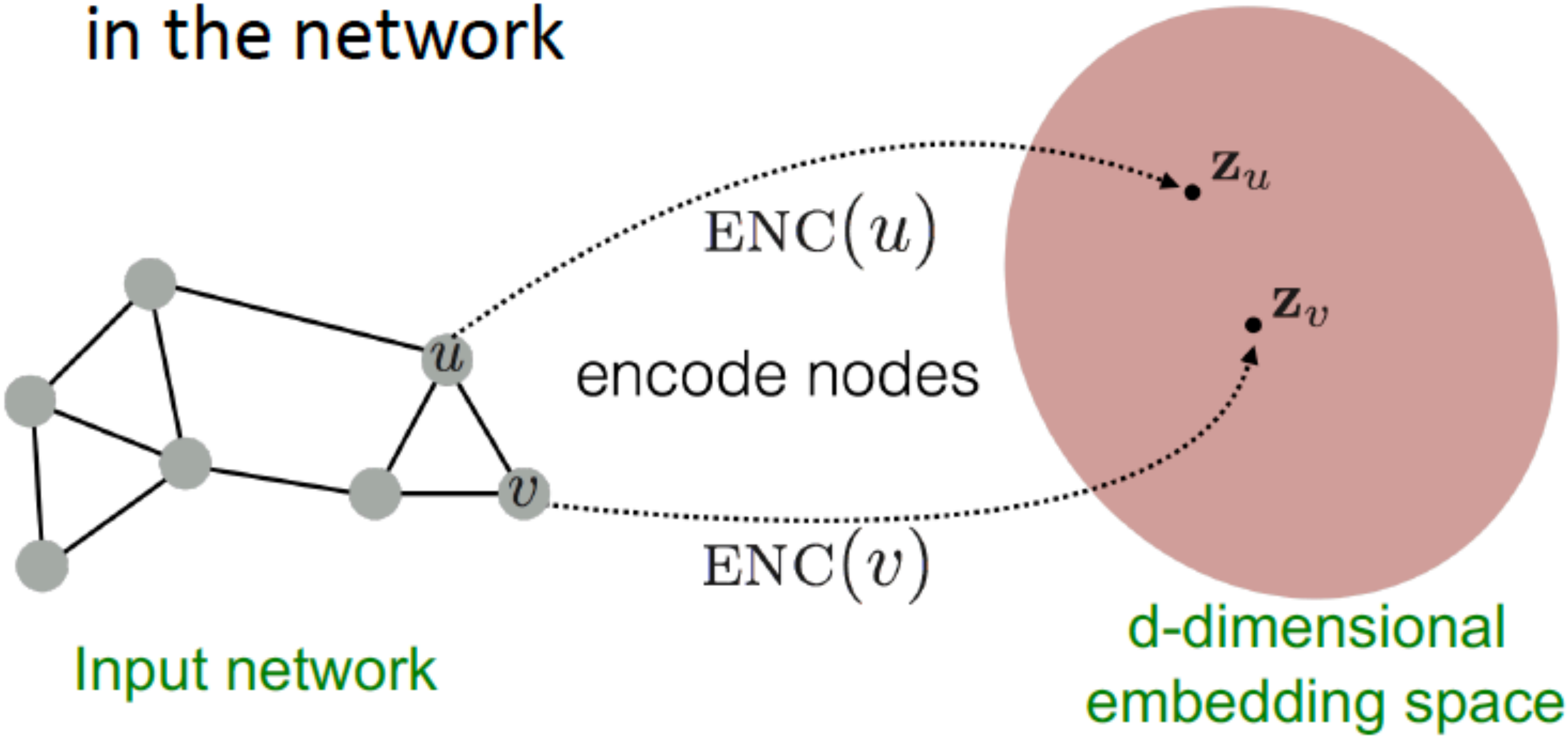
- **Intuition:** Map nodes to d-dimensional embeddings such that similar nodes in the graph are embedded close together



How to learn mapping function f?

Goal of node embedding

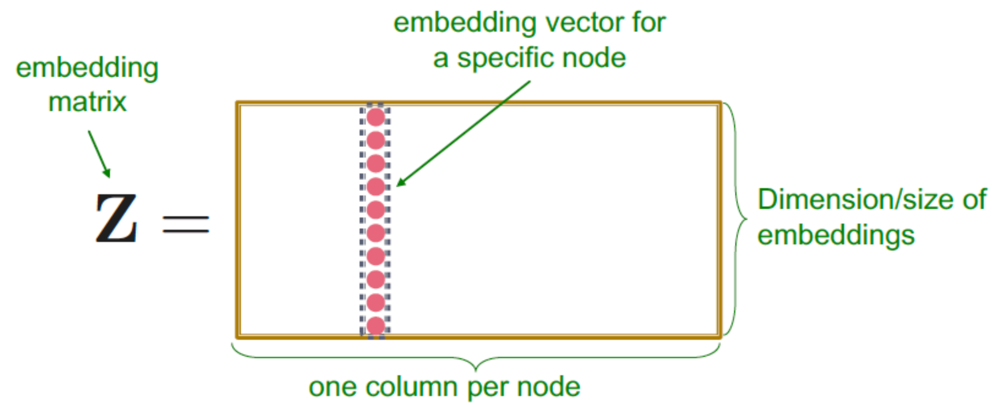
- **Goal:** Map nodes so that similarity in the embedding space (e.g., dot product) approximates similarity (e.g., proximity) in the network

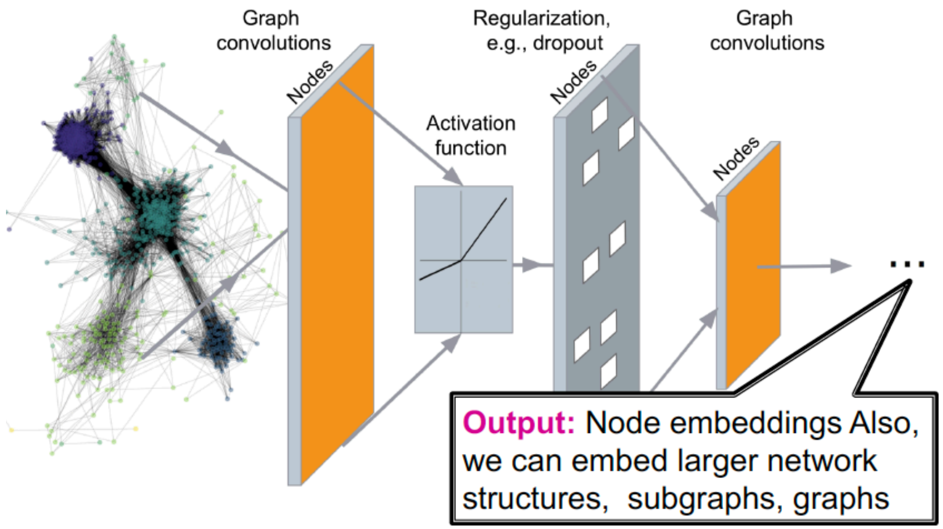


How to encode nodes: shallow/deep encoding

- ❑ Just embedding lookup or deep neural networks

Shallow encoding



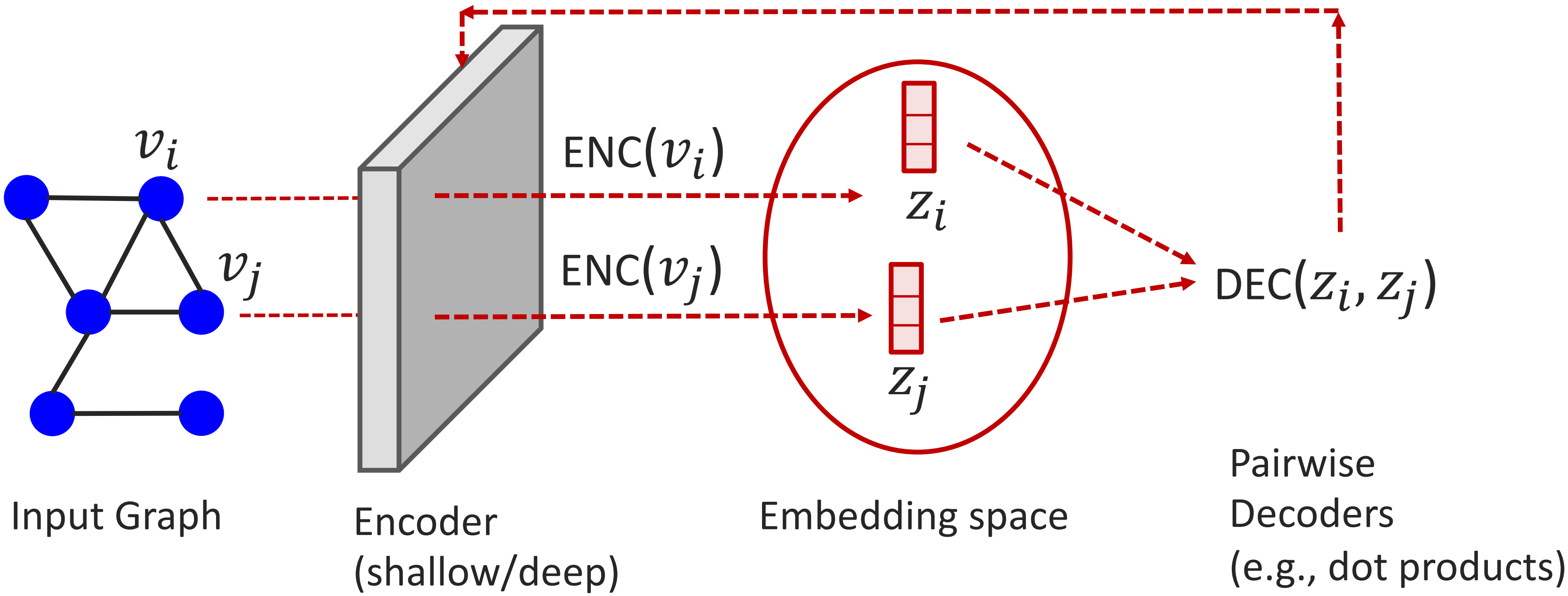


Deep encoding

The encoder-decoder approach

Proximity of nodes
in the original graph

$$Loss = \sum_{(v_i,v_j)\in V\times V} l\left(\mathrm{DEC}(z_i, z_j)\ , s_G(v_i, v_j)\right)$$



Most Graph ML methods follow variants of this approach
Node2vec, GNNs, Graph Visualization……

