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

Md. Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad
Luddy School of Informatics, Computing, and Engineering
Indiana University Bloomington, IN, USA
Email: {morahma, msujon, azad}@iu.edu

***Abstract*—We develop a fused matrix multiplication kernel that unifies sampled dense-dense matrix multiplication and sparse-dense matrix multiplication under a single operation called FusedMM. By using user-defined functions, FusedMM can capture almost all computational patterns needed by popular graph embedding and GNN approaches.**

FusedMM is an order of magnitude faster than its equivalent kernels in Deep Graph Library. The superior performance of FusedMM comes from the low-level vectorized kernels, a suitable load balancing scheme and an efficient utilization of the memory bandwidth. FusedMM can tune its performance using a code generator and perform equally well on Intel, AMD and ARM processors. FusedMM speeds up an end-to-end graph embedding algorithm by up to $28\times$ on different processors. The source code is available at https://github.com/HipGraph/FusedMM.

***Index Terms*—message passing, GNN, graph embedding**

I. Introduction

Message passing is a powerful paradigm for designing various graph analytics and learning algorithms. Given a graph $G(V, E)$ with $\mathbf{x}_u$ denoting node attributes for $u{\in}V$ and $\mathbf{a}_{uv}$ denoting edge attributes for $(u, v){\in}E$, a message passing system typically operates in two phases: (a) a message $\mathbf{h}_{uv}$ is generated on each edge $(u, v){\in}E$ using a function $\psi(\mathbf{x}_u, \mathbf{x}_v, \mathbf{a}_{uv})$ and (b) messages are aggregated at $u{\in}V$ using a function $\bigoplus_{v\in N(u)} \phi(\mathbf{x}_u, \mathbf{x}_v, \mathbf{h}_{uv})$. Here, $N(u)$ denotes in-neighbors of u, ψ and ϕ are application-defined functions and $\bigoplus$ is an application-defined aggregator. By changing the functions ψ, ϕ, and $\bigoplus$, we can easily derive force-directed graph layout [1, 2], graph embedding [3, 4], graph convolutional network (GCN) [5], and graph neural networks (GNNs) algorithms [6] as shown in Fig. 1. This flexibility and intrepretability of message passing made it a widely-used paradigm for designing high-level graph learning algorithms.

Even though a message passing API makes high-level graph algorithm easy to describe, high-performance linear algebra kernels are often used under the hood for performance. Conceptually, the message generated on edges can be mapped to a sampled dense-dense matrix multiplication (SDDMM) [7]–[9] and the message aggregation is performed by a sparse-dense matrix multiplication (SpMM). For example, GNN frameworks such as PyTorch geometric (PyG) [10] and Deep Graph Library (DGL) [11] rely on SDDMM and SpMM to implement their high-level message passing API. When standard addition and multiplication operations can capture GNN's internal operations, PyG and DGL rely on vendor-provided sparse libraries (e.g., MKL for Intel CPU, cuSPARSE for GPU) that offer highly optimized implementations. However, to capture complex and diverse operations such as those shown in Fig. 1, PyG and DGL also support user-defined operations using general SDDMM and SpMM kernels.

In almost all applications (except in attention-based GNNs [12]), messages generated on edges are immediately aggregated on vertices. Computationally, it means that an SDDMM is almost always followed by an SpMM operation. Existing libraries such as DGL provide separate SDDMM and SpMM kernels, forcing applications to generate intermediate outputs from SDDMM. This can incur significant computational and memory bottlenecks, especially when each edge generates high-dimensional messages. To address this problem, we develop a unified kernel called *FusedMM* that captures the overall computation offered by SDDMM and SpMM. Conceptually, the fused kernel generates and aggregates messages collectively without explicitly storing messages. Thus, the updated feature $\mathbf{z}_u$ of vertex u is generated in one shot as follows:

$$\mathbf{z}_u = \bigoplus_{v\in N(u)} \phi(\mathbf{x}_u, \mathbf{x}_v, \psi(\mathbf{x}_u, \mathbf{x}_v, \mathbf{a}_{uv})). \tag{1}$$

We develop efficient parallel algorithms for the FusedMM kernel that computes Eq. 1 for every vertex. We show that this single kernel with the support of user-defined operations can capture almost all computational patterns arisen in graph layout, graph embedding and GNN algorithms. We make FusedMM general purpose by dividing it into five steps where each step performs vectorized operations with user-supplied functions. We integrated FusedMM with DGL and show that even a naive implementation of FusedMM is always faster than DGL's SDDMM and SpMM operations on Intel, AMD and ARM processors.

For sparse graphs, FusedMM (as well as SDDMM and SpMM) is expected to be bound by the memory bandwidth. Keeping this in mind, we developed a multithreaded algorithm that minimizes data movements from the main memory and utilizes cache and registers as much as possible. To achieve best performance, we developed a library with code generator and tuned the factor of the register blocking after applying different strategies. Based on these tuned parameters, we automatically generated SIMD vectorized kernels with best register blocking supported on different SIMD architecture (e.g., AVX512/AVX in X86 and ASIMD/NEON in ARM).

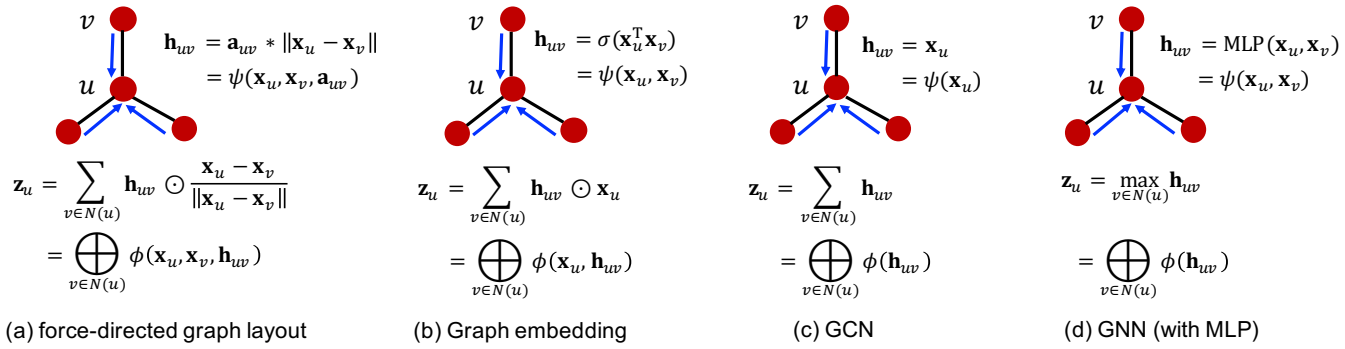
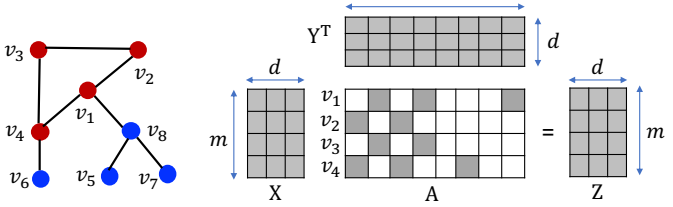


Fig. 1: Message passing models in (a) force-directed graph drawing, (b) graph embedding (e.g., in VERSE [3]), (c) graph convolutional network (GCN), and (d) general graph neural networks with multilayer perceptron (MLP). In all cases, the message $\mathbf{h}_{uv}$ that passes from v to u is a function of the form $\psi(\mathbf{x}_u, \mathbf{x}_v, \mathbf{a}_{uv})$, where $\mathbf{x}_u$ and $\mathbf{x}_v$ are node features and $\mathbf{a}_{uv}$ is the feature of the edge (u, v). The messages are aggregated at the target vertex u using an operation of the form $\bigoplus_{v \in N(u)} \phi(\mathbf{x}_u, \mathbf{x}_v, \mathbf{h}_{uv})$. The functions $\bigoplus$, ϕ, and ψ change based on the high-level algorithm. The entire process of message generation and aggregation can be mapped to the flexible FusedMM operation developed in this paper.



