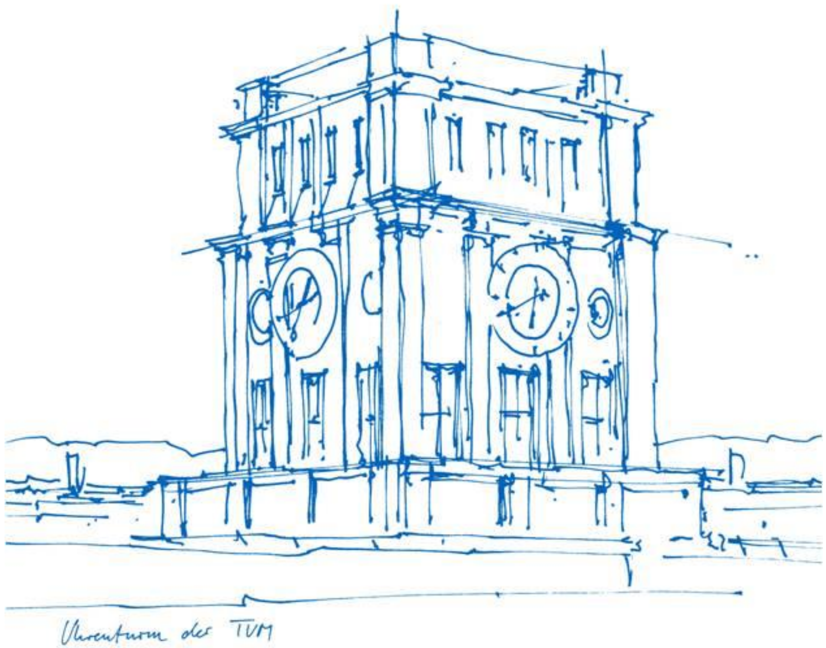
FusedMM- A Unified SDDMM-SpMM Kernel for Graph Embedding and Graph Neural Networks

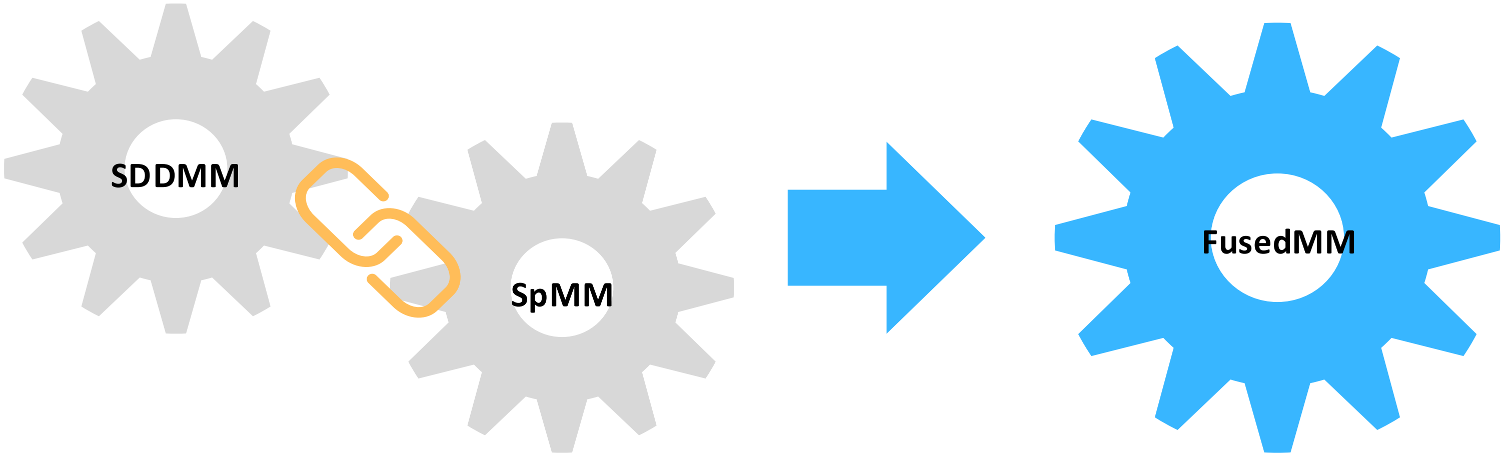
Haru Kobayashi



What is FusedMM?

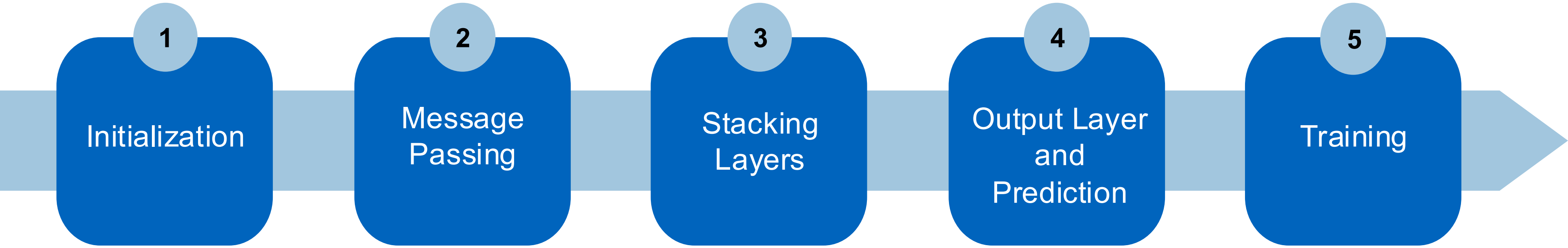
A single operation to replace the separate SDDMM and SpMM steps in Graph Neural Networks and Graph Embedding.