2

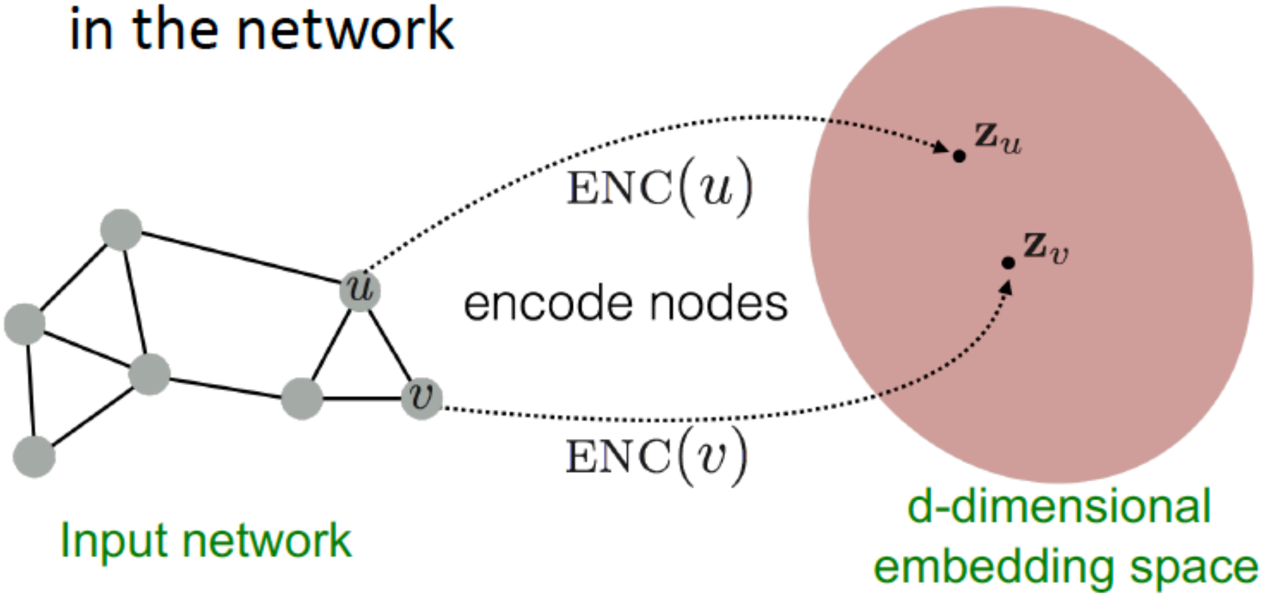
Computational patterns in graph embedding :

All we need is

Message passing

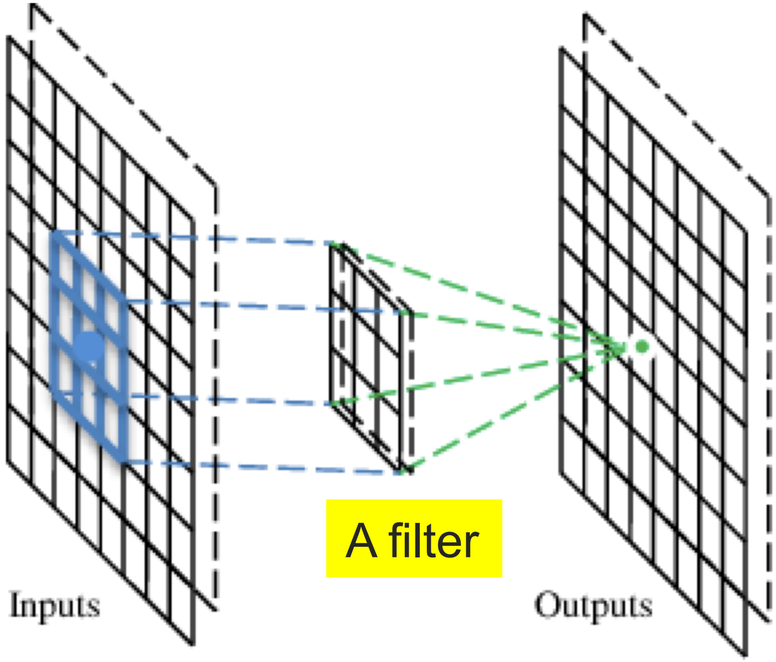
Node embedding goals

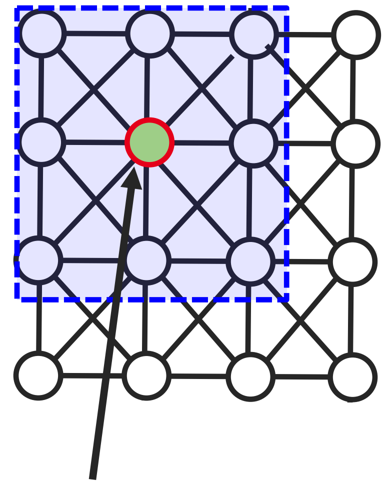
- **Goal:** Map nodes so that similarity in the embedding space (e.g., dot product) approximates similarity (e.g., proximity) in the network



- Lesson from the image embedding: how do we embed images in a latent space?

Embedding images to latent space

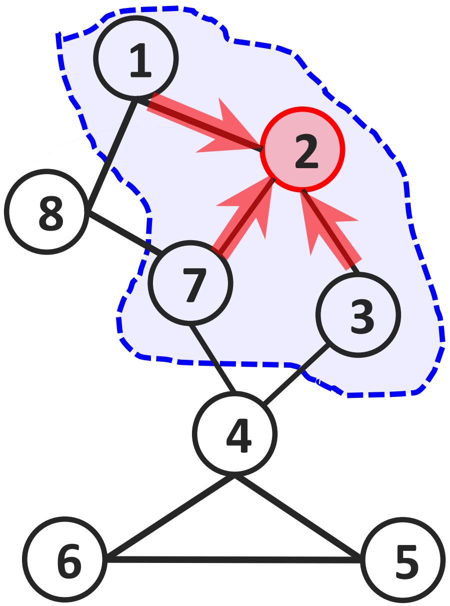




Graph analogy (embedding this node)

3x3 **dense** and **regular filter** based on neighboring pixels

Node embedding
based on information received from neighbors



Sparse and irregular relative to images

- Different filter size at different nodes
- Highly irregular

Transform information at the neighbors and combine it:

- Transform "messages" h_i from neighbors: $W_i \, h_i$
- Add them up: $\sum_i W_i \, h_i$

Message passing is all you need

The setting

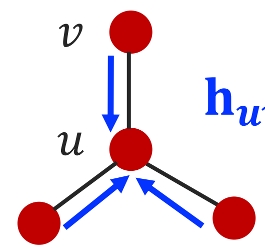
1. **Message generation** on an edge (v,u): $\mathbf{h}_{uv}$
   - Determined by a message generation function $\boldsymbol{\psi}(\mathbf{x}_u, \mathbf{x}_v, \mathbf{a}_{uv})$

2. **Message aggregation** on vertices
   - Determined by an aggregator

3. User defined: $\boldsymbol{\psi}, \boldsymbol{\phi}, \bigoplus$

vertex features

edge features



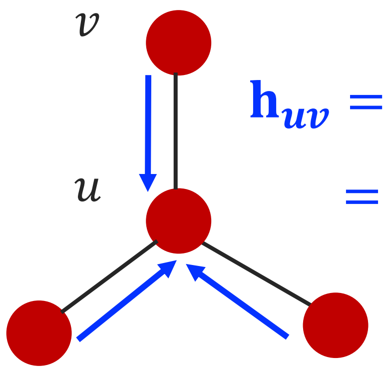
$$\mathbf{h}_{uv} = \boldsymbol{\psi}(\mathbf{x}_u, \mathbf{x}_v, \mathbf{a}_{uv})$$

$$\mathbf{z}_u = \bigoplus_{v \in N(u)} \boldsymbol{\phi}(\mathbf{x}_u, \mathbf{x}_v, \mathbf{h}_{uv})$$

Updated vertex features

Example: Graph Convolutions

1. **Message generation**: just pass node features
2. **Message aggregation:** sum messages (followed by non-linear activations)



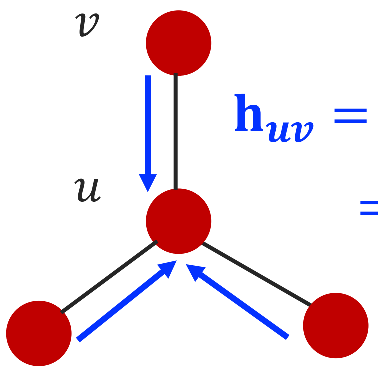
$$\mathbf{h}_{uv} = \mathbf{x}_u$$
$$= \psi(\mathbf{x}_u)$$

$$\mathbf{z}_u = \sum_{v \in N(u)} \mathbf{h}_{uv}$$
$$= \bigoplus_{v \in N(u)} \phi(\mathbf{h}_{uv})$$

Kipf and Welling, Semi-supervised classification with graph convolutional networks, ICLR 2017

Example: Graph Embedding

1. **Message generation**: dot product and then sigmoid
2. **Message aggregation:** elementwise multiply and then sum



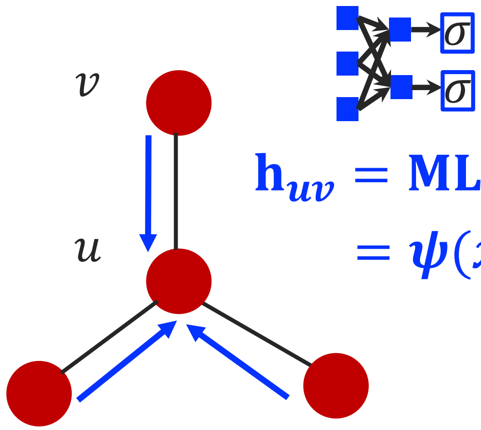
$$\mathbf{h}_{uv} = \sigma(\mathbf{x}_u^{\mathrm{T}}\mathbf{x}_v) = \psi(\mathbf{x}_u, \mathbf{x}_v)$$

$$\mathbf{z}_u = \sum_{v \in N(u)} \mathbf{h}_{uv} \odot \mathbf{x}_v = \bigoplus_{v \in N(u)} \phi(\mathbf{x}_v, \mathbf{h}_{uv})$$

Rahman, Sujon, Azad, Force2Vec: Parallel force-directed graph embedding, ICDM 2020

Example: Complex GNNs

1. **Message generation**: multilayer perceptron
2. **Message aggregation:** max pooling



$$\mathbf{h}_{uv} = \mathbf{MLP}(\mathbf{x}_u, \mathbf{x}_v) = \psi(x_u, x_v)$$

$$\mathbf{z}_u = \max_{v \in N(u)} \mathbf{h}_{uv} = \bigoplus_{v \in N(u)} \phi(\mathbf{h}_{uv})$$