(a) A graph (b) Message generation & aggregation using matrices

Fig. 2: (a) An example graph. We consider message aggregations on a subgraph induced by $\{v_1, v_2, v_3, v_4\}$ (shown in red). (b) $\mathbf{A}$ denotes the adjacency matrix of the induced subgraph, $\mathbf{X}$ denotes features of $\{v_1, v_2, v_3, v_4\}$ and $\mathbf{Y}$ denotes features of all vertices. The updated features of $\{v_1, v_2, v_3, v_4\}$ are stored in $\mathbf{Z}$.

The optimized FusedMM is an order of magnitude faster than its equivalent kernels in DGL on Intel, AMD and ARM processors. FusedMM speeds up end-to-end graph embedding algorithms by up to 28×. The main contributions of the paper are summarized below.

1) We introduce FusedMM, a general-purpose kernel for various graph embedding and GNN operations.
2) FusedMM requires less memory and utilizes memory bandwidth efficiently by fusing SDDMM and SpMM operations.
3) FusedMM employs autotuned vectorized operations that run up to 34× faster than equivalent kernels in DGL. FusedMM performs equally well on Intel, AMD, and ARM processors.
4) FusedMM expedites end-to-end training of a graph embedding algorithm by 28× relative to DGL.

II. Linear-algebraic Kernels in Graph Learning

Notations. We use uppercase boldfaced letters to denote matrices. $\mathbf{A}$ denotes the adjacency matrix, $\mathbf{X}$ denotes features of the current subset of vertices, $\mathbf{Y}$ denotes feature of all vertices, and $\mathbf{Z}$ denotes updated features of the current subset of vertices. We use lowercase boldfaced letters to denote vectors. The uth row of $\mathbf{A}$, $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ are denoted by $\mathbf{a}_u$, $\mathbf{x}_u$, $\mathbf{y}_u$, and $\mathbf{z}_u$, respectively. Hence, $\mathbf{x}_u = \mathbf{X}[u, :]$ represents the feature vector of the vertex u. The feature of the edge (u, v) is denoted by $\mathbf{a}_{uv} = \mathbf{A}[u, v]$. Table I summarizes our notations.

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

The problem setting. Let $G(V, E)$ denote a graph with a set of n vertices V and a set of edges E. In most practical settings, an induced subgraph (e.g., a minibatch of vertices) is considered at a given point. For example, vertices $\{v_1, v_2, v_3, v_4\}$ and their adjacent edges form a minibatch in Fig. 2(a). We consider developing a linear-algebra kernel that can capture both message generation and aggregation for all vertices in a given subgraph (also covers the case for the entire graph).

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ denote the sparse adjacency matrix of the given subgraph of m vertices where $\mathbf{a}_{uv} \neq 0$ if $(u, v) \in E$, otherwise $\mathbf{a}_{uv} = 0$. Here, the rectangular matrix $\mathbf{A}$ represents a slice of the original adjacency matrix and thereby captures a minibatch of vertices needed in GNN training. Similarly, $\mathbf{A}$ can also represent a bipartite graph with different number of vertices in its two parts. $\mathbf{a}_{uv}$ can be either Boolean (unweighted graphs) or a user-defined data type depending on edge features.

Let $\mathbf{X} \in \mathbb{R}^{m \times d}$ be the dense matrix storing features of vertices in the current subgraph and $\mathbf{Y} \in \mathbb{R}^{n \times d}$ be a feature matrix for all vertices. Each row of $\mathbf{X}$ or $\mathbf{Y}$ stores a d-dimensional vertex feature vector. Even though $\mathbf{X}$ can be a submatrix of $\mathbf{Y}$ in most practical applications, we separate them for generality. For example, $\mathbf{X}$ and $\mathbf{Y}$ may store different features in heterogeneous and bipartite graphs. The final output of a message passing step is an updated feature matrix $\mathbf{Z} \in \mathbb{R}^{m \times d}$,

where the uth row $\mathbf{z}_u$ stores the updated features of u. We seek a fused matrix multiplication kernel FusedMM that outputs $\mathbf{Z}$ from $\mathbf{A}, \mathbf{X}$, and $\mathbf{Y}$ as follows $\mathbf{Z}$=FusedMM$(\mathbf{A}, \mathbf{X}, \mathbf{Y})$. Fig. 2(b) shows the linear-algebraic representation.

Even though a FusedMM operation is what most applications need, PyG and DGL map edge-wise computations to an SDDMM kernel and vertex-wise computations to an SpMM kernel. We briefly discuss how these separate kernels are used.

Edge-wise message kernel: SDDMM. As shown in Fig. 1, messages are generated on an edge (u, v) based on features of vertices u and v. When we consider messages generated on all edges corresponding to $\mathbf{A}$, the underlying operation is an SDDMM operation that operates on two dense input matrices guided by a sparse matrix, and produces a sparse matrix $\mathbf{H}$:

$$\text{[SDDMM]} \qquad \mathbf{H} = (\mathbf{X} \times \mathbf{Y}^T) \bigodot \mathbf{A}. \tag{2}$$

Here, $\mathbf{H}$ is an $m \times n$ sparse matrix or tensor with exactly the same sparsity pattern of $\mathbf{A}$, and $\bigodot$ represents element-wise multiplication. Since $\mathbf{A}$ is a sparse matrix, any practical implementation of SDDMM would only compute entries where $\mathbf{A}$ has nonzeros as shown in Fig. 3(b). The actual computation needed to generate $\mathbf{H}[u, v]$=$\mathbf{h}_{uv}$ is application dependent and is represented by the message generation function ψ. Hence, in a general-purpose SDDMM (gSDDMM), we have $\mathbf{h}_{uv}$=$\psi(\mathbf{x}_u, \mathbf{y}_v, \mathbf{a}_{uv})$, where $\mathbf{x}_u$ and $\mathbf{y}_v$ denote the feature vectors of u and v. Notice in Fig. 3(b) that ψ can generate a vector as an output making $\mathbf{H}$ a sparse tensor in some applications. At the end of the gSDDMM operation, $\mathbf{h}_{uv}$ stores the message generated on the edge (u, v).

Vertex-wise message aggregation kernel: SpMM. After generating messages, most applications aggregate them on target vertices. This operation can be captured by an SpMM operation:

$$\text{[SpMM]} \qquad \mathbf{Z} = \mathbf{H} \times \mathbf{Y}, \tag{3}$$

where each row of $\mathbf{Z}$ stores the updated feature vectors of vertices. As with the gSDDMM operation, a generalized SpMM (gSpMM) can take user-defined multiplication operation ϕ and aggregation function as shown in Fig. 3(c). Thus, the uth row of the output can be formed as follows: $\mathbf{z}_u = \bigoplus_{\mathbf{h}_{uv} \neq 0} \phi(\mathbf{y}_v, \mathbf{h}_{uv})$, where $\mathbf{z}_u = \mathbf{Z}[u, :]$ denotes the updated feature vector of the target vertex u.

The need for a fused matrix-multiplication kernel. The first and most obvious reason to develop a unified kernel is to reduce the memory requirement. For example, the intermediate matrix $\mathbf{H}$ storing edge-wise messages can take $O(d * nnz(\mathbf{A}))$ space when the application generates d-dimensional messages on edges. This extra memory required to store $\mathbf{H}$ can make it downright impossible to solve certain problems if we use separate SDDMM and SpMM kernels. Separate kernels also require us to read/write $\mathbf{H}$ and $\mathbf{Y}$ more than once, which negatively impacts the performance of these memory-bound kernels. Finally, a unified kernel enables a joint optimization of message generation and aggregation, offering more flexibility to applications and more opportunity to tune performance on different processors and accelerators. Motivated by these potential benefits in multiple frontiers, we develop FusedMM that offers both flexibility and high performance.