It is 34x faster than its equivalent kernels in Deep Graph Library



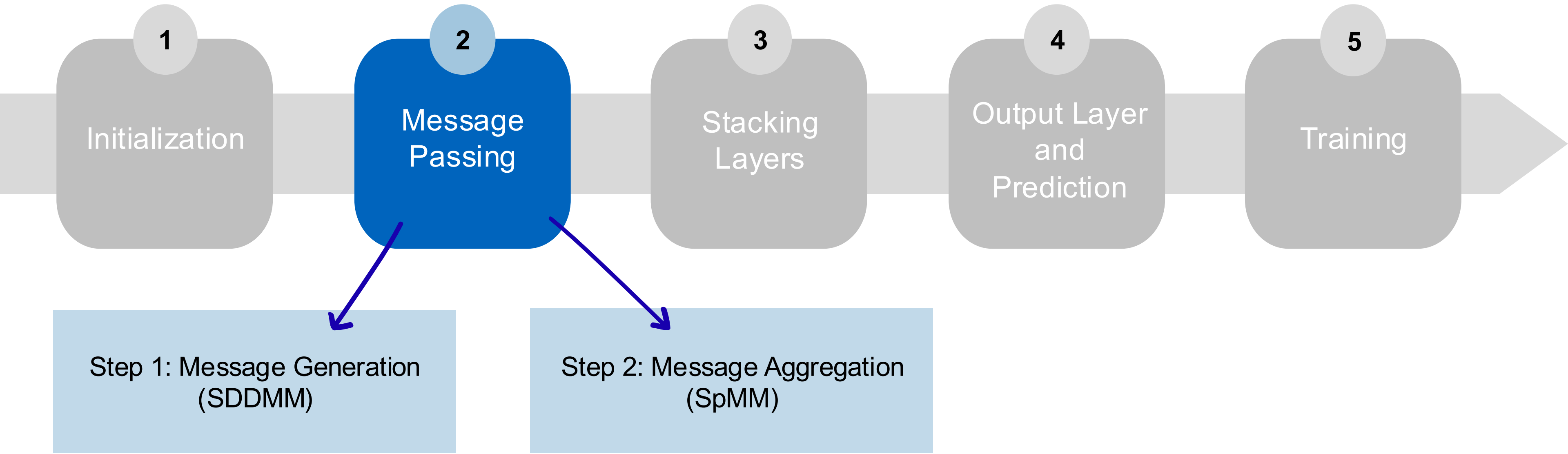
The Workflow of a GNN

The workflow of a traditional GNN is 5 steps.



The Core Operation of a GNN

Traditional GNN separates “Message Passing” phase into two steps:



Generate unique message for every node

Collect and re-read, process all messages

Current Framework Limitations

Traditional GNN separates “Message Passing” phase into two steps:
... **forcing applications to generate <u>intermediate outputs</u> from SDDMM.**

Step 1: Message Generation (SDDMM)

Step 2: Message Aggregation (SpMM)

$$\mathbf{H} = (\mathbf{X} \times \mathbf{Y}^T) \odot \mathbf{A}.$$

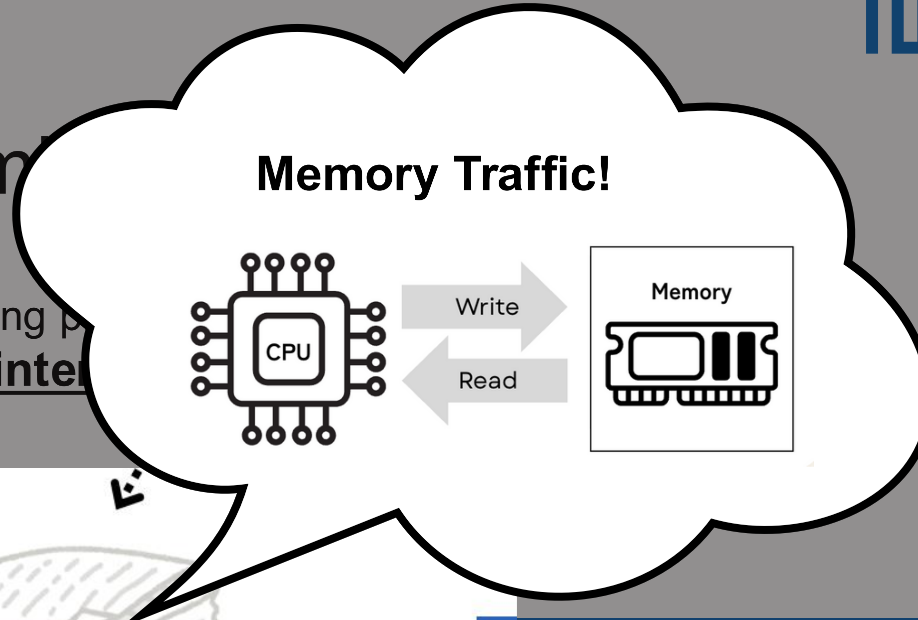
$$\mathbf{H}$$

$$\mathbf{Z} = \mathbf{H} \times \mathbf{Y}$$

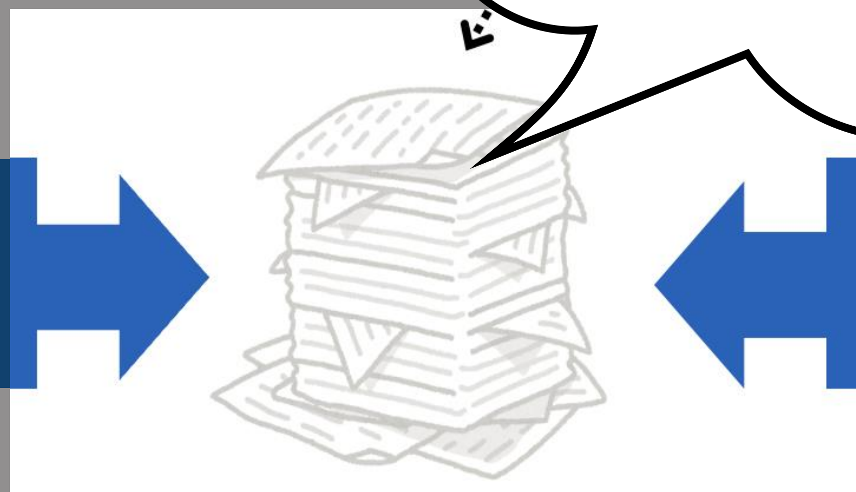
*In this paper, matrices are denoted as follows; A = the adjacency matrix, X = features of the current subset of vertices, Y = feature of all vertices, and Z = updated features of the current subset of vertices. Full details-> See appendix

Current Framework Lim

Traditional GNN separates Message Passing p
... **forcing applications to generate inte**



Step 1: Message Generation (SDDMM)



Step 2: Message Aggregation (SpMM)

$$\mathbf{H} = (\mathbf{X} \times \mathbf{Y}^T) \bigodot \mathbf{A}.$$

$$\mathbf{H}$$

$$\mathbf{Z} = \mathbf{H} \times \mathbf{Y}$$

*In this paper, matrices are denoted as follows; A = the adjacency matrix, X = features of the current subset of vertices, Y = feature of all vertices, and Z = updated features of the current subset of vertices. Full details-> See appendix

Introducing **FusedMM** as a Solution

A memory efficient message passing operation, that has a generalized formula to fit different problems.