Similarly for graph drawing

Thus, message passing models are widely used to implement almost all graph ML algorithms
(e.g., in Deep Graph Library and PyTorch Geometric)

3

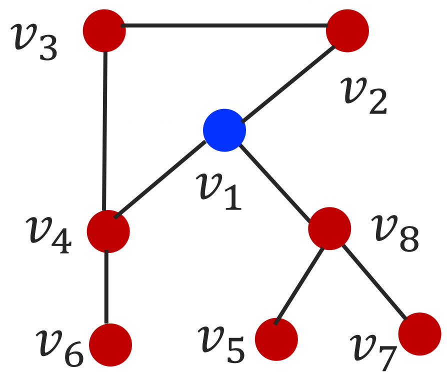
Central computations in message passing:

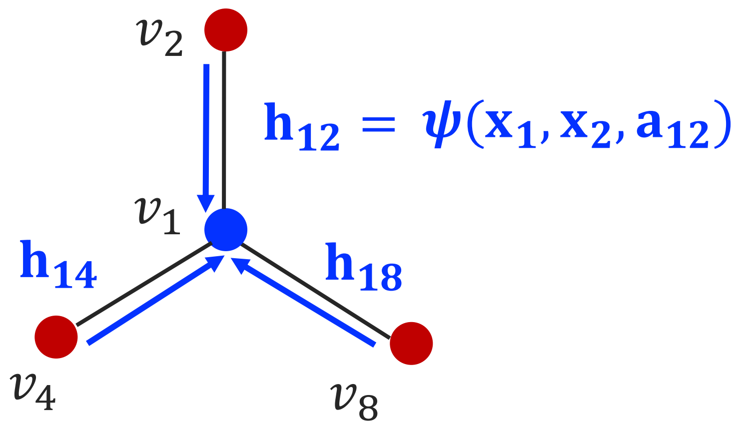
All we need is

sparse-dense (SpMM) and **sampled dense-dense matrix multiplications (SDDMM)**

Computational kernels in message generation

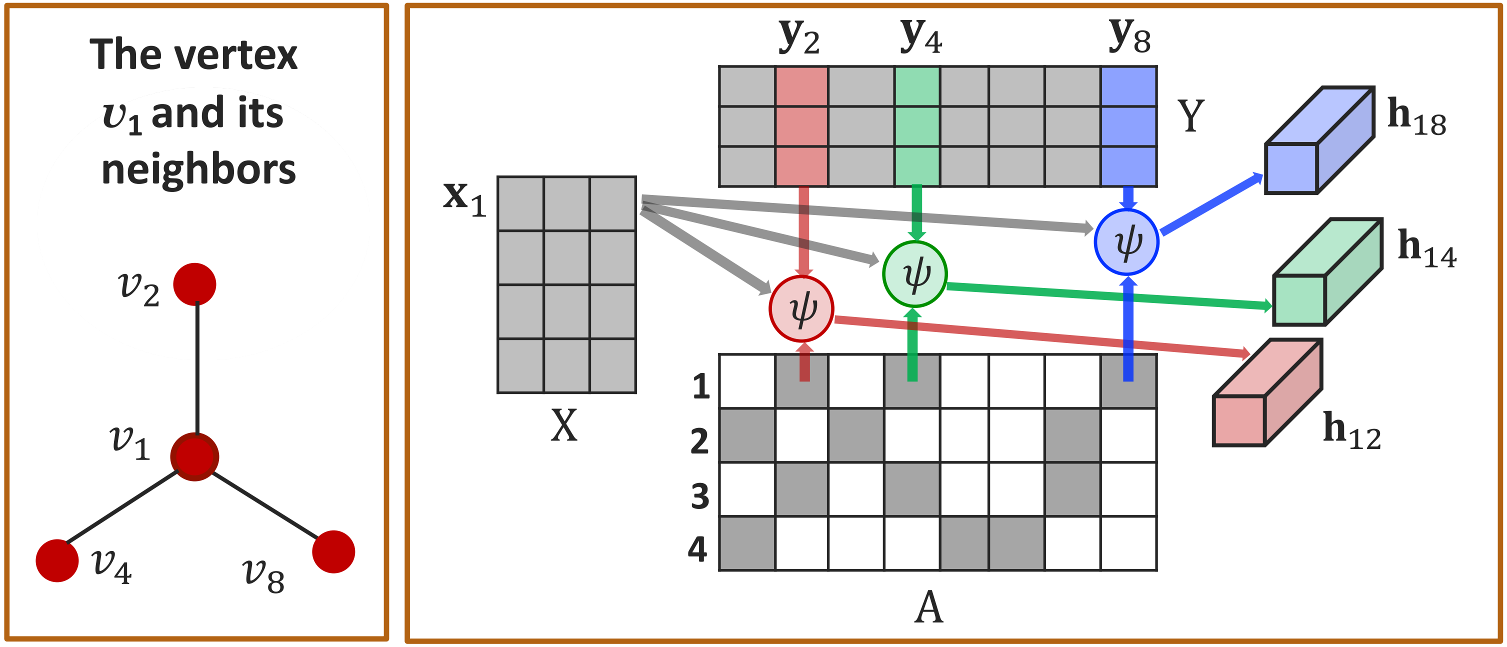
- Message generations on edges adjacent to v_1





Computational kernels in message generation

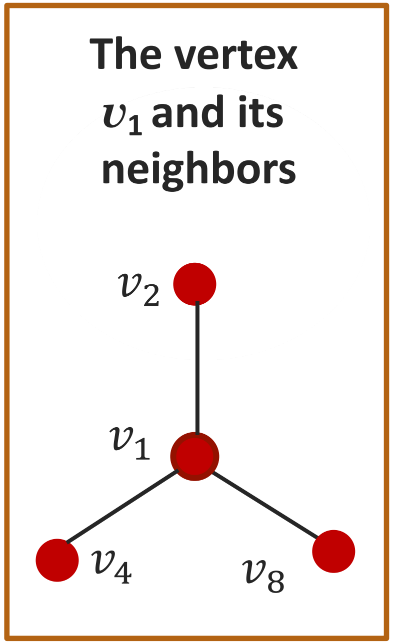
Using y for generality

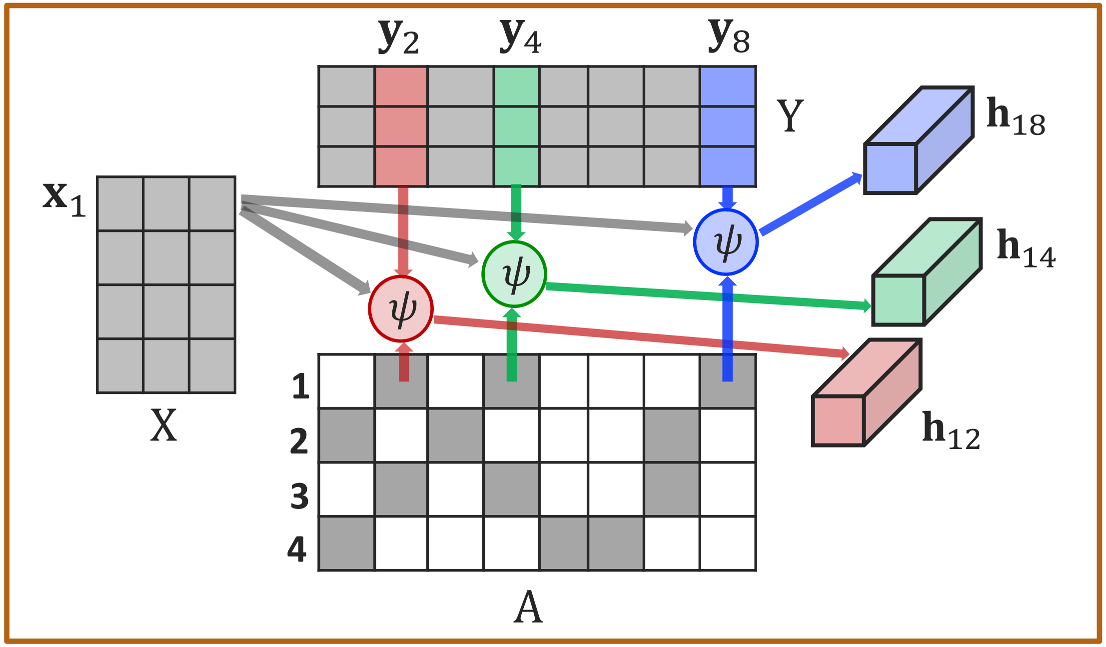


- ✓ Message can be scalars, or vectors depending on the operation
- ✓ We consider message generations on edges adjacent to v_1