III. A Flexible FusedMM Kernel

Design objectives. The first objective of the FusedMM kernel is to capture the entire message generation and aggregation process for a given subgraph as formulated in Fig. 2(b). The main challenge in designing such a unified kernel is to make it flexible for diverse applications that use various message generation functions ψ, multiplication operations ϕ, and message aggregators $\oplus$. We address this challenge by splitting the whole computation into a sequence of five well-defined steps where each step accepts user-defined functions. The second objective of FusedMM is to efficiently utilize processor resources (registers, cache, memory bandwidth, etc.) to maximize performance. We achieve this objective by fully utilizing SIMD units available in modern processors while maximizing the utilization of the memory bandwidth.

A. The anatomy of FusedMM

We begin with a graph embedding example shown in Fig. 1(b). The message $\mathbf{h}_{uv}$ generated on edge (u, v) is defined by $\sigma(\mathbf{x}_u^T \mathbf{y}_v)$, where σ is the Sigmoid function. As mentioned before, we separate features of source and destination vertices for generality. We split the computation of $\mathbf{h}_{uv}$ into three steps.

[Step 1: VOP] We perform elementwise "multiplication" of $\mathbf{y}$ and $\mathbf{x}$ and return another vector $\mathbf{w}$ of the same length of $\mathbf{y}$ and $\mathbf{x}$. For our graph embedding example, this step computes the first part of the dot product. As this step operates on two vectors, we call it VOP. More specifically, VOP$(\mathbf{x}, \mathbf{y}) = \mathbf{x} \odot \mathbf{y} = \mathbf{z}$, where $\odot$ is an element-wise vector operation. Any user-defined function following this syntax is also allowed.

[Step 2: ROP] The second step potentially reduces a vector to a scalar. For our graph embedding example, this step computes the summation part of the dot product computation. As this step reduces elements of a vector, we call it ROP. More specifically, ROP$(\mathbf{z}) = \oplus_i \mathbf{z}_i = s$, where s is a scalar and $\oplus$ is a reduction operation. VOP and ROP together can compute the dot product between two vectors.

[Step 3: SOP] The third step scales a vector/scalar using a linear or nonlinear function. For our graph embedding example, this step applies the sigmoid function to $\mathbf{x}_u^T \mathbf{y}_v$, hence it is called a scaling operation or SOP. More specifically, SOP$(\mathbf{z}) = \sigma(\mathbf{z}_i); \forall i$, where σ is a linear or nonlinear function. $\mathbf{z}$ can also be scalar such as in computing $\sigma(\mathbf{x}_u^T \mathbf{y}_v)$.

These three steps deliver a flexible message-generation operation ϕ and form the SDDMM phase of FusedMM. Next, we consider the message aggregation phase in graph embedding shown in the Fig. 1(b). The message aggregation is defined by: $\mathbf{z}_u$=$\sum_{v \in N(u)} \mathbf{h}_{uv} \odot \mathbf{y}_v$, where $\mathbf{h}_{uv}$ is the output of SDDMM on the edge (u, v). $\mathbf{h}_{uv}$ is either a scalar or a vector. We split the message aggregation into two steps.

[Step 4: MOP] We perform elementwise "multiplication" of $\mathbf{h}$ and $\mathbf{y}$ and return another vector $\mathbf{w}$ of the same length of $\mathbf{y}$. If either input is a scalar (e.g., the message generated on an edge is scalar), MOP simply scales the other vector with the scalar. For graph embedding where $\mathbf{h}_{uv}$ is a scalar, this operation scales $\mathbf{y}_v$ by $\mathbf{h}_{uv}$. As this step multiplies a vector by a vector or

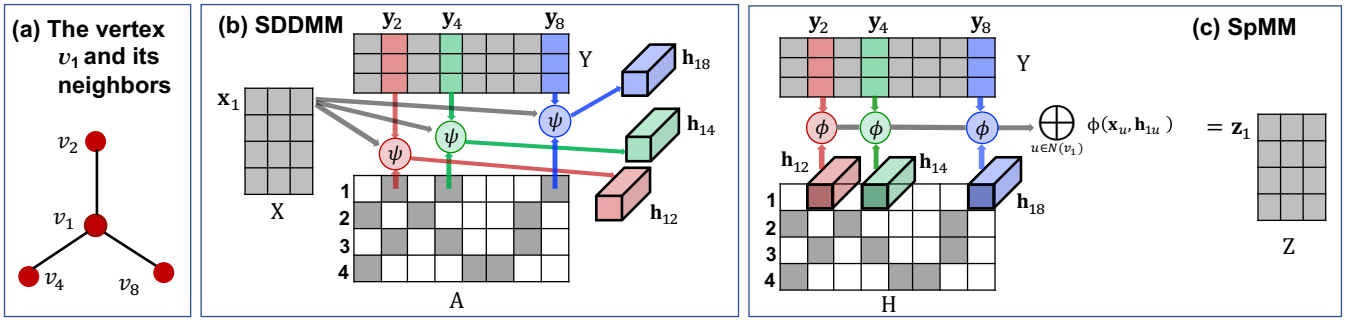


Fig. 3: (a) A vertex v_1 and its neighbors. We would like to generate messages on all edges adjacent to v_1 and then aggregate the messages to get new feature vector for v_1. (b) $\mathbf{x}_1$ denotes the feature vector of v_1. $\mathbf{y}_2$, $\mathbf{y}_4$, and $\mathbf{y}_8$ denote feature vectors of v_1's neighbors v_2, v_4, and v_8, respectively. An SDDMM is used to generate messages $\mathbf{h}_{12}, \mathbf{h}_{14}$, and $\mathbf{h}_{18}$ for the edges adjacent to v_1. Here, $\mathbf{h}_{12}, \mathbf{h}_{14}$, and $\mathbf{h}_{18}$ can be vectors. (c) The messages are aggregated using an SpMM operation that generates the updated vector $\mathbf{z}_1$ for v_1. When SDDMM and SpMM separated as shown here, the messages are explicitly stored in $\mathbf{H}$. This paper combines SDDMM and SpMM into a FusedMM operation.

TABLE II: Standard operations that could be used in five steps of FusedMM. $\mathbf{x}, \mathbf{y}, \mathbf{z}$ are vectors, α is a scalar, σ is a unary function. $\oplus$ and $\odot$ denote "addition" and "multiplication" operations, respectively. In the most general case, $\oplus$ and $\odot$ are defined as part of a semiring provided by the application. [1]Our ASUM is different from the absolute sum of L1 BLAS.

Type	Op	In 1	In 2	Return	Where used
Binary	ADD	$\mathbf{x}$	$\mathbf{y}$	$\mathbf{x} \oplus \mathbf{y}$	VOP
	MUL	$\mathbf{x}$	$\mathbf{y}$	$\mathbf{x} \odot \mathbf{y}$	VOP, MOP
	SEL2ND	$\mathbf{x}$	$\mathbf{y}$	$\mathbf{y}$	VOP, MOP
Unary	SIGMOID	$\mathbf{x}$	σ	$\sigma(\mathbf{x})$	SOP, MOP
	SCAL	$\mathbf{x}$	α	$\alpha\mathbf{x}$	SOP, MOP
Reduction	RSUM	$\mathbf{x}$	-	$\oplus_i \mathbf{x}_i$	ROP
	RMUL	$\mathbf{x}$	-	$\odot_i \mathbf{x}_i$	ROP
Accumulate	ASUM[1]	$\mathbf{z}$	$\mathbf{x}$	$\mathbf{z} \leftarrow \mathbf{z} \oplus \mathbf{x}$	AOP
	AMAX	$\mathbf{z}$	$\mathbf{x}$	$\mathbf{z} \leftarrow \max(\mathbf{z}, \mathbf{x})$	AOP

scalar, we call it MOP. Specifically, $\text{MOP}(\mathbf{h}, \mathbf{y}) = \mathbf{w} = \mathbf{h} \odot \mathbf{y}$, where $\odot$ is an element-wise "multiplication" operation.