SDDMM + SpMM	**FusedMM**

$$\mathbf{H} = (\mathbf{X} \times \mathbf{Y}^T) \odot \mathbf{A}.$$

$$\mathbf{Z} = \mathbf{H} \times \mathbf{Y}$$

No Intermediate Matrix!

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

$$Memory = O(md + nd + nnz) + \underbrace{O(d \cdot nnz)}_{\text{Memory(H)}}$$

$$Memory = O(md + nd + nnz) + \underbrace{0}_{\text{Memory(H)}}$$

Introducing **FusedMM** as a Solution

The anatomy of FusedMM can be roughly explained in two parts:

1. **Parallelization**
2. **Computation**

Algorithm 1 The FusedMM algorithm

Input: $\mathbf{A}$: the adjacency matrix, $\mathbf{X}$: the dense embedding matrices of dimension $m \times d$, $\mathbf{Y}$: the dense embedding matrices of dimension $n \times d$ **Output:** $\mathbf{Z}$: an $m \times d$ matrix

procedure FUSEDMM($\mathbf{A}, \mathbf{X}, \mathbf{Y}$)
  $\{\mathbf{A}_1, ..., \mathbf{A}_t\} \leftarrow$ PART1D($\mathbf{A}$) ▷ $nnz(\mathbf{A}_i) \approx \frac{1}{t} nnz(\mathbf{A})$
  $\{\mathbf{X}_1, ..., \mathbf{X}_t\} \leftarrow$ PART1D($\mathbf{X}$) ▷ nrow($\mathbf{X}_i$)=nrow($\mathbf{A}_i$)
  for $i \in 1..t$ **in parallel do** ▷ Thread parallel
    for each row u of $\mathbf{A}_i$ **do** ▷ Iterate over rows
      $\mathbf{x}_u \leftarrow \mathbf{X}_i[u,:]$ $\mathbf{a}_u \leftarrow \mathbf{A}_i[u,:]$
      $\mathbf{z}_u \leftarrow$ UPDATEU($\mathbf{a}_u, \mathbf{x}_u, \mathbf{Y}$)
  return $\mathbf{Z}$
procedure UPDATEU($\mathbf{a}_u, \mathbf{x}_u, \mathbf{Y}$) ▷ Message generation and aggregation for the vertex u
  $\mathbf{z}_u \leftarrow 0$
  for each v with $\mathbf{a}_{uv} \neq 0$ **do**
    $\mathbf{y}_v \leftarrow \mathbf{Y}[v,:]$
    $\mathbf{z} \leftarrow$ VOP($\mathbf{x}_u$, $\mathbf{y}_v$)
    $s \leftarrow$ ROP($\mathbf{z}$)
    $\mathbf{h} \leftarrow$ SOP(s or $\mathbf{z}$) ▷ directly use $\mathbf{z}$ if ROP is a NOOP, otherwise use s
    $\mathbf{w} \leftarrow$ MOP($\mathbf{h}, \mathbf{y}_v$)
    $\mathbf{z}_u \leftarrow$ AOP($\mathbf{z}_u, \mathbf{w}$)
  return $\mathbf{z}_u$

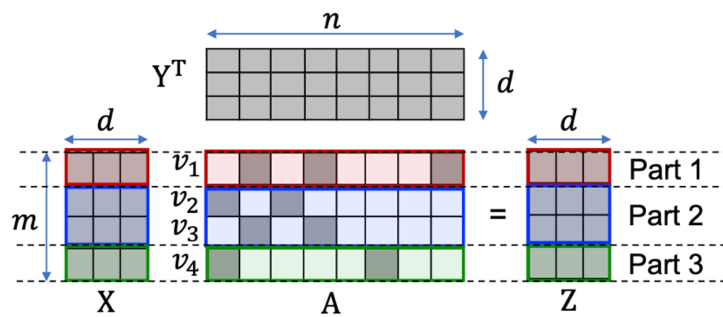
---

1. Parallelization

FusedMM uses thread-level parallelism based on 1D partitioning.

Work is split by vertices & balanced by nnz.
One thread owns z_u, **so it's sync-free.**

Maximized memory-bandwidth efficiency



procedure FUSEDMM($\mathbf{A}, \mathbf{X}, \mathbf{Y}$)
  $\{\mathbf{A}_1, ..., \mathbf{A}_t\} \leftarrow$ PART1D($\mathbf{A}$) $\triangleright$ $nnz(\mathbf{A}_i) \approx \frac{1}{t} nnz(\mathbf{A})$
  $\{\mathbf{X}_1, ..., \mathbf{X}_t\} \leftarrow$ PART1D($\mathbf{X}$) $\triangleright$ nrow($\mathbf{X}_i$)=nrow($\mathbf{A}_i$)
  for $i \in 1..t$ **in parallel do** $\triangleright$ Thread parallel
    for each row u of $\mathbf{A}_i$ **do** $\triangleright$ Iterate over rows
      $\mathbf{x}_u \leftarrow \mathbf{X}_i[u,:]$ $\mathbf{a}_u \leftarrow \mathbf{A}_i[u,:]$
      $\mathbf{z}_u \leftarrow$ UPDATEU($\mathbf{a}_u, \mathbf{x}_u, \mathbf{Y}$)
  return $\mathbf{Z}$

2. Computation

UpdateU: The core procedure of FusedMM

1 **Parallelization**

2 **Computation**

Algorithm 1 The FusedMM algorithm

Input: $\mathbf{A}$: the adjacency matrix, $\mathbf{X}$: the dense embedding matrices of dimension $m \times d$, $\mathbf{Y}$: the dense embedding matrices of dimension $n \times d$ **Output:** $\mathbf{Z}$: an $m \times d$ matrix