Computational kernels in **message generation**

Using y for generality

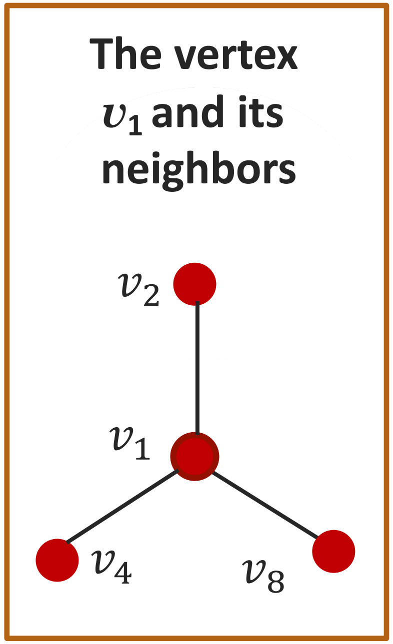


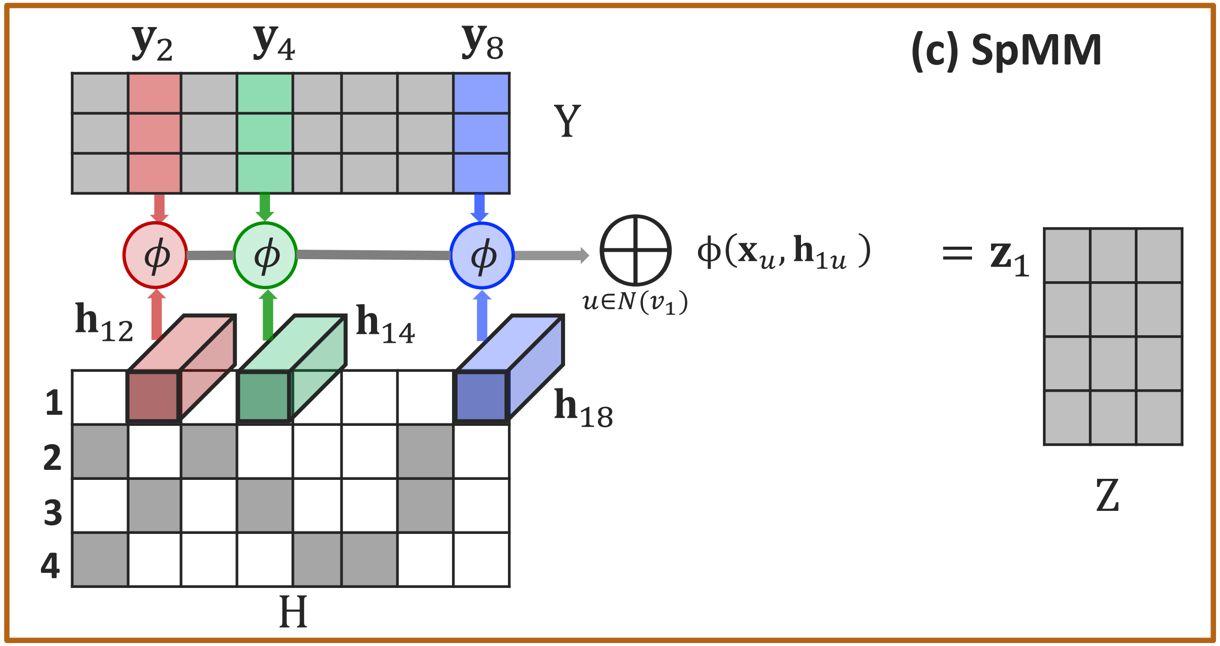


✓ This operation is called **SDDMM: Sampled Dense-Dense Matrix Multiplication**: $H = X \times Y^T \odot A$

Dense multiplication sampled by the adjacency matrix A

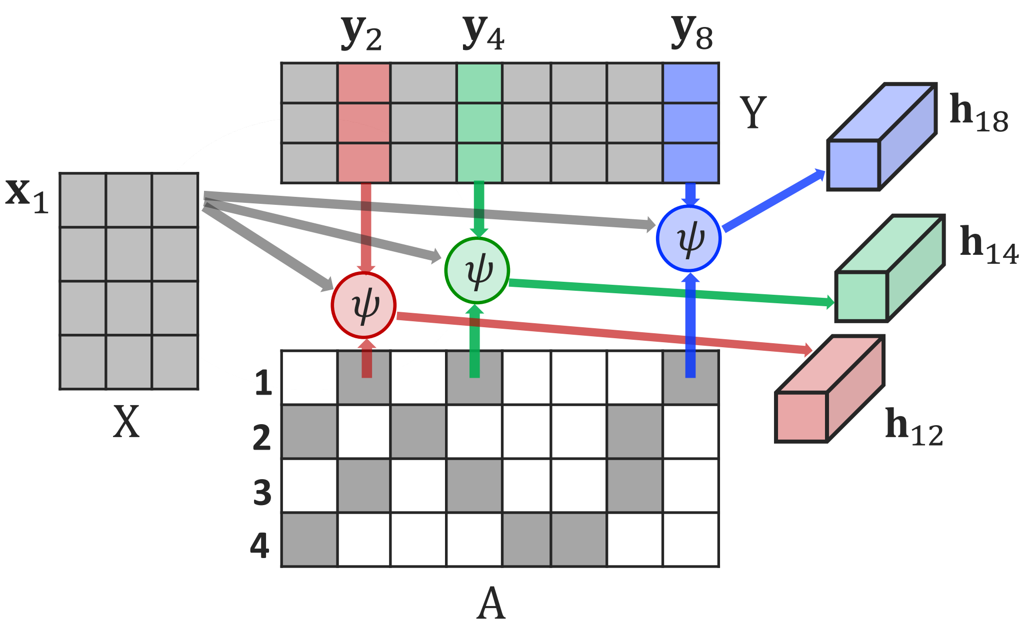
Computational kernels in **message aggregation**

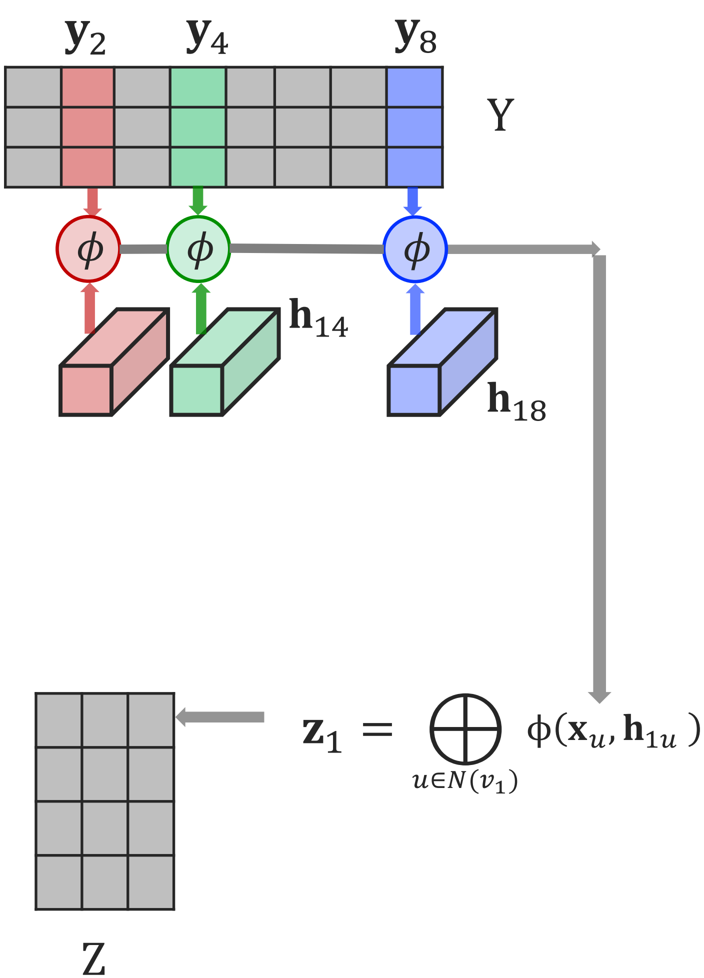




✓ This operation is called **SpMM: Sparse-dense matrix (or tensor) multiplication**: $Z = H \times Y$

FusedMM: SDDMM+SpMM





No intermediate messages

Rahman, Sujon, Azad, FusedMM: A Unified SDDMM-SpMM Kernel for Graph Embedding and Graph Neural Networks, IPDPS 2021 (to appear)

Message passing ML via Linear Algebra

- Graph ML libraries such as Deep Graph Library (DGL) and PyTorch Geometric (PyG) either implement SpMM and SDDMM or rely on vendor-provided code from Intel MKL and NVIDIA CuSPARSE

Thus, SpMM , SDDMM and a few other sparse kernels are all we need to fully implement message-passing based graph ML algorithms

4

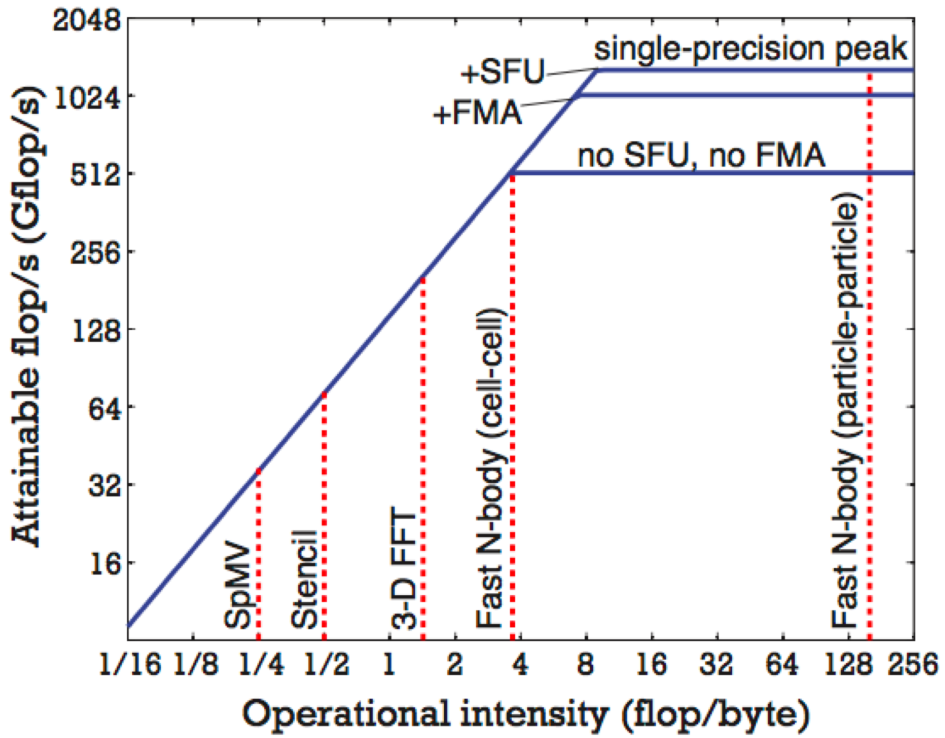
Efficient computations :

All we need is

Optimizing memory utilization and load balancing

Why does memory matter?

- Sparse operations are memory bound.
  - How do we know?
  - How to estimate the peak performance?
- **Roofline model**



Why does memory matter?