[Step 5: AOP] The last step "accumulates" a vector with another vector. For graph embedding, this step accumulates messages received from adjacent vertices; hence it is called an accumulation operation or AOP. Specifically, $\text{AOP}(\mathbf{z}, \mathbf{y}) = \mathbf{z} = \mathbf{z} \oplus \mathbf{y}$, where $\oplus$ is a use-defined "addition" function.

All of these five steps together form the core computations of FusedMM. We define these operation in an abstract sense so that users can use standard or custom-built operations to substitute them. Our library accepts function pointers for each of these five operations to facilitate user-defined functions. An application can skip some of these steps by passing a NOOP.

The aforementioned breakdown serves two key objectives of FusedMM. First, these steps are flexible enough to capture almost all operations in various graph layout, graph embedding and GNN algorithms (see Table III). By changing the operations, users can design their own high-level applications. Second, all five operations are similar to level-1 BLAS operations that can be vectorized using SIMD units.

B. Standard operations used in FusedMM with applications

While designing FusedMM with VOP, ROP, SOP, MOP, and AOP makes a very general kernel, users have to provide definitions of these operations to design a new application. To this end, we develop optimized implementations for a few common operations that users can directly plug into their applications. Table II shows some standard operations that we implemented in our software. We also show where these operations could be used inside FusedMM. For example, binary operations ADD and MUL denote element-wise addition and multiplication operations that could be used to substitute VOP and MOP. The SEL2ND operations simply copies the second operand to the output. Unary operations such as SIGMOID and SCAL are used to scale a vector using non-linear and linear functions. Hence, these operations can be used in place of SOP and MOP in FusedMM. Finally, reduction operations RSUM and RMUL are used in ROP, and accumulation operations ASUM and AMAX are used in AOP. Note that Table II only shows a few examples that can be used in FusedMM. Any user-defined functions are allowed as long as they satisfy the I/O requirements of VOP, ROP, SOP, MOP, and AOP.

Table III shows how we can implement four applications described in Fig. 1. We already discussed the graph embedding application when we defined VOP, ROP, SOP, MOP, and AOP in Section III-A. For GCN as shown in Fig. 1(c), the first VOP operation simply selects neighbor's feature using SEL2ND. Since a vanilla GCN does not perform a reduction on edges, we use NOOP for ROP and SOP in the 2nd row in Table III. The message aggregation in GCN multiplies messages by edge features using MUL for MOP and finally messages are pooled using ASUM. Different variants of GCN use different pooling options [6] such as maximum, minimum, mean, etc. All of these options can be captured by MOP and AOP in FusedMM. The fourth row in Table III shows a simple GNN layer that uses MLP to generate messages. This is an example where a user-defined VOP is needed.

C. The parallel FusedMM algorithm

Using all building blocks of FusedMM discussed thus far, Algorithm 1 describes a multithreaded FusedMM algorithm that takes $\mathbf{A}, \mathbf{X}$, and $\mathbf{Y}$ as inputs and returns updated vertex features $\mathbf{Z}$ as the output. FusedMM does not perform mini-batching, which is done at the application layer.

TABLE III: Using FusedMM to develop four applications. NOOP means no operation needed at that step. [1] Needs a user-provided MLP function. [2] A function that computes vector norms.

Application	Ref	VOP	ROP	SOP	MOP	AOP
Graph Layout (FR model [1])	Fig. 1(a)	ADD	NORM[2]	SCAL	MUL	ASUM
Node embedding (Force2Vec [4] and VERSE [3] with sigmoid)	Fig. 1(b)	MUL	RSUM	SIGMOID	MUL	ASUM
Graph Convolution Network [5]	Fig. 1(c)	SEL2ND	NOOP	NOOP	MUL	ASUM
Graph Neural Network with MLP	Fig. 1(d)	MLP[1]	NOOP	SIGMOID	MUL	AMAX

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

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

Partitioning matrices for thread-level parallelization. Parallel FusedMM in Algorithm 1 starts with load-balanced partitioning of matrices. We partition input matrices with two key objectives in mind: (a) computations in different parts should be independent of each other so that threads can process different partitions in parallel without synchronization and (b) the computational cost for each partition should be approximately equal. If we aim to implement SDDMM and SpMM separately, we could use either 1D (i.e., vertex partitioning) or 2D (edge partitioning) partitioning of the adjacency matrix $\mathbf{A}$. However, when we fuse SDDMM and SpMM in FusedMM, 2D partitioning of $\mathbf{A}$ may be very inefficient or even outright impossible. In our graph embedding example, the message on edge (u, v) is computed using a dot product of node feature vectors and then, a sigmoid function is applied on the output of the dot product. In this case, it is not possible to generate messages from partial vertex features without changing the mathematical interpretation. Even when partial messages make sense, 2D partitioning may be very inefficient because the last step of FusedMM aggregates messages from in neighbours $N(u)$ of u, but all vertices in $N(u)$ may not be in the current partition. Consequently, 2D partitioning will require storing partially computed results, which could be detrimental for performance. Thus, we opt to use 1D partitioning of $\mathbf{A}$.

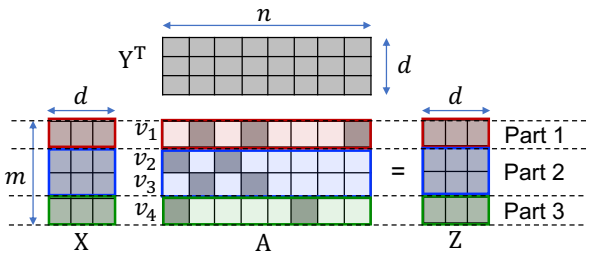


Fig. 4: 1D partitioning of $\mathbf{X}$, $\mathbf{A}$ and $\mathbf{Z}$ for thread-level parallelization. Three partitions are shown in three colors. For example, Algorithm 1 uses $\mathbf{A}_2$ to denote the middle partition (shown in blue) of $\mathbf{A}$.

The PART1D function at line 2 of Algorithm 1 partitions $\mathbf{A}$ into t parts $\mathbf{A}_1, ..., \mathbf{A}_t$, where each part has the same number of columns but different numbers of rows. We use a simple load-balancing scheme where *nnz* of all parts $\mathbf{A}_i$ are approximately equal: $nnz(\mathbf{A}_i) \approx 1/t * nnz(\mathbf{A})$. This partitioning is done by scanning the row pointer array of $\mathbf{A}$ stored in the Compressed Sparse Row format. The complexity of PART1D is $O(m)$, where m is the number of rows in $\mathbf{A}$. $\mathbf{X}$ and $\mathbf{Z}$ are partitioned following the partitions of $\mathbf{A}$ such that nrow($\mathbf{X}_i$)=nrow($\mathbf{A}_i$)=nrows($\mathbf{Z}_i$). The other input matrix $\mathbf{Y}$ is not partitioned. Fig. 4 shows an example of 1D partitioning.

Fused operation within a thread. After the partitioning is performed, the ith group of partition $\{\mathbf{A}_i, \mathbf{X}_i, \mathbf{Y}\}$ are processed by a thread to generate the corresponding output $\mathbf{Z}_i$. Here, threads may perform concurrent reads on $\mathbf{Y}$, but they do not perform any concurrent writes. Hence, threads can proceed in parallel (line 5-7 of Alg. 1) without any synchronization.