procedure FUSEDMM($\mathbf{A}, \mathbf{X}, \mathbf{Y}$)
    $\{\mathbf{A}_1, ..., \mathbf{A}_t\} \leftarrow$ PART1D($\mathbf{A}$) ▷ $nnz(\mathbf{A}_i) \approx \frac{1}{t} nnz(\mathbf{A})$
    $\{\mathbf{X}_1, ..., \mathbf{X}_t\} \leftarrow$ PART1D($\mathbf{X}$) ▷ nrow($\mathbf{X}_i$)=nrow($\mathbf{A}_i$)
    for $i \in 1..t$ **in parallel do** ▷ Thread parallel
        for each row u of $\mathbf{A}_i$ **do** ▷ Iterate over rows
            $\mathbf{x}_u \leftarrow \mathbf{X}_i[u, :]$ $\mathbf{a}_u \leftarrow \mathbf{A}_i[u, :]$
            $\mathbf{z}_u \leftarrow$ UPDATEU($\mathbf{a}_u, \mathbf{x}_u, \mathbf{Y}$)
    return $\mathbf{Z}$
procedure UPDATEU($\mathbf{a}_u, \mathbf{x}_u, \mathbf{Y}$) ▷ Message generation and aggregation for the vertex u
    $\mathbf{z}_u \leftarrow 0$
    for each v with $\mathbf{a}_{uv} \neq 0$ **do**
        $\mathbf{y}_v \leftarrow \mathbf{Y}[v, :]$
        $\mathbf{z} \leftarrow$ VOP($\mathbf{x}_u$, $\mathbf{y}_v$)
        $s \leftarrow$ ROP($\mathbf{z}$)
        $\mathbf{h} \leftarrow$ SOP(s or $\mathbf{z}$) ▷ directly use $\mathbf{z}$ if ROP is a NOOP, otherwise use s
        $\mathbf{w} \leftarrow$ MOP($\mathbf{h}, \mathbf{y}_v$)
        $\mathbf{z}_u \leftarrow$ AOP($\mathbf{z}_u, \mathbf{w}$)
    return $\mathbf{z}_u$

The Core Computations of UpdateU

The whole computation in UpdateU is decomposed into **5 steps:**

1. VOP: Vector Operation
2. ROP: Reduction Operation
3. SOP: Scaling Operation
4. MOP: Multiplication Operation
5. AOP: Accumulation Operation

All steps are Level-1 BLAS and SIMD-friendly

...**Works Well on CPUs!**

The Core Computations of UpdateU

Many applications can be expressed by different combinations of VOP ~ AOP operations.

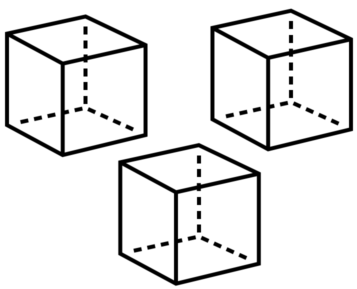
FusedMM is a very flexible operation!

Application	VOP	ROP	SOP	MOP	AOP
Graph Layout	ADD	NORM[2]	SCAL	MUL	ASUM
Node embedding	MUL	RSUM	SIGMOID	MUL	ASUM
Graph Convolution Network	SEL2ND	NOOP	NOOP	MUL	ASUM
Graph Neural Network with MLP	MLP[1]	NOOP	SIGMOID	MUL	AMAX

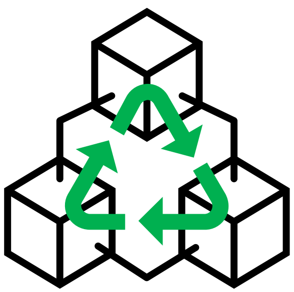
Additional Optimization of FusedMM

Optimizing the whole kernel by feeding the output of one operation directly to the next operation without storing the results.

Process feature dimensions in small blocks



Reuse loaded data across multiple operations



Additional Optimization of FusedMM for sigmoid-based graph embedding

Steps of UpdateU with SIMD optimization

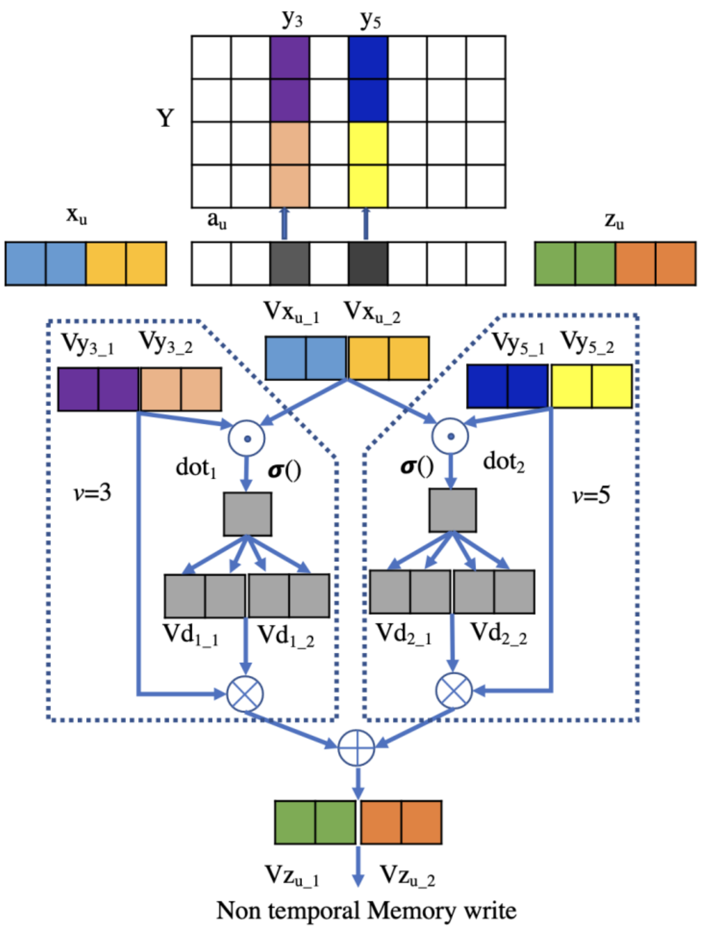
Load feature blocks

Load blocks of x_u into SIMD registers (V).