- **Roofline model**
- Operational intensity (also called arithmetic intensity) of FusedMM (message generation + aggregation):

$$\frac{\delta}{\frac{3\delta}{d}+2+\delta},$$

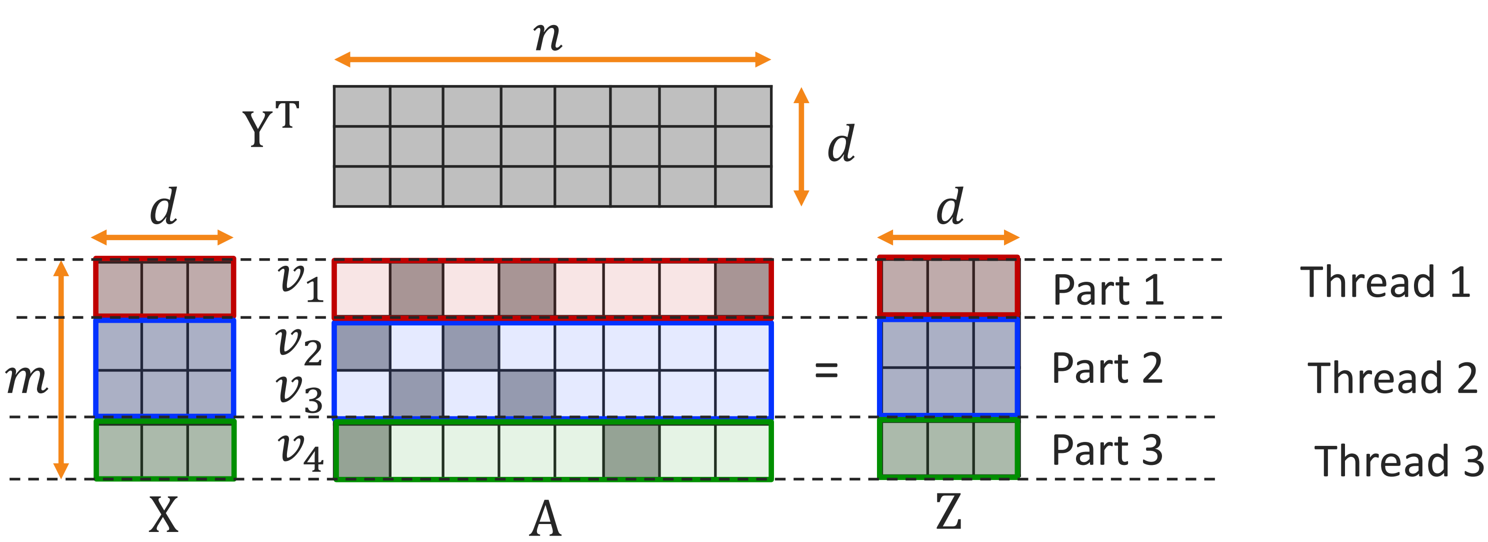
- where δ is the average degree of the graph and d is the embedding dimension
- For example, for δ=16, d=128: the arithmetic intensity is close to 1
- Hence, operations are almost always **memory bound**

How to optimize memory?

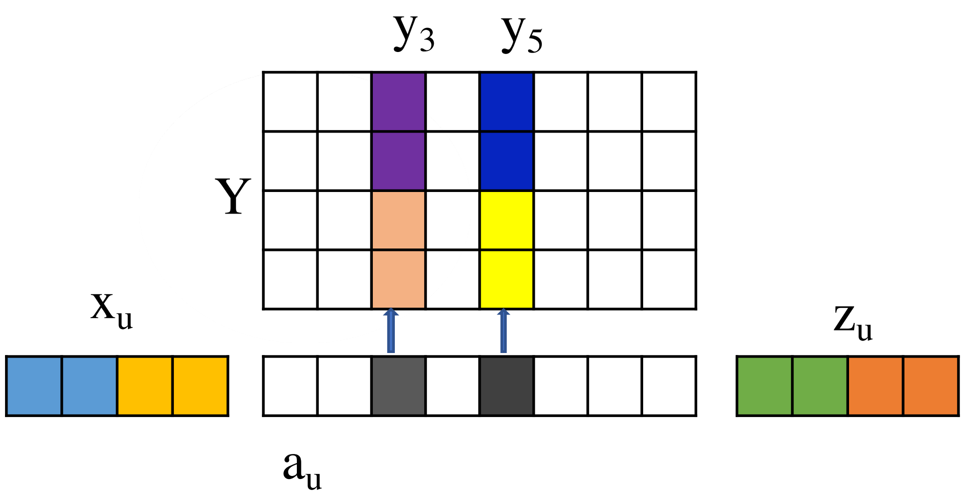
- **Utilize memory bandwidth**: Stream data
- **Utilize temporal locality**
  - Load data to cache/register once and use many times

Parallelization
(with an aim to minimize memory traffic)

- Partitioning (simple 1D) with balanced nonzero distributions (other partitioning possible)
- Access X, A and Z once (in most cases)
  - May access Y several times