Within its private data partitions, a thread processes one vertex (that is one row of $\mathbf{A}_i$) at a time (line 5 of Alg. 1). Assume that u is the current vertex under consideration. Then, the UPDATEU function generates messages for all edges adjacent to u and aggregates the messages to return the updated feature vector $\mathbf{z}_u$. Thus, the UPDATEU function needs the entire $\mathbf{Y}$ and the uth rows of $\mathbf{X}$ and $\mathbf{A}$ to generate $\mathbf{z}_u$. Inside the UPDATEU function (line 9-18 of Alg. 1), we call five building blocks VOP, ROP, SOP, MOP, and AOP to obtain our desired vector $\mathbf{z}_u$. In our library, UPDATEU can take predefined or user-defined functions.

IV. Optimizations and Code Generation

While the general FusedMM algorithm (Alg. 1) provides ample flexibility to applications, it can perform sub-optimally because it stores outputs after each of its five steps. If we recognize a pattern from predefined VOP, ROP, SOP, MOP, and AOP operations, we can optimize the whole kernel by feeding the output of one operation directly to the next operation without storing the results. For example, the second row in

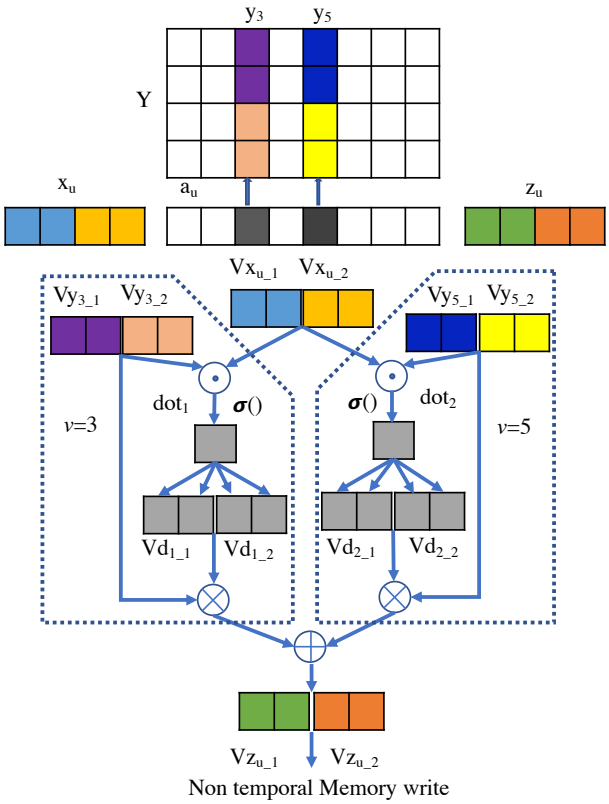


Fig. 5: SIMD vectorized implementation of UPDATEU function for specialized sigmoid based graph embedding. All the labels, starting with 'V', indicate the SIMD registers and they are color-coded with the portion of the vector they access or update.

Table III uses the (MUL, RSUM, SIGMOID, MUL, ASUM) sequence of operations, which is a common pattern used by various node embedding algorithms such as VERSE [3] and Force2Vec [4]. Hence, we develop optimized FusedMM kernels for patterns denoted by the first three rows in Table III. This allows us to optimize them using architecture-dependent intrinsic codes to be discussed in the next section.

A. SIMD Vectorization and Register Blocking

While each thread computes the UPDATEU operation in Algorithm 1 for vertices of its own partition in a core, these computations can be further optimized by Single Instruction Multiple Data (SIMD) parallelism. By employing register blocking of $\mathbf{x}_u$, $\mathbf{y}_v$ and $\mathbf{z}_u$ in SIMD registers, we can effectively reduce the number of their accesses. Note that the same vectors $\mathbf{x}_u$ and $\mathbf{z}_u$ are accessed in line 13 and line 17 for all iterations of the loop in UPDATEU in Algorithm 1. Therefore, we load $\mathbf{x}_u$ in SIMD registers and initialize the registers for $\mathbf{z}_u$ with zero before entering into the loop and use those registers inside for the intermediate computations throughout all iterations of the loop. We can avoid loading of $\mathbf{z}_u$ and use non-temporal writes from registers to the memory directly without polluting the cache in this way. Therefore, the register blocking of $\mathbf{x}_u$ and $\mathbf{z}_u$ will reduce the number of accesses (either in cache or memory) for those by the number of their neighbors (average degree of the graph). We can also register block $\mathbf{y}_v$ inside the loop at line 12. However, it will reduce the number of access for $\mathbf{y}_v$ by only a factor of two since we are accessing this vector in VOP (line 13) and MOP (line 16) operations, and each iteration of the loop will access a different $\mathbf{y}_v$ vector.

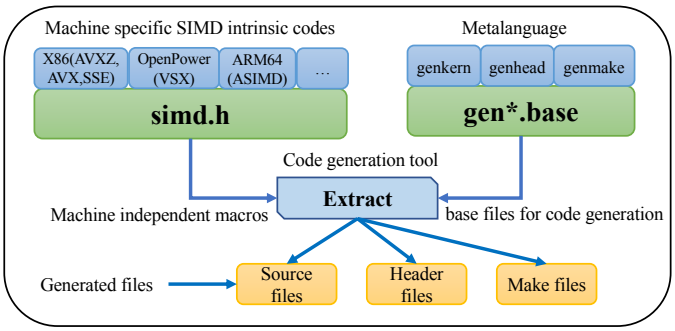


Fig. 6: Block diagram for the SIMD code generation. Three different basefiles are used to generate source, header, and makefiles using *extract* tool. The header file *simd.h* is used to provide macros for SIMD operations and to hide architecture-specific intrinsic codes.

Fig. 5 shows an example of SIMD vectorized UPDATEU operation of a specialized sigmoid-based graph embedding algorithm where the dimension d is four and the width of the SIMD register is two. Note that the pattern of this algorithm is known to our library and it already has specialized implementation for this type of operation. At the beginning of the UPDATEU function, we load $\mathbf{x}_u$ into two SIMD registers $\mathbf{Vx}_{u\_1}$ and $\mathbf{Vx}_{u\_2}$. We initialize the SIMD registers $\mathbf{Vz}_{u\_1}$ and $\mathbf{Vz}_{u\_2}$ with zero for the $\mathbf{z}_u$. We load each $\mathbf{y}_v$ into two registers in each iteration of the loop and dot-product them with $\mathbf{Vx}_{u\_1}$ and $\mathbf{Vx}_{u\_2}$ (VOP + ROP). The reduced scalar value from the dot product is scaled with sigmoid function (SOP) and broadcasted to SIMD registers. The registers $\mathbf{Vz}_{u\_1}$ and $\mathbf{Vz}_{u\_2}$ which hold the intermediate results for $\mathbf{z}_u$ are fused multiply-accumulated (FMAC) with those broadcasted values and the registers which hold the values of $\mathbf{y}_v$ (MOP + AOP). After accumulating all the values for all the neighbors into the $\mathbf{Vz}_{u\_1}$ and $\mathbf{Vz}_{u\_2}$, we store the values of these registers to memory only once after exiting from the loop and hence reduce the memory accesses for $\mathbf{z}_u$.

B. Code Generation

We use a code generation tool, *extract*, from [13] to generate SIMD intrinsic codes on different hardware architectures. Fig. 6 shows a block diagram of the structure of the code generator. We use metalanguage of *extract* to write basefiles which are used to generate source codes, header files and makefiles in our library. The header file simd.h in Fig. 6 is used to hide the hardware specific SIMD intrinsic codes and provides a common interface for all supported hardware using C preprocessor macros. Note that this design makes it very easy to add support for new SIMD instruction set architectures (ISA). We will only need to add the implementation of the macros using the intrinsic of new hardware supported by compilers. We parameterize the code generation process and generate kernels for predefined patterns of operations based on a range of dimension values, register blocking strategies,

and data types. In our implementation, we can choose vectors, prioritize the output vectors over read-only vectors for register blocking, and even limit register blocking up to a threshold when the dimension is large.