For each neighbor loop:

1. Load y_v and Compute dot product.
2. Apply sigmoid and Broadcast the result.
3. Multiply and Accumulate.

Store Z_u after the loop

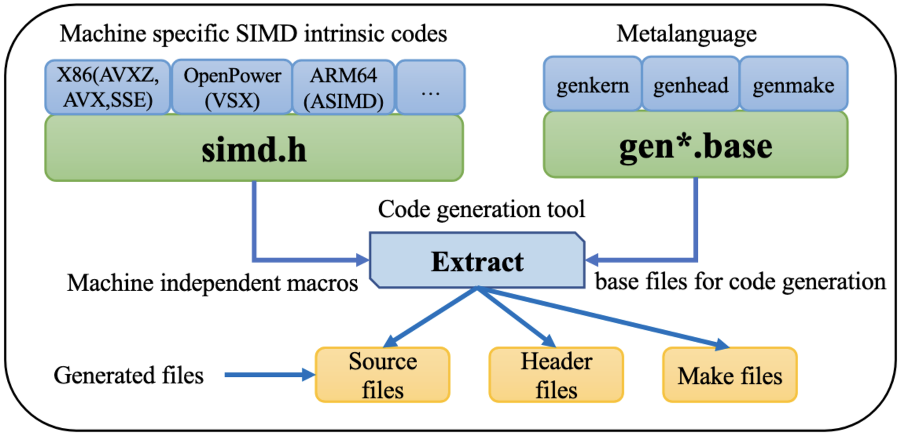


Optimizing FusedMM with Code Generation Tool

Optimizing by hand is a lot of work because there are many patterns with 5 steps, also different hardware architectures.

Code Generation Tool: *Extract*

Provide a common macro interface that hides architecture-specific details.



Use metalanguage to generate templates for different architectures.

Follows Automatically Tuned Linear Algebra Software (**ATLAS**) approach!

Experimental Results

1. Kernel time performance
2. Sensitivity Analysis
3. Application-Level Speedup

1. Kernel time performance on Intel

FusedMM (with SIMD optimization) is up to 34x faster than equivalent DGL kernels.
Speedup is achieved with FusedMM without optimization as well.

Performance on Intel Server

Graphs	Methods	Graph Embedding					FR model					GCN				
		Dimensions (d)					Dimensions (d)					Dimensions (d)				
		32	64	128	256	512	32	64	128	256	512	32	64	128	256	512
Ogbprot.	DGL	0.766	1.394	3.275	8.077	18.236	2.547	4.915	11.115	23.320	×	0.859	1.644	3.71	8.681	×
	FusedMM	0.506	0.859	1.648	3.016	5.703	0.510	0.892	1.737	3.124	5.921	0.343	0.498	0.872	1.442	2.579
	FusedMMopt	0.226	0.247	0.345	0.775	1.358	0.222	0.249	0.323	0.730	1.409	0.114	0.122	0.166	0.449	0.74
	Speedup	3.385	5.655	9.488	10.428	13.433	11.487	19.737	34.389	31.947	-	7.535	13.475	22.349	19.334	-
Youtube	DGL	0.112	0.234	0.493	1.121	2.628	0.192	0.340	[illegible]	1.335	3.007	0.091	0.168	0.338	0.765	1.798
	FusedMM	0.033	0.055	0.090	0.161	0.296	0.032	0.049	0.099	0.165	0.306	0.026	0.037	0.061	0.119	0.226
	FusedMMopt	0.026	0.032	0.058	0.123	0.226	0.024	0.033	0.057	0.121	0.231	0.019	0.035	0.061	0.106	0.164
	Speedup	4.255	7.258	8.463	9.080	11.647	7.899	10.290	11.174	11.007	13.04	4.789	4.800	5.541	7.217	10.963
Orkut	DGL	1.760	3.336	6.851	15.734	34.014	4.044	7.682	14.098	×	×	1.045	1.922	3.993	8.137	×
	FusedMM	0.969	1.601	3.247	5.441	9.665	0.993	1.662	3.352	5.975	9.758	0.746	1.076	2.077	3.71	6.083
	FusedMMopt	0.346	0.523	0.951	3.117	4.961	0.327	0.506	0.978	3.036	5.369	0.15	0.241	0.451	1.462	2.543
	Speedup	5.089	6.381	7.202	5.048	6.856	12.372	15.192	14.414	-	-	6.967	7.975	8.854	5.566	-

*Kernel time (in sec.).

1. Kernel time performance on Intel

Speedup increases with higher feature dimension.

Performance on Intel Server

Graphs	Methods	Graph Embedding Dimensions (d)					FR model Dimensions (d)					GCN Dimensions (d)				
		32	64	128	256	512	32	64	128	256	512	32	64	128	256	512
Ogbprot.	DGL	0.766	1.394	3.275	8.077	18.236	2.547	4.915	11.115	23.320	×	0.859	1.644	3.71	8.681	×
	FusedMM	0.506	0.859	1.648	3.016	5.703	0.510	0.892	1.737	3.124	5.921	0.343	0.498	0.872	1.442	2.579
	FusedMMopt	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.222	0.249	0.323	0.730	1.409	0.114	0.122	0.166	0.449	0.74
	Speedup	3.385	5.655	9.488	10.428	13.433	[illegible]1.487	19.737	34.389	31.947	-	7.535	13.475	22.349	19.334	-
Youtube	DGL	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.192	0.340	0.638	1.335	3.007	0.091	0.168	0.338	0.765	1.798
	FusedMM	0.033	[illegible]	[illegible]	[illegible]	[illegible]	0.032	0.049	0.099	0.165	0.306	0.026	0.037	0.061	0.119	0.226
	FusedMMopt	0.026	0.032	0.058	0.123	0.226	0.024	0.033	0.057	0.121	0.231	0.019	0.035	0.061	0.106	0.164
	Speedup	4.255	7.258	8.463	9.080	11.647	7.899	10.290	11.174	11.007	13.04	4.789	4.800	5.541	7.217	10.963
Orkut	DGL	1.760	3.336	6.851	15.734	34.014	4.044	7.682	14.098	×	×	1.045	1.922	3.993	8.137	×
	FusedMM	0.969	1.601	3.247	5.441	9.665	0.993	1.662	3.352	5.975	9.758	0.746	1.076	2.077	3.71	6.083
	FusedMMopt	0.346	0.523	0.951	3.117	4.961	0.327	0.506	0.978	3.036	5.369	0.15	0.241	0.451	1.462	2.543
	Speedup	5.089	6.381	7.202	5.048	6.856	12.372	15.192	14.414	-	-	6.967	7.975	8.854	5.566	-

*Kernel time (in sec.)