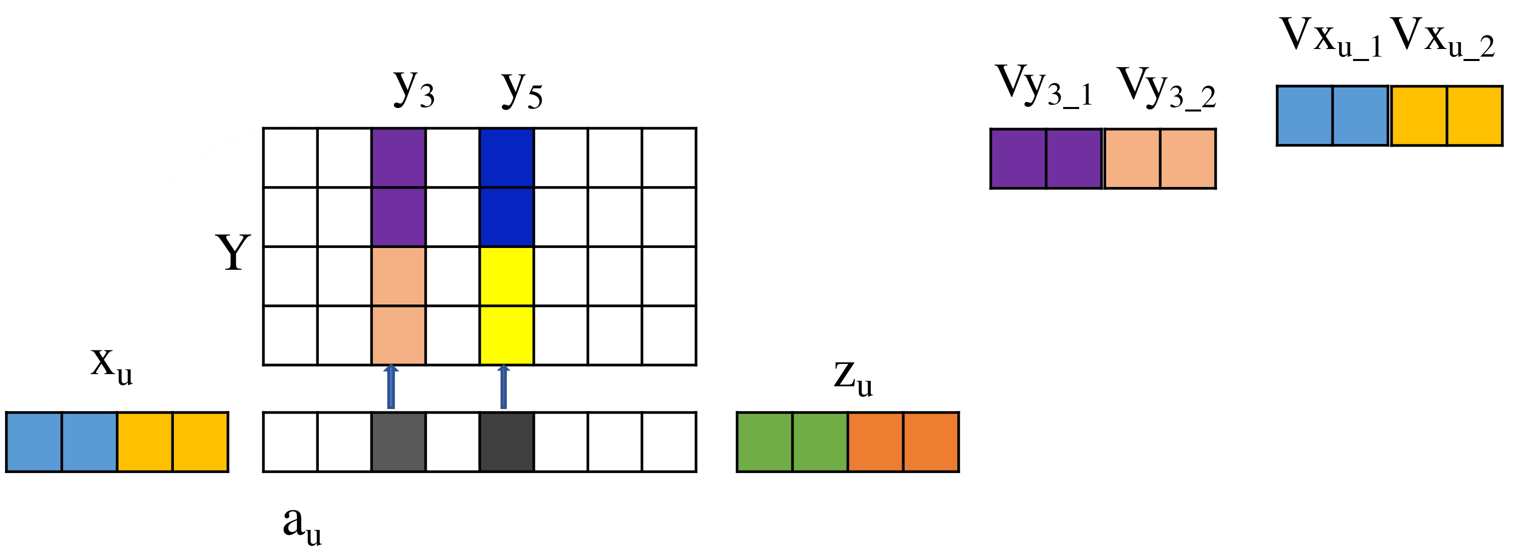


Temporal locality via register blocking



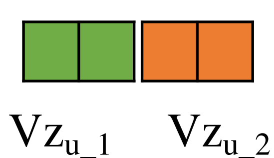
Suppose register length=2

Temporal locality via register blocking



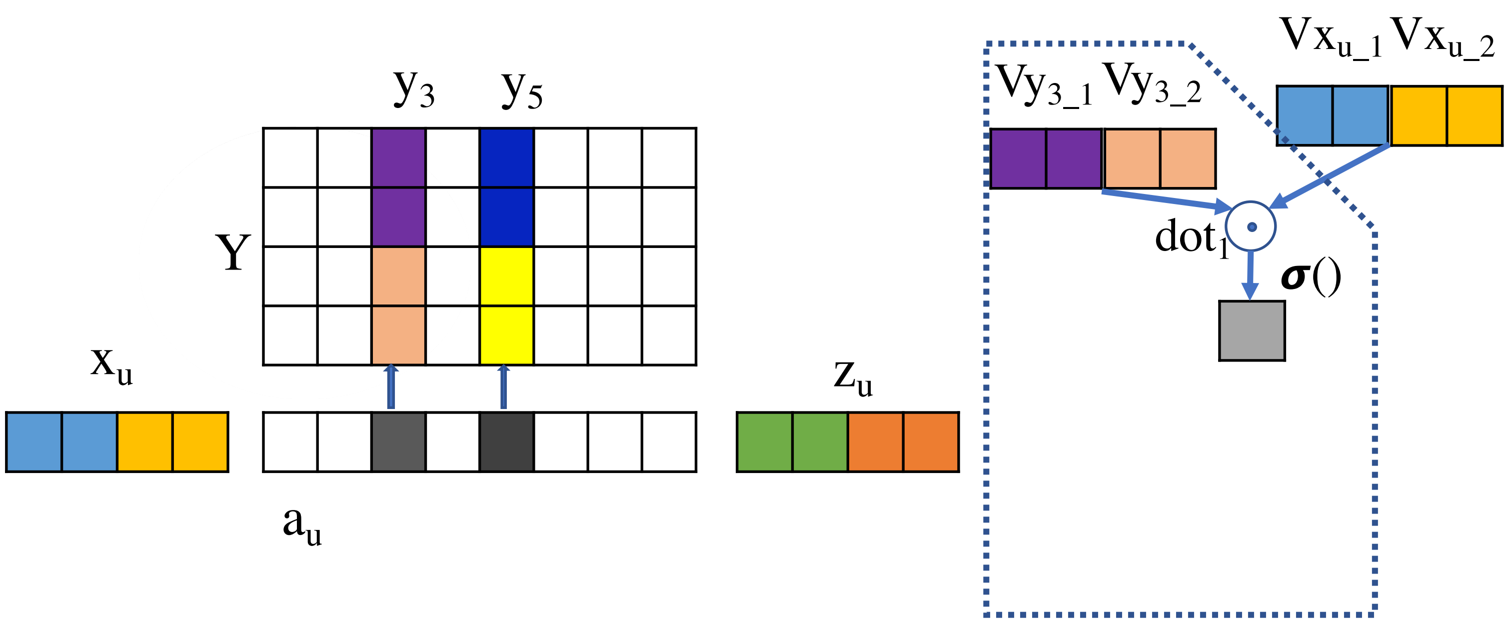
Suppose register length=2

Input: Load necessary data to 4 registers
Output: Reserve two registers

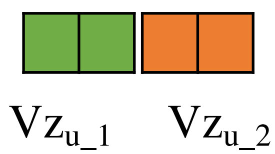


and perform the entire computation

Temporal locality via register blocking

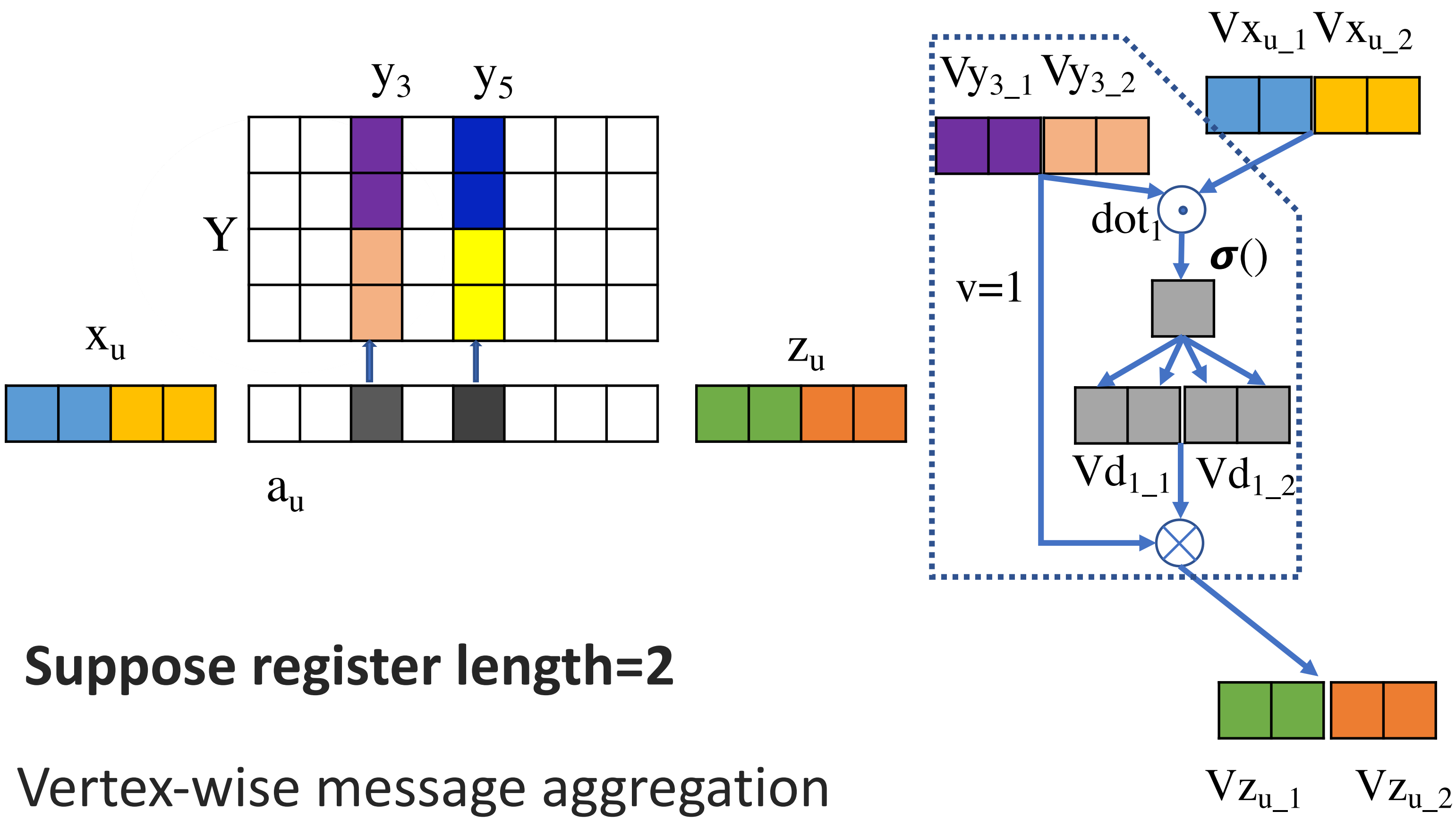


Suppose register length=2



Edge-wise message generation for the edge (u,3)

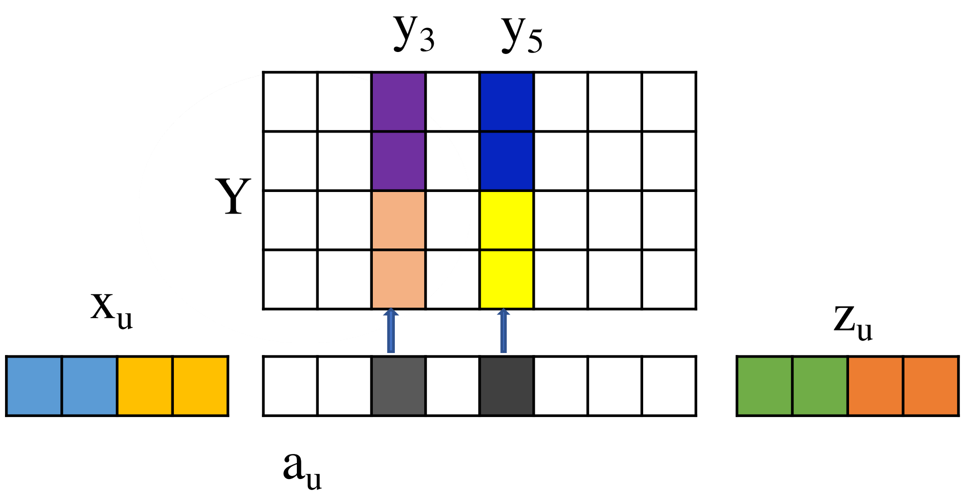
Temporal locality via register blocking



Suppose register length=2

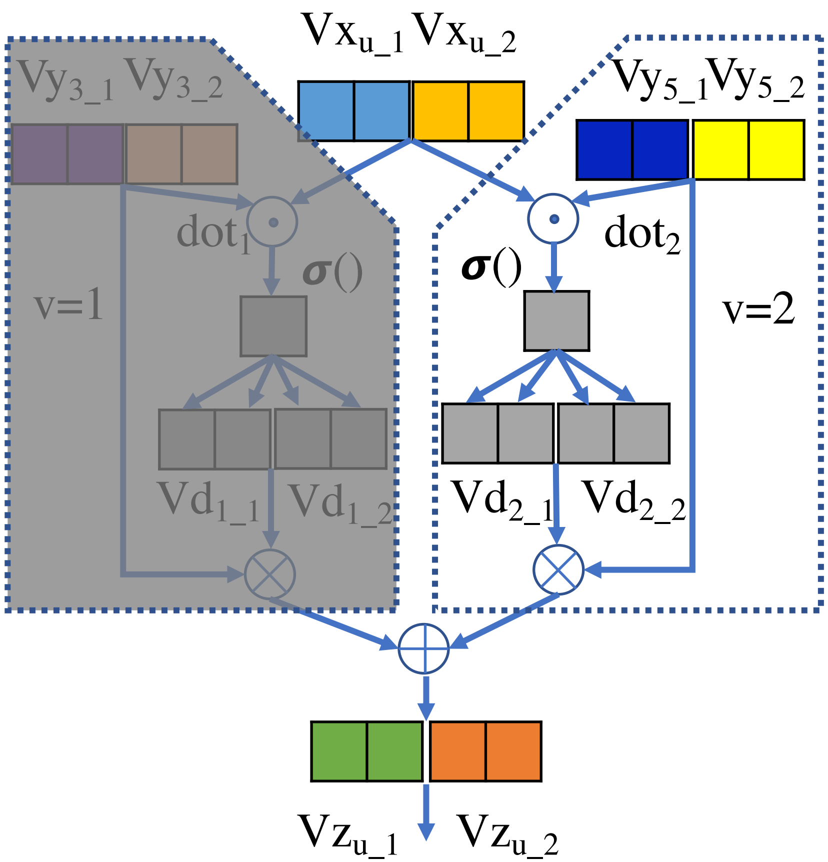
Vertex-wise message aggregation
(partial via edge (u,3))

Temporal locality via register blocking



Suppose register length=2

Next for the edge (u,5)
All necessary vectors (x and z)
are still in registers
(full data reuse)

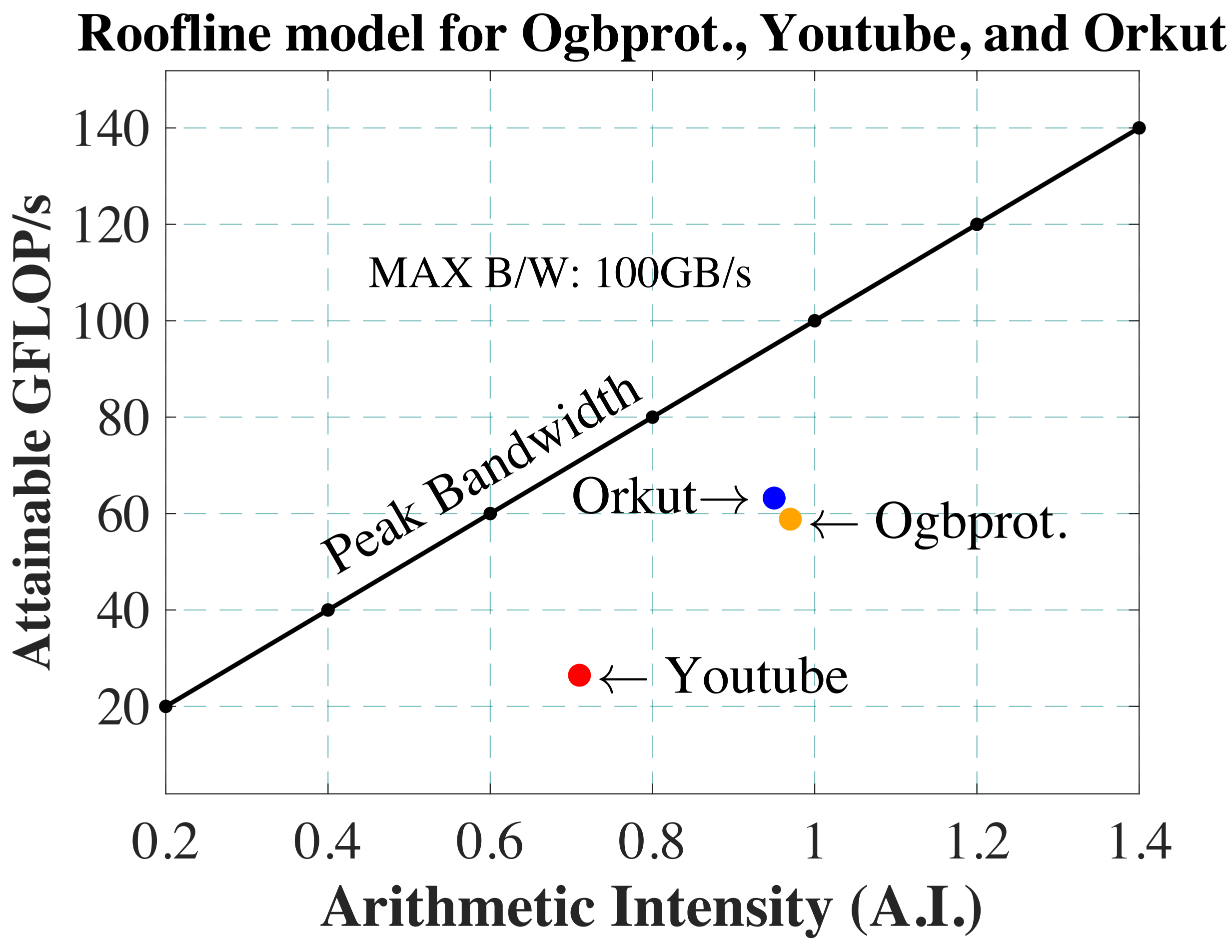


Non temporal Memory write

Temporal locality via register blocking

- Register blocking (within each thread)
  - Works perfectly fine as long as all data (for one vertex) fits in available registers
  - Otherwise, we will reload registers from cache