C. Complexity and expected performance

As shown in Alg. 1, FusedMM uses five operations in the UPDATEU function. Assuming each vector is of length d, the complexity of any of the five operations is $O(d)$. As FusedMM calls these operations once for every nonzero in $\mathbf{A}$, the overall computational complexity of FusedMM is $O(d * nnz)$. For memory estimations, we assume 8 byte indices and single precision values. Thus, $\mathbf{X}$ needs $4md$ bytes, $\mathbf{Z}$ needs $4md$ bytes, $\mathbf{Y}$ needs $4nd$ bytes, and $\mathbf{A}$ needs $12nnz$ bytes to store. Thus the total memory requirement of FusedMM is $8md+4nd+12nnz$ bytes. If the fused kernel is not used, one needs to store the intermediate matrix $\mathbf{H}$ that may require $12nnz * d$ bytes. Thus, FusedMM needs asymptotically less memory than separate SDDMM and SpMM operations.

To estimate the peak performance of FusedMM, we compute the arithmetic intensity (AI) which is the ratio of floating point operations to the bytes moved. Our SIMD vectorized FusedMM implementation optimizes the number of memory accesses for $\mathbf{X}$ and $\mathbf{Z}$ to their optimal values to md. The sparse matrix $\mathbf{A}$ is streamed only once (nnz). However, the number of accesses for $\mathbf{Y}$ can be $nnz * d$ (assuming no spatial or temporal locality when accessing $\mathbf{Y}$). Assuming δ to be the average degree of the graph, AI is bounded as follows:

$$\text{AI} > \frac{2dm\delta + 2dm\delta}{12m\delta + 8md + 4dm\delta} = \frac{\delta}{3\frac{\delta}{d} + 2 + \delta} \qquad (4)$$

where we used $nnz{=}m\delta$ and considered both addition and multiplications as floating point operations. Clearly, AI depends on both average degree δ and feature dimension d. To see the relative influence of δ and d, we rearrange Eqn. 4 as $(3/d + 2/\delta + 1)^{-1}$. Hence for a typical embedding dimension of $d = 128$, AI is mostly determined by the sparsity of the graph. For denser graphs ($\delta \gg 2$) with $d \gg 3$, AI approaches to its best value of 1. The worst AI of 1/6 is obtained when the graph is very sparse with $\delta = 1$ and $d = 1$. Therefore, we expect better performance for denser graphs. Nevertheless, FusedMM is still memory bound for all reasonable values of d and δ. Hence, its peak performance is bounded by the memory bandwidth.

V. EXPERIMENTS

A. Experimental Setup

Overview of experiments. We perform experiments with kernels needed by the force-directed graph layout based on the Fruchterman-Reingold (FR) model (Fig. 1(a)), the graph embedding (Fig. 1(b)), and GCN (Fig. 1(c)) algorithms. We primarily focus on the kernel timing on three different architectures. We also show end-to-end training and accuracy to assess the quality.

Experimental platforms. We conduct all of our experiments on three different servers with Intel, AMD, and ARM processors as described in Table IV. We have implemented FusedMM in C/C++ programming language with OpenMP multi-threading and SIMD intrinsic support. Our code generator can automatically generate intrinsic codes for different architectures. In our experiments, we only considered single-precision values for $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$. However, our code generator can generate efficient codes for double precision values as well. Except in the scalability experiment, we use all available cores in each processor. For all of our experiments, we measure the time for 10 iterations and report the average time.

TABLE IV: Hardware configurations of our experiments.

	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

Baselines. We use DGL (version 0.5.2) on top of PyTorch (version 1.5.1) as a baseline to compare most of our results. DGL supports a C++ backend where users can implement their own C++ functions and integrate to DGL. In addition, it has native multi-threaded SDDMM and SpMM implementations in C++. DGL also provides a python message passing API for application developments. These features give us an opportunity to integrate FusedMM with DGL, develop various high-level algorithms, and then make fair comparisons between FusedMM and DGL kernels. For both FusedMM and DGL, we measure runtime from the python interface. We also compare the SpMM specialization of FusedMM (i.e., the third row in Table III) with MKL (version 2019.5.281). In general, we term FusedMM to represent the SIMD vectorized implementation of our kernel except in Table VI.

Datasets. Table V shows a diverse set of graphs used in our experiments. It includes graphs having various number of vertices and edges, high average degree, low average degree, power-law property, etc. Some of them (e.g., Cora and Pubmed) are widely used to benchmark graph embedding and GNN algorithms. We also generate several RMAT graphs by PaRMAT [14] to assess the parameter sensitivity of FusedMM.

TABLE V: Graph datasets used in our experiments. Graphs are available at http://networkrepository.com and https://sparse.tamu.edu/

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

B. Kernel time performance

We integrate FusedMM into DGL and implement graph embedding, FR model, and GCN algorithms discussed in Fig. 1 and Table III. We also implement these algorithms

TABLE VI: Kernel time (in sec.) for Graph Embedding, FR model, and GCN on Inter server. A '×' sign represents memory allocation error (i.e., out of memory). The columns 32, 64, 128, 256, and 512 represent the dimensions (d). FusedMM and FusedMMopt represent the general implementation and SIMD vectorized implementation of our proposed kernel, respectively. Speedup is computed for FusedMMopt over DGL.

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	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	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	-

using the SDDMM and SpMM kernels of DGL. Then, we run FusedMM-based algorithms and DGL's kernel based algorithms from the python interface and measure the runtimes of FusedMM and DGL kernels (excluding IO and prepossessing).

Performance on Intel Server. We report the kernel time of FusedMM and DGL in Table VI for graph embedding, FR graph layout model, and GCN. Here, FusedMMopt represents the SIMD vectorized implementation of FusedMM. Due to space restriction, we choose Ogbprot., Youtube, and Orkut graphs as representatives of high average degree, low average degree, and bigger size graph, respectively. To analyze the efficiency of register blocking, we show the results for various dimensions (i.e., d) of $\mathbf{X}$. Our key findings are listed below.

(1) *FusedMM even without any optimization runs up to* $9.8\times$ *faster than DGL.*The speedups of unoptimized FusedMM relative to DGL are: best $9.8\times$ for $d = 512$ with FR model on Youtube graph; worst $1.4\times$ for $d = 32$ with GCN on Orkut graph; average $4.2\times$ over all graphs and dimensions. These results clearly demonstrate the benefit of a fused kernel that does not store intermediate matrices.

(2) *The optimized FusedMM kernel gives us up to* $5.4\times$ *speedups over unoptimized FusedMM.* The speedups of FusedMMopt relative to unoptimized FusedMM are: best $5.4\times$ for d=128 with FR model using Ogbprot.; worst $1.0\times$ for d=128 with GCN using Youtube; average $2.7\times$ over all graphs and dimensions. These results clearly demonstrate the effectiveness of vectorization and autotuned intrinsic operations.

(3) *DGL may go out of memory when high-dimensional messages are used.* The FR graph layout algorithm generates d-dimensional messages on each edge. This may require prohibitive memory to store the intermediate matrix $\mathbf{H}$. For example, Table VI shows that DGL goes out of memory on Orkut for 256 and 512 dimensions. As FusedMM does not store $\mathbf{H}$, it is more robust with respect to feature dimensions. For the same reason, FusedMMopt achieves its best speedup over DGL when used in the FR model.

(4) *FusedMM performs better on denser graphs.* Table VI shows that FusedMM performs better on Ogbprot., the most dense graph in our test suite. This is expected because denser graph can amortize memory latency costs and usually have higher arithmetic intensities.