1. Kernel time performance on Intel

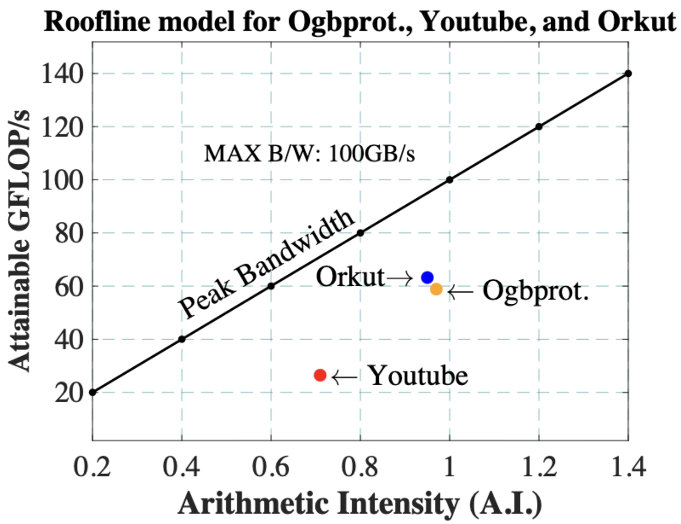
FusedMM (with/without SIMD optimization) did not face the out-of-memory issue like DGL during the experiment.

Performance on Intel Server

Graphs	Methods	Graph Embedding Dimensions (d) 32	64	128	256	512	FR model Dimensions (d) 32	64	128	256	512	GCN Dimensions (d) 32	64	128	256	512
Ogbprot.	DGL	0.766	1.394	3.275	8.077	18.236	2.547	4.915	11.115	23.320	×	0.859	1.644	3.71	8.681	×
	FusedMM	0.506	0.859	1.648	3.016	5.703	0.510	0.892	1.737	3.124	3.921	0.343	0.498	0.872	1.442	2.379
	FusedMMopt	0.226	0.247	0.345	0.775	1.358	0.222	0.249	0.323	0.730	1.409	0.114	0.122	0.166	0.449	0.74
	Speedup	3.385	5.655	9.488	10.428	13.433	11.487	19.737	34.389	31.947	-	7.535	13.475	22.349	19.334	-
Youtube	DGL	0.112	0.234	0.493	1.121	2.628	0.192	0.340	0.638	1.335	3.007	0.091	0.168	0.338	0.765	1.798
	FusedMM	0.033	0.055	0.090	0.161	0.296	0.032	0.049	0.099	0.165	0.306	0.026	0.037	0.061	0.119	0.226
	FusedMMopt	0.026	0.032	0.058	0.123	0.226	0.024	0.033	0.057	0.121	0.231	0.019	0.035	0.061	0.106	0.164
	Speedup	4.255	7.258	8.463	9.080	11.647	7.899	10.290	11.174	11.007	13.04	4.789	4.800	5.541	7.217	10.963
Orkut	DGL	1.760	3.336	6.851	15.734	34.014	4.044	7.682	14.098	×	×	1.045	1.922	3.993	8.137	×
	FusedMM	0.969	1.601	3.247	5.441	9.665	0.993	1.662	3.352	3.973	9.738	0.746	1.076	2.077	3.71	6.083
	FusedMMopt	0.346	0.523	0.951	3.117	4.961	0.327	0.506	0.978	3.036	5.369	0.15	0.241	0.451	1.462	2.543
	Speedup	5.089	6.381	7.202	5.048	6.856	12.372	15.192	14.414	-	-	6.967	7.975	8.854	5.566	-

*Kernel time (in sec.)

1. Kernel time performance on Intel: **Roofline Analysis**



**on Intel server for graph embedding*

STREAM bandwidth on Intel server = 100 GB/s

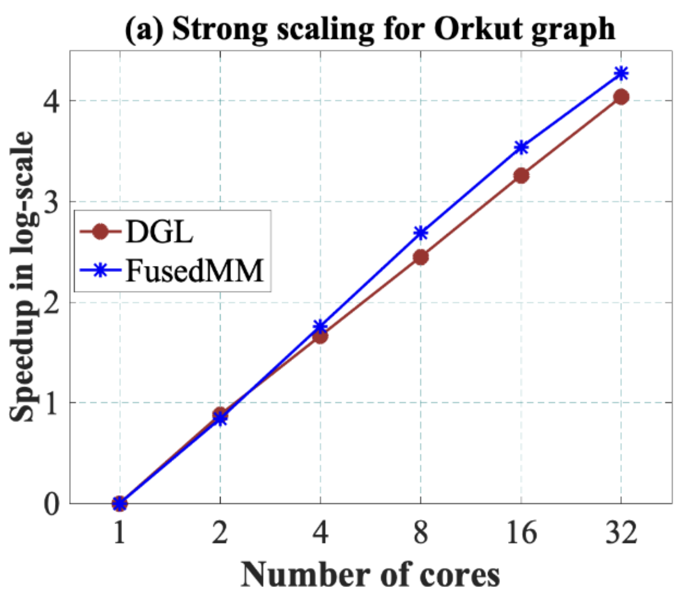
FusedMM ≈ 63 GFLOP/s（Orkut）

$Pmax$ ≈ 95 GFLOP/s

approx. 66% of the bandwidth roof.

2. Sensitivity Analysis

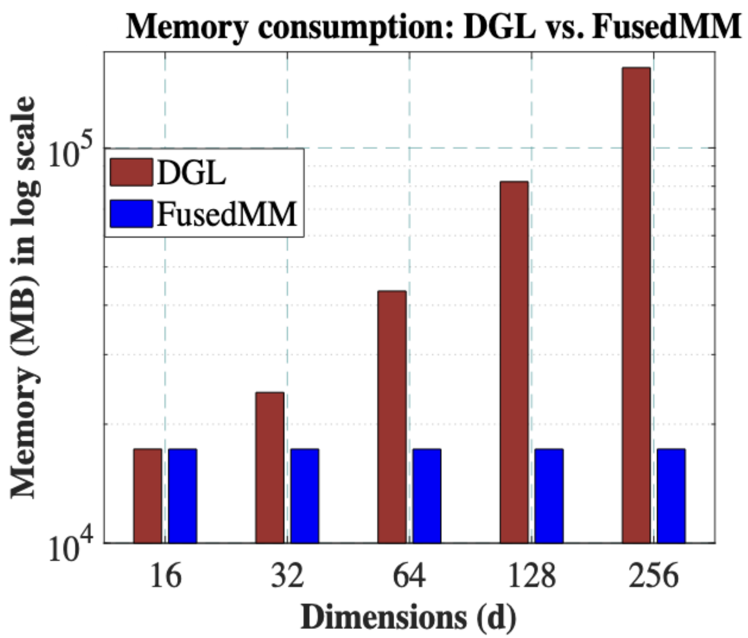
FusedMM on 32 cores is ~20x faster than its sequential runtime.
Consistently faster than DGL at all thread counts.