Observed performance (Intel Skylake: 100GB/s bandwidth)
Example: Graph Embedding



Still does not achieve the best performance

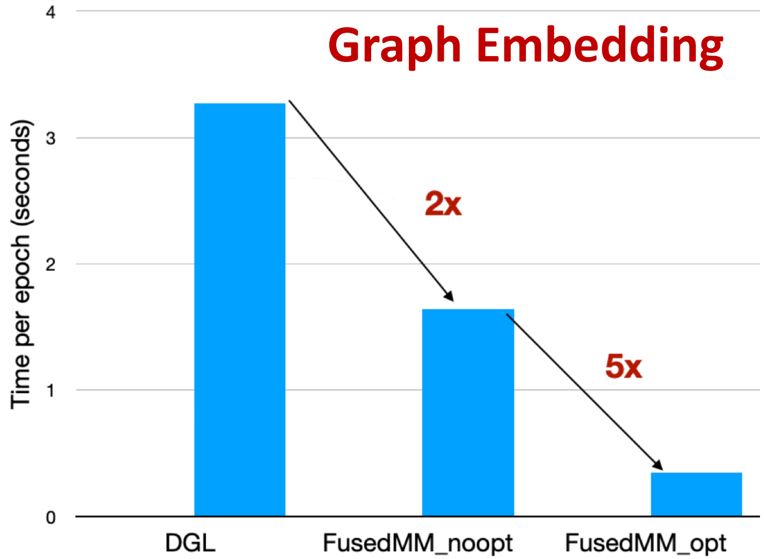
Runtime measured from a Python interface

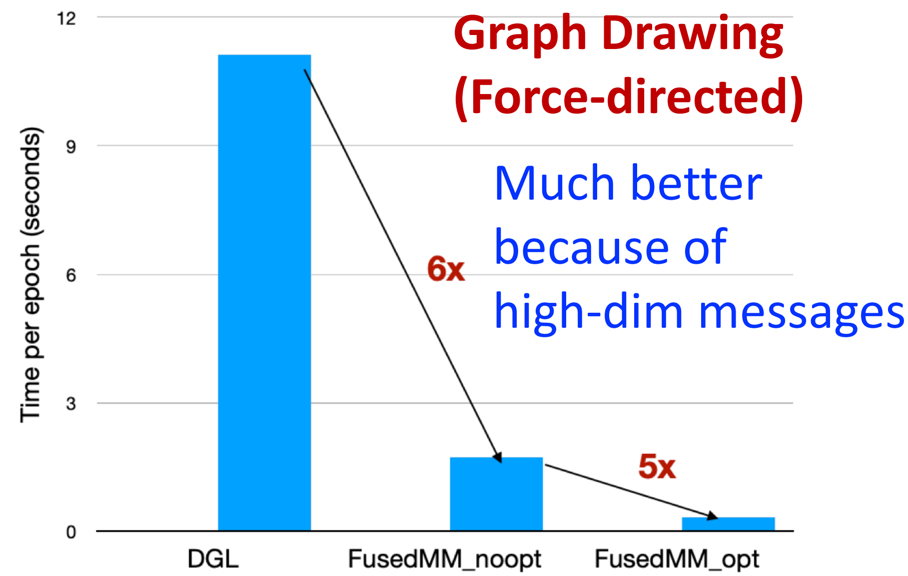
We speculate: Python to C++ interface is slowing things down

Compare with Deep Graph Library

- Deep Graph Library (DGL)
  - Uses C++ backend
  - We consider DGL based on PyTorch
  - DGLS also uses SDDMM and SpMM in the backend

Graph embedding on Intel Skylake



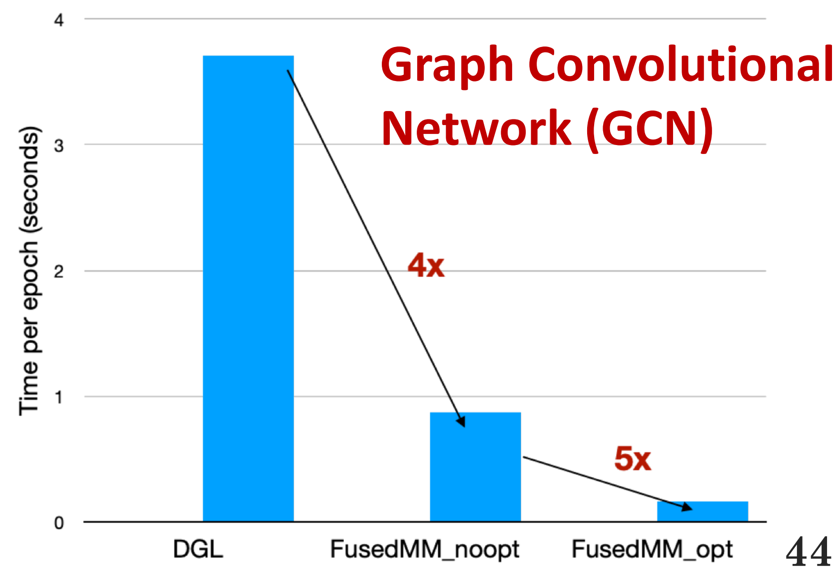


Just SDDMM
+
SpMM

Just
FusedMM

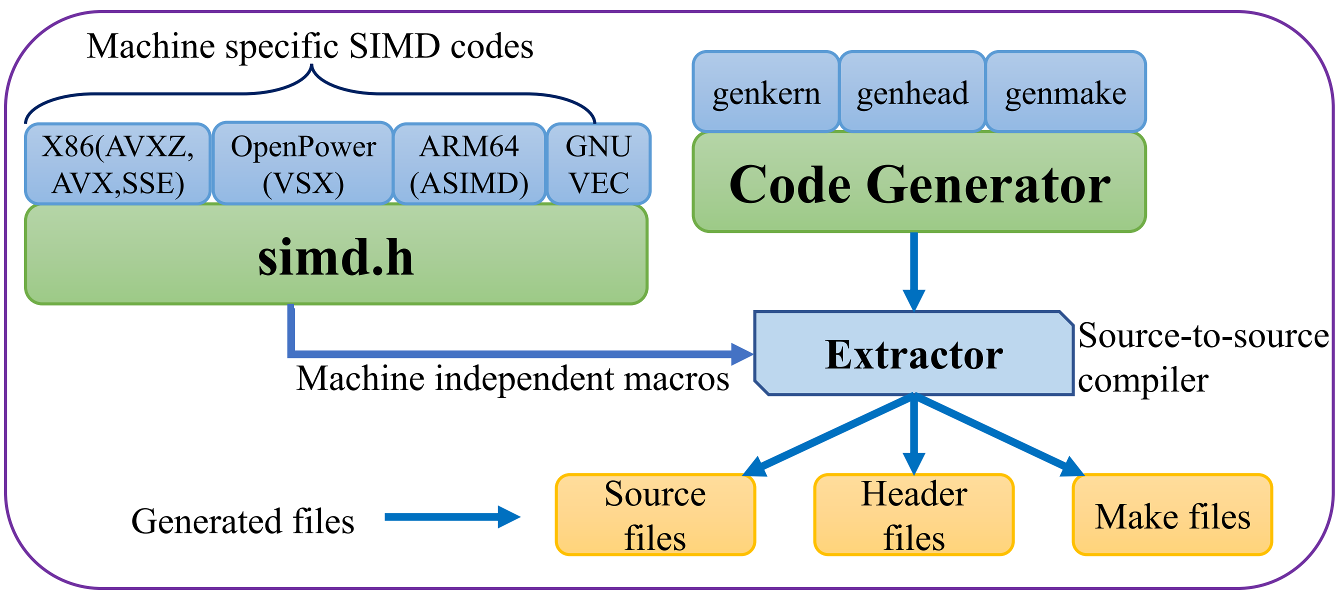
FusedMM
with register
Blocking and
Other tricks

Ogbprot. Graph
Vertices 132K
Edges 39M



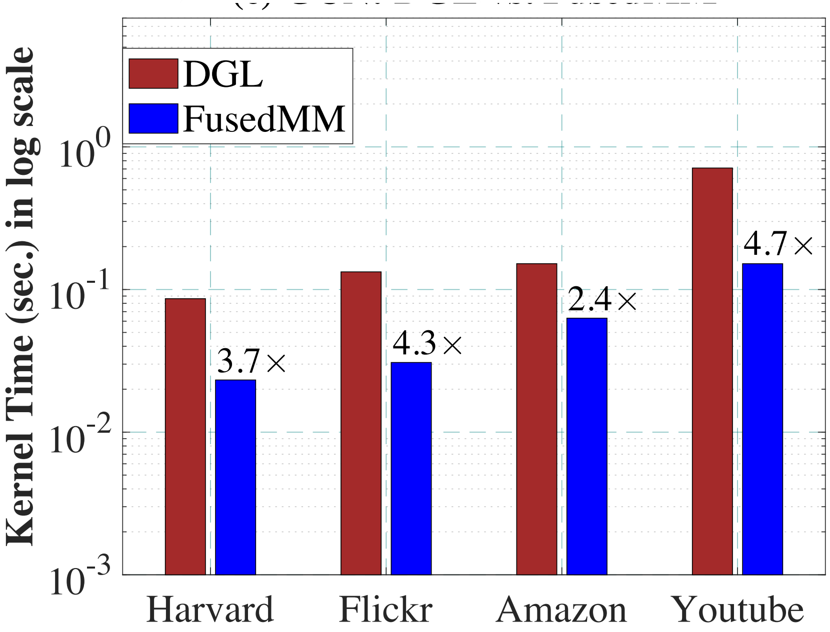
Performance portability

- What about other processors (ARM/AMD/IBM)?
- We developed a code generator for different Single Instruction Multiple Data (SIMD) units
  - Based on an autotuned linear algebra library called ATLAS (Clint Whaley; Antoine Petitet, Jack J. Dongarra , 2001)