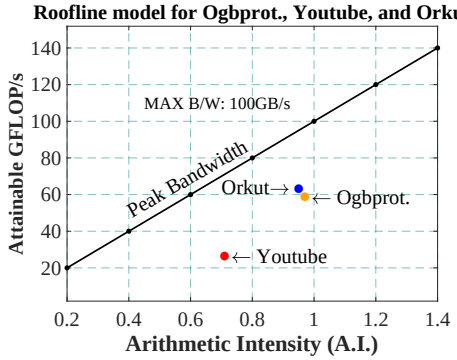


Fig. 7: Roofline model of FusedMM for Ogbprot., Youtube, and Orkut graphs on Intel server for graph embedding. The STREAM bandwidth on this server is 100 GB/s. We set d to 128.

(5) *FusedMM performs better for higher dimensions.* Table VI clearly shows that FusedMMopt becomes more effective at higher dimensions (relative to DGL). This is due to FusedMM's register blocking strategy that efficiently utilizes available registers with d-dimensional feature vectors.

Overall, FusedMMopt comprehensively outperforms DGL by a significant margin on all graphs for all dimensions. It is possible that the speedup of FusedMMopt drops a little after the dimension 128 (especially in GCN) because we may observe some register spilling at higher dimensions.

Roofline analysis of FusedMM. Based on Eq. 4, we show a roofline model [15] of the graph embedding task on Intel server in Fig. 7. We observe that FusedMM achieves 63.21 GFLOP/s for the Orkut graph with an AI of 0.95. For Orkut, the best possible performance is 95.27 GFLOP/s according to the Roofline model. Similarly for other graphs, the observed performance is reasonably good, but they fall a little short of the best possible performance. This gap between the observed and attainable performance comes from the overheads associated with python function calls. When we directly called FusedMM in a C++ code, the observed performance is very close to the attainable performance. We still report the performance observed from the python interface because this performance is realized by end users.

Comparison with Intel MKL. Since MKL [16] does not have an SDDMM operation, we only compare SpMM-based GCN (3^{rd} row of Table III) with MKL's SpMM function. For this experiment, we measure the kernel time of SpMM from the C++ interface. We measure both inspection and execution time for MKL. Table VII shows that the SpMM specialization of FusedMM performs comparably to MKL's SpMM implementation. Thus, despite being a multipurpose kernel, FusedMM can match the best performing specializations of SpMM.

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

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

Performance on ARM ThunderX and AMD EPYC. We conduct similar experiments on ARM and AMD servers and report the results in Figs. 8 and 9. For all these results, we use d=128. Due to memory limitations, we could not generate results for Ogbprot. and Orkut. In Fig. 8 (a), we observe that FusedMM is up to $19.2\times$ faster than DGL. As with the Intel processor, the observed performance originates from the fused design and effective register blocking of SIMD vectorization. As our ARM server has no L2 cache, FusedMM gets the full advantage of register blocking. Fig. 9 shows that FusedMM achieves up to $11.4\times$ speedup over DGL. Notably, optimized codes for ARM and AMD processors were autotuned using our code generator. Hence, application developers do not need to write optimized codes for different architectures.

C. Sensitivity Analysis

Scalability. Fig. 10 shows the scalability of FusedMM and DGL for graph embedding. We perform this experiment on the Intel server for Orkut graph with d=256. We observe that FusedMM on 32 cores is $\sim 20\times$ faster than its sequential runtime. DGL's kernels also scale well achieving up to $16\times$ speedup, but runs slower than FusedMM for all thread counts.

Memory consumption. When SDDMM and SpMM operates in tandem in DGL, we need to store intermediate results in $\mathbf{H}$ (Eq. 2). This can consume a significant amount of memory when an application (e.g., FR graph layout) generates a sparse-tensor as depicted in Fig. 3. Fig. 10(b) shows that DGL's memory requirement grows linearly with d for the FR model while the memory consumption of FusedMM remains stable. This gives FusedMM a clear advantage over unfused kernels in DGL for tasks that require high-dimensional messages.

Parameter sensitivity. We study the performance of FusedMM by changing average degrees of graphs and feature dimensions. Fig. 11(a) shows the speedup of FusedMM over DGL for various average degrees of RMAT graphs with 100K vertices. The initial graph has one million edges and we increase this by a factor of 2. We observe that the speedup of FusedMM increases with the increase in the average degree. These results are consistent for both the FR model and graph embedding. In Fig. 11 (b), we show the kernel time of FusedMM and DGL for Flickr varying the dimension. We observe that both kernels show similar sensitivity to d. FusedMM is significantly faster than DGL for all values of d, and their performance gap widens as d increases.

TABLE VIII: Graph Embedding application time per-epoch for different methods ($d = 128$, and batch size is 256).

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×

D. End-to-End training

We developed three implementations of the Force2Vec [4] graph embedding algorithm to perform end-to-end training. We implement Force2Vec using standard kernels in PyTorch, SDDMM and SpMM kernels in DGL, and using FusedMM. We set embedding dimension to 128, batch size to 256, and the number of epochs to 800. Due to the high mini-batch processing time of DGL, we use Cora and Pubmed for this experiment. These two graphs are widely used to benchmark graph embedding and graph neural network methods [5, 17]. When the end-to-end training is considered, Table VIII shows that FusedMM is up to $25\times$ and $45\times$ faster than DGL and PyTorch, respectively. FusedMM performs significantly better than DGL because FusedMM can directly take a scaling operations (SOP in Table III). By contrast, DGL needs to generate the intermediate results to perform the scaling.

Accuracy: Finally, we assess the quality of the embedding generated by the Force2Vec algorithm implemented using FusedMM. As FusedMM does not alter the actual computations performed, we do not expect any performance loss compared to the original implementation of Force2Vec. Indeed the original Force2Vec and FusedMM-based Force2Vec both achieve the same F1-micro scores of 0.78 and 0.79 when performing node classifications on Cora and Pubmed datasets.

VI. Related Work

Graphs and sparse matrices are fundamentally related, and their duality [18] has been exploited in many graph algorithms [19]–[21] and libraries [22]–[26]. Over the last few years, graph ML algorithms have been increasingly using linear algebra kernels to capture various message passing operations. For example, the original GCN implementation from Kipf and Welling [5] used SpMM to capture the graph convolution operation. SpMM-like operations were also used in high-performance graph layout [2] and embedding [4] algorithms.

Recently, several specialized frameworks were developed to make the processing of GNN workloads easier and faster. Among them PyG [10] and DGL [11] provides message-passing APIs to develop high-level applications. However, they use linear algebra kernels in the back end for performance.

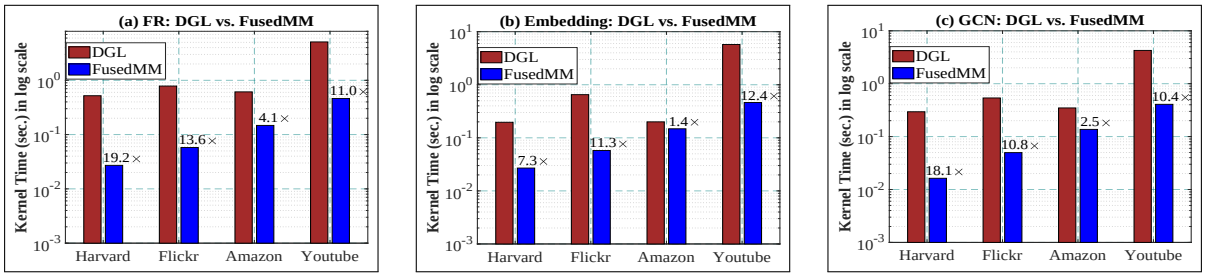


Fig. 8: Kernel time of FusedMM and DGL on ARM server using various benchmark graphs ($d = 128$) for (a) FR model, (b) Graph Embedding, and (C) GCN. In all the figures, the speedup of FusedMM over DGL is shown above its representative (blue colored) bars.