* Graph Embedding using Orkut graph (d = 256)

2. Sensitivity Analysis

Memory requirement of DGL grows linearly with d while the memory consumption of FusedMM remains stable.

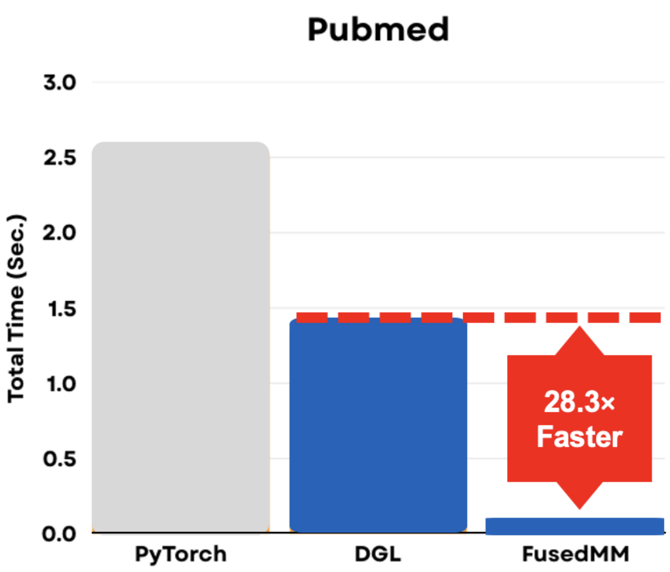


*the FR model for Ogbprot (in megabytes)

3. Application-Level Speedup

A complete AI training session is accelerated by 28x.
Kernel-level optimization translates directly to application-level speedup.

Comparison with PyTorch, DGL and FusedMM



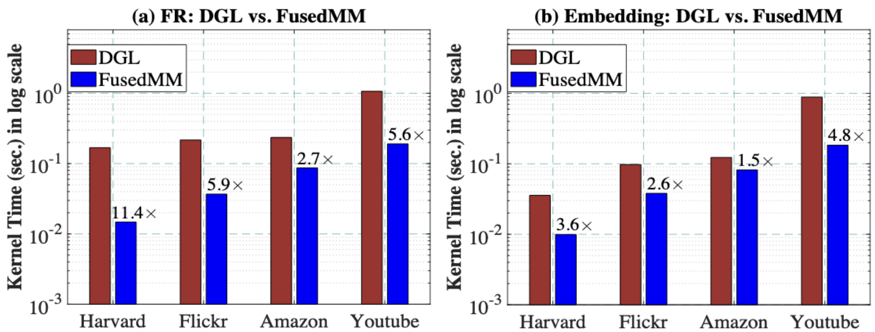
Graphs	Method	Total Time (Sec.)	Speedup
Cora	PyTorch	0.342	48.9×
	DGL	0.177	25.3×
	FusedMM	**0.007**	1.0×
Pubmed	PyTorch	2.590	45.4×
	DGL	1.415	$\boxed{28.3\times}$
	FusedMM	**0.057**	1.0×

**Graph Embedding application time, d=128, batch size = 256*

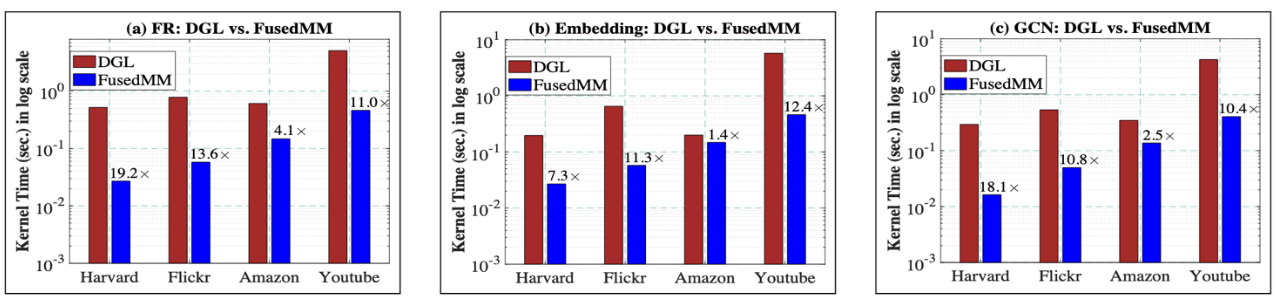
Performance on servers other than Intel

FusedMM performs equally well on Intel, AMD, and ARM processors.
Speedup is up to 19.2× on AMD, up to 11.4× on ARM.

Kernel time on AMD server (d = 128)



Kernel time on ARM server (d = 128)



What is achieved and What’s more?

Achieved

- Reduced memory traffic
- Dramatic kernel time speedup
- Good performance on various servers (Intel, AMD, and ARM)

Next Possibilities

- **GPU implementation**
  - SIMD → SIMT (warp as vector)
- **Tensor Cores**
  - Limited benefit

Appendix

List of notations used in the paper

TABLE I: List of notations used in the paper

Symbol	Description
$\mathbf{A}$	A sparse matrix with dimension: $m \times n$
m	The number of rows in $\mathbf{A}$
n	The number of columns in $\mathbf{A}$
$nnz(\mathbf{A})$	The number of non-zero elements in $\mathbf{A}$
d	The dimension of embedding
$\mathbf{X}$	A dense input matrix with dimension: $m \times d$
$\mathbf{Y}$	A dense input matrix with dimension: $n \times d$
$\mathbf{Z}$	A dense output matrix with dimension: $m \times d$
$\mathbf{A} \times \mathbf{B}$	Matrix-matrix multiplication
$\mathbf{A} \odot \mathbf{B}$	Element-wise multiplication
$\mathbf{a}_{uv} = \mathbf{A}[u, v]$	features of the edge (u, v)
$\mathbf{x}_u = \mathbf{X}[u, :]$	d-dimensional feature vector of vertex u
$\mathbf{a}_u = \mathbf{A}[u, :]$	uth row of the adjacency matrix storing edges adjacent to u

Experimental Setup

Baseline:
- DGL (version 0.5.2)
- PyTorch (version 1.5.1)

Hardware Configurations

	Property	Intel Skylake 8160	AMD EPYC 7551	ARM ThunderX CN8890
Core	Clock	2.10 GHz	2 GHz	1.9 GHz
	L1 cache	32KB	32KB	32KB
	L2 cache	1MB	512KB	×
	LLC	32MB	8MB	16MB
Node	Sockets	2	2	1
	Cores/soc.	24	32	48
	Memory	256GB	128GB	64GB
Env.	Compiler	gcc 10.1.0	gcc 5.4.0	gcc 7.5.0
	Flags	O3, mavx512f, mavx512dq	O3, mavx, mfma	O3, asimd, armv8-a