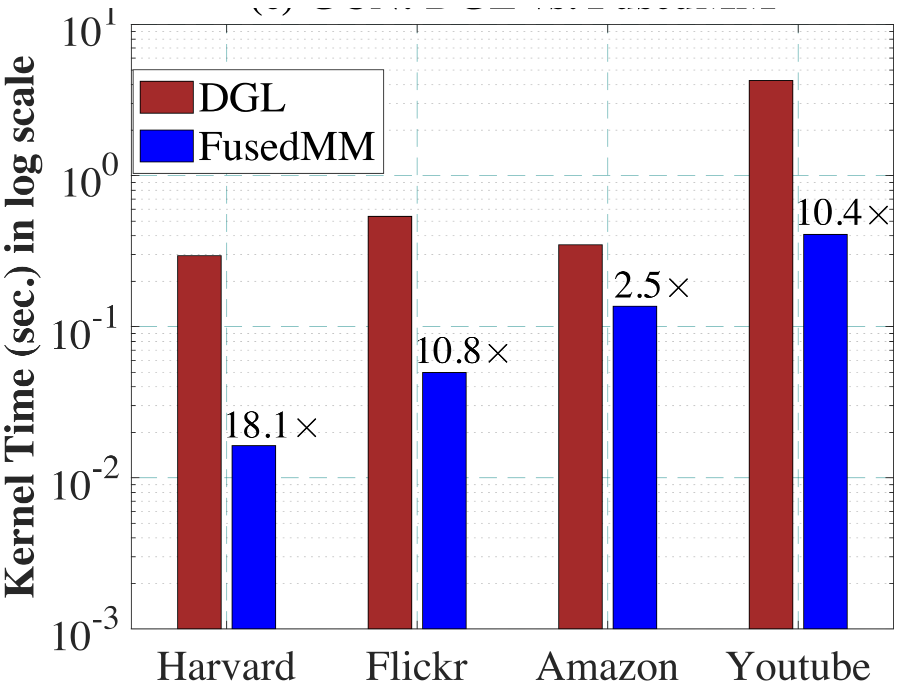


GCN on AMD and ARM processors

Same algorithms with automatically tuned code for processors



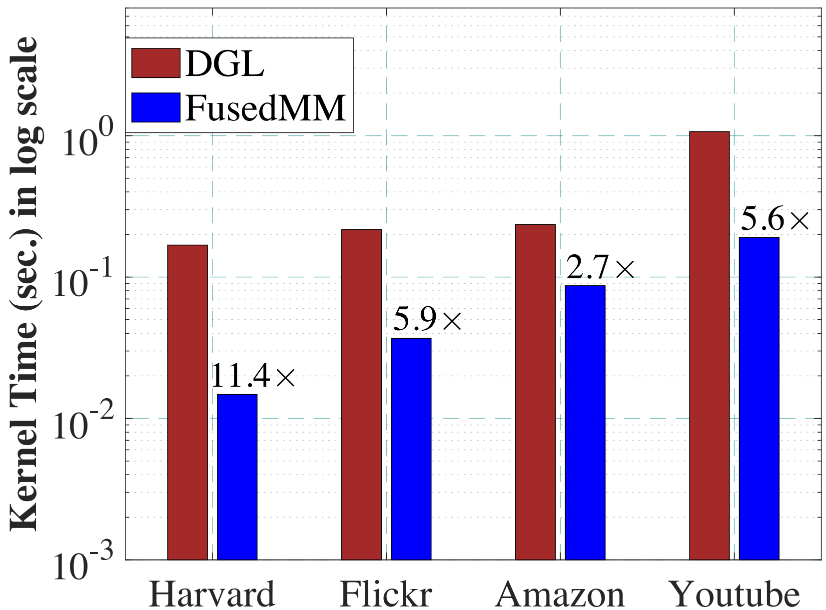
AMD



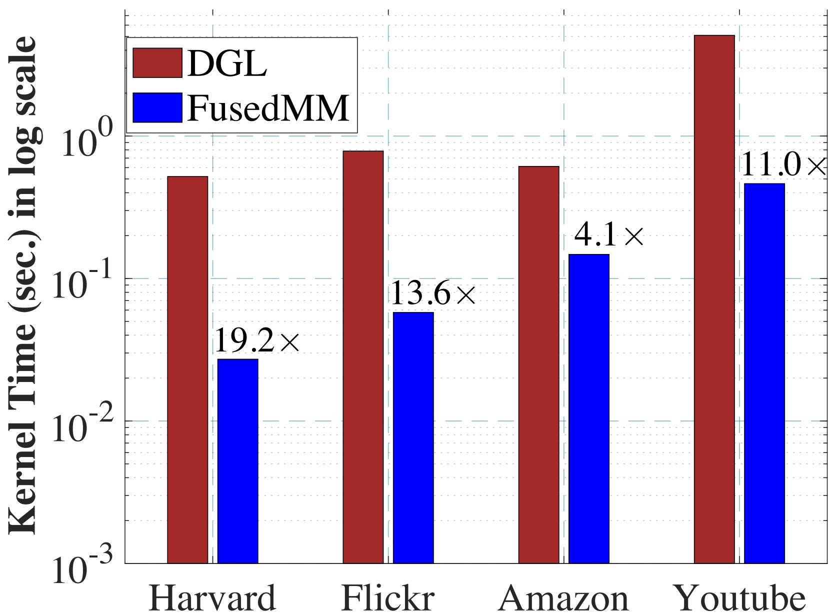
ARM

Graph drawing on AMD and ARM processors

Same algorithms with automatically tuned code for processors



AMD



ARM

End-to-end Performance

- Graph embedding from python (end-to-end)
- Same algorithm => **same accuracy**

Graphs	Method	Total Time (Sec.)	Speedup
Cora	PyTorch	0.342	48.9×
	DGL	0.177	25.3×
	FusedMM	**0.007**	1.0×
Pubmed	PyTorch	2.590	45.4×
	DGL	1.415	28.3×
	FusedMM	**0.057**	1.0×

Thus, optimized and portable sparse kernels (SpMM + SDDMM) speed up various graph ML algorithms significantly

Summary

1. **Learning on graphs**: All we need is **graph embedding**
   - Shallow embedding/deep embedding

2. **Computational patterns in graph embedding:** All we need is **passing messages among nodes**

3. **Centra**… eed is **sparse** … **atrix multip**…
   - Exam…

We developed FusedMM and experimentally validated these hypotheses

Rahman, Sujon, Azad (IPDPS 21)

4. **Efficient computations**: All we need is **optimizing memory utilization** and load balancing

5. **Portable implementations:** All we need is an **auto-tuning framework**

What next?

- The evidence tells us that sparse linear algebra can help Graph ML run faster on CPUs and GPUs

- Two questions that remain mostly unanswered
  - How to exploit **sparsity**, e.g., sparse embedding?
    - NVIDIA's new GPUS (A100) will have limited features for sparse operations
  - How to **distribute** the computations, e.g., in supercomputers?

Future direction: Distributed Graph Neural Network

- Main equation in forward propagation

$$\mathbf{H}^{(\mathbf{l})} = \boldsymbol{\sigma}(\mathbf{A}\mathbf{H}^{(\mathbf{l-1})}\mathbf{W}^{(\mathbf{l})})$$

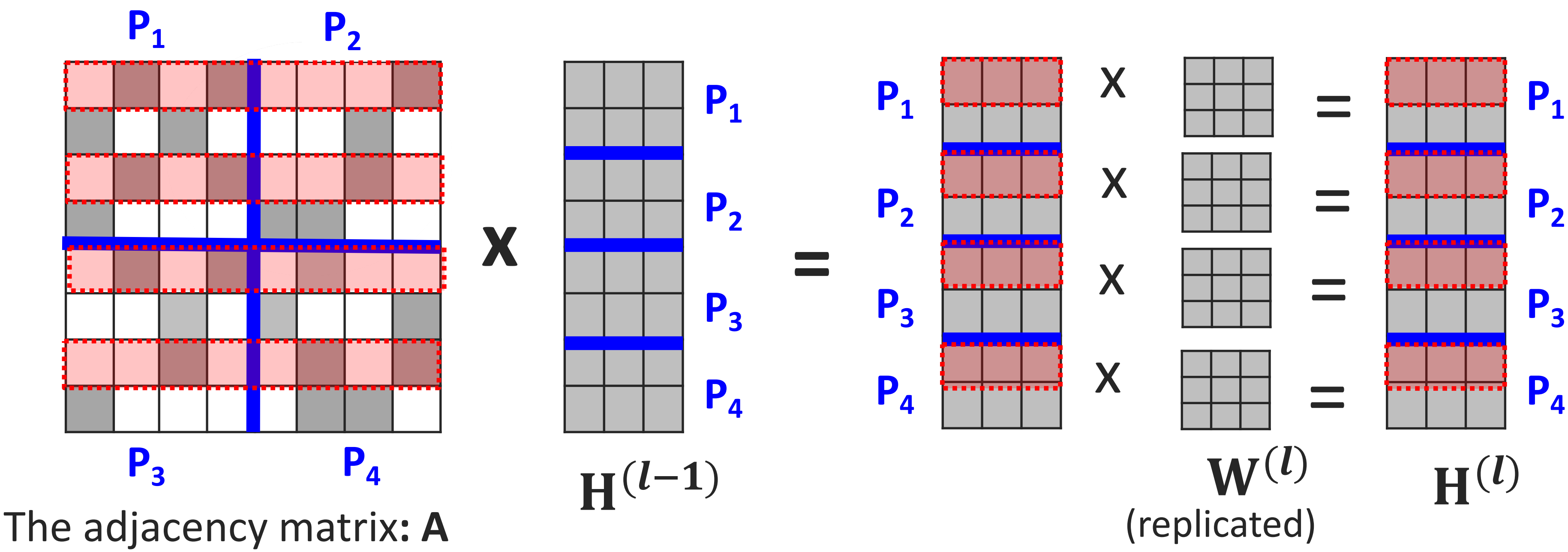
nxn sparse matrix
(graph adjacency)
n: number of vertices

nxd dense matrix
(embedding matrix)
d: embedding dimension

dxd dense matrix
(weight matrix)

We aim to make them sparse without sacrificing accuracy

Parallelizing GNN (1D/2D/3D matrix multiplications)



$$\mathbf{H}^{(\mathbf{l})} = \boldsymbol{\sigma}(\mathbf{A}\mathbf{H}^{(\mathbf{l}-\mathbf{1})}\mathbf{W}^{(\mathbf{l})})$$

Joint work with

Md. Khaledur Rahman and Majedul Haque Sujon

Indiana University

Thanks for your attention