Datasets

Graphs	#Vertices	#Edges	Avg. Degree	Max. Degree
Cora	2708	5278	3.90	168
Harvard	15126	824617	109.03	1183
Pubmed	19717	44324	4.49	171
Flickr	89250	449878	10.08	5425
Ogbprot.	132534	39561252	597	7750
Amazon	334863	925872	5.59	549
Youtube	1138499	2990443	5.25	28754
Orkut	3072441	117185083	76.28	33313

Limitations and Trade-offs of FusedMM

Limitations

Less effective if:

- Messages must be reused
- Benefits decrease if messages are reused multiple times

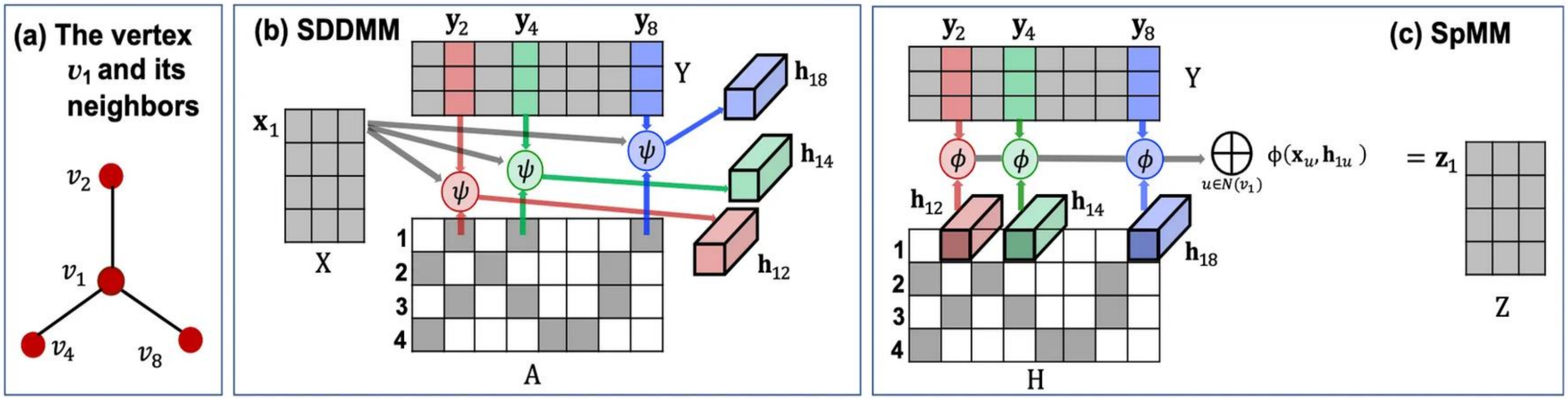
Best for memory-bound sparse workloads, single-pass message generation + aggregation.

Trade-Offs

Reduced optimization freedom

- no separate tuning of SDDMM / SpMM
- fixed execution order, 1D partitioning only

Current Framework Limitations



- x1 denotes the feature vector of v1.
- y2, y4, and y8 denote feature vectors of v1's neighbors v2, v4, and v8.
- An SDDMM is used to generate messages h12, h14, and h18 for the edges adjacent to v1.
- The messages are aggregated using an SpMM operation that generates the updated vector z1 for v1.

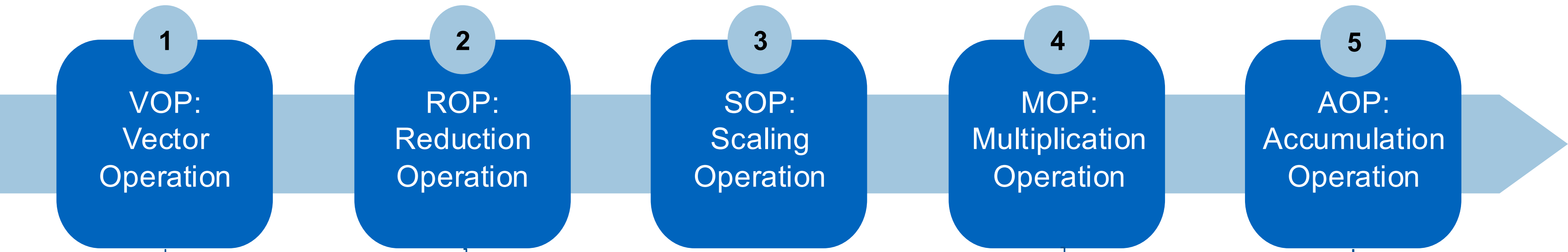
The Core Computations of UpdateU

To remain flexible for diverse applications, the whole computation in UpdateU is splitted into **5 steps:**

1. **VOP: Vector Operation** — takes two vectors of the same length, multiplies them and outputs a new vector of the same size.
2. **ROP: Reduction Operation** — reduces a vector to a scalar.
3. **SOP: Scaling Operation** — scales a vector/scalar using a linear or nonlinear function.
4. **MOP: Multiplication Operation** — multiplies a vector element-wise by another vector or scales it by a scalar, producing a vector of the same length.
5. **AOP: Accumulation Operation** — accumulates vectors by combining each incoming message vector with the current result

The Core Computations of UpdateU

The whole computation in UpdateU is decomposed into **5 steps:**



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

Experimental Results: **Comparison w/ Intel MKL SpMM**

Despite being a multipurpose kernel, FusedMM can match the vendor-optimized SpMM.

Graphs	Method	Single Thread			48 Threads (2 soc.)		
		64	128	256	64	128	256
Ogbprot.	MKL	1.017	2.310	5.318	0.034	0.094	**0.264**
	FusedMM	**0.951**	**1.990**	**4.125**	**0.031**	**0.075**	0.336
Youtube	MKL	0.142	0.310	0.606	**0.012**	0.031	**0.071**
	FusedMM	**0.132**	**0.261**	**0.524**	0.015	**0.028**	0.082
Orkut	MKL	6.336	14.356	29.348	**0.380**	0.852	**1.961**
	FusedMM	**5.876**	**11.897**	**23.292**	0.389	**0.828**	2.775

*Kernel time (in sec.) of SpMM on Intel server for various dimensions. Best value is marked in bold.