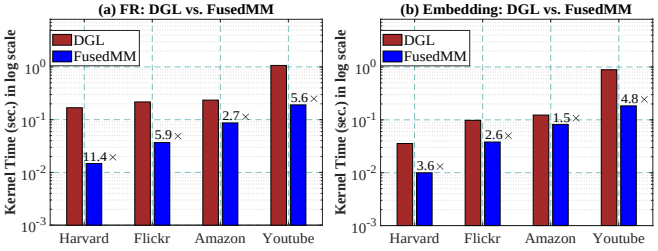


Fig. 9: Kernel time of FusedMM and DGL on AMD server using various benchmark graphs (here, $d = 128$) for (a) FR model, and (b) Graph Embedding. In all the figures, the speedup of FusedMM with respect to DGL is shown above its representative (blue colored) bars.

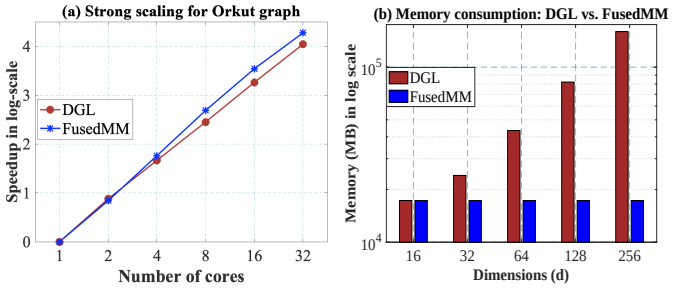


Fig. 10: (a) Strong scaling of FusedMM and DGL for Graph Embedding using Orkut graph with respect to their sequential execution (here, $d = 256$). (b) Memory consumption in megabytes in the FR model of DGL and FusedMM for Ogbprot.

DGL explicitly calls SDDMM and SpMM implementations for message generation and aggregation operations. Hence, FusedMM can directly substitute linear algebra kernels used by DGL. Note that the fusion of SDDMM and SpMM kernels has also been used to compute word mover's distance [27].

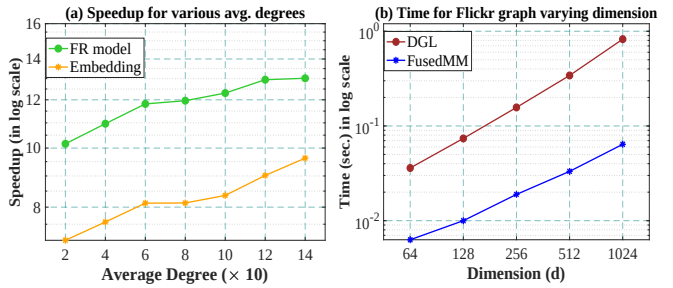


Fig. 11: (a) Speedup of FusedMM over DGL for various RMAT graphs with 100K vertices and various average degrees. (b) Kernel time of Flickr graph for graph embedding varying the size of d.

When standard SpMM and SDDMM kernels are used in GNN, PyG and DGL call vendor-provided libraries (e.g., MKL [16] and cuSPARSE [28]) because these libraries provide highly optimized implementations for sparse kernels. However, DGL also provides general implementations of SpMM and SDDMM kernels to capture complex graph convolutions and attention mechanisms used in GNNs [5, 12, 29]–[31]. Recently, Huang et al. [17] developed a general-purpose SpMM algorithm for GPUs. When integrated with DGL and PyG, their GE-SpMM kernel can expedite the computations of GCN and pooling based GNNs such as GraphSAGE [30]. FeatGraph [32] provides efficient implementations of SDDMM and SpMM for both CPUs and GPUs. In that sense, FusedMM developed in this paper is a generalization of FeatGraph.

VII. Discussions and Conclusions

We present a flexible linear algebra kernel called FusedMM that captures the core computations of most graph embedding and graph neural network algorithms. Conventionally, graph learning libraries such as PyG and DGL rely on at least two matrix operations for edge-wise message generations and vertex-wise message aggregations. FusedMM substitutes them with just one general-purpose operation with the support of uses-defined functions. Our results are unexpectedly positive because FusedMM even without any optimization runs significantly faster than equivalent kernels used in DGL. This clearly demonstrates the value of reducing memory traffic using fused operations in graph machine learning. Even though graph algorithms are harder to vectorize due to irregular computations, we were able to make FusedMM up to $5\times$ faster using our automatically tuned vectorized operations. Thus, this paper brings in the philosophy of Automatically Tuned Linear Algebra Software (ATLAS) [13] in sparse computations, which was an unexplored territory in graph analytics. Fused kernels and autotuning approaches together make high-level graph learning algorithms at least an order of magnitude faster than unfused and untuned kernels.

While this paper only considers CPU implementations, the vectorization and autotuning techniques developed for FusedMM are easily applicable to GPUs as well. On GPUs, we will employ threads (SIMT) in place of lanes of registers (SIMD) such that each thread in a block will compute an element of the row of the dense matrix. Thus, FusedMM provides

a general-purpose technique to accelerate graph analysis and graph machine learning.

ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their feedback. Funding for this work was provided by the Indiana University Grand Challenge Precision Health Initiative.

REFERENCES

[1] M. Jacomy, T. Venturini, S. Heymann, and M. Bastian, "ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software," *PloS one*, vol. 9, no. 6, 2014.

[2] M. K. Rahman, M. H. Sujon, and A. Azad, "BatchLayout: A batch-parallel force-directed graph layout algorithm in shared memory," in *Proceedings of PacificVis*. IEEE, 2020, pp. 16–25.

[3] A. Tsitsulin, D. Mottin, P. Karras, and E. Müller, "VERSE: Versatile graph embeddings from similarity measures," in *Proceedings of WWW*, 2018, pp. 539–548.

[4] M. K. Rahman, M. H. Sujon, and A. Azad, "Force2Vec: Parallel force-directed graph embedding," in *Proceedings of ICDM*. IEEE, 2020.

[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *Proceedings of ICLR*, 2017.

[6] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *Proceedings of ICLR*, 2019.

[7] H. Zhao, "High performance machine learning through codesign and rooflining," Ph.D. dissertation, UC Berkeley, 2014.

[8] I. Nisa, A. Sukumaran-Rajam, S. E. Kurt, C. Hong, and P. Sadayappan, "Sampled dense matrix multiplication for high-performance machine learning," in *HiPC*. IEEE, 2018, pp. 32–41.

[9] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 300–314.

[10] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[11] 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," *arXiv preprint arXiv:1909.01315*, 2019.

[12] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *Proceedings of ICLR*, 2017.

[13] R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of SC*, 1998.

[14] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-efficient graph processing on GPUs," 2015, pp. 39–50.

[15] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[23] A. Buluç and J. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *IJHPCA*, vol. 25, no. 4, pp. 496–509, 2011.

[16] *Intel Corporation, Intel Math Kernel Library*. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html

[17] G. Huang, G. Dai, Y. Wang, and H. Yang, "GE-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks," *Proceedings of SC*, 2020.

[18] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011, vol. 22.

[19] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, and A. Buluç, "HipMCL: A high-performance parallel implementation of the Markov clustering algorithm for large-scale networks," *Nucleic acids research*, vol. 46, no. 6, pp. e33–e33, 2018.

[20] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *IPDPSW*, 2015, pp. 804–811.

[21] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[22] T. A. Davis, "Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.

[24] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *arXiv preprint arXiv:1503.07241*, 2015.

[25] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–9.

[26] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[27] J. J. Tithi and F. Petrini, "An efficient shared-memory parallel sinkhorn-knopp algorithm to compute the word mover's distance," *arXiv preprint arXiv:2005.06727*, 2020.

[28] *Nvidia Corporation, cuSPARSE library Library*. [Online]. Available: https://developer.nvidia.com/cusparse

[29] K. K. Thekumparampil, C. Wang, S. Oh, and L.-J. Li, "Attention-based graph neural network for semi-supervised learning," *arXiv preprint arXiv:1803.03735*, 2018.

[30] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NeurIPS*, 2017, pp. 1024–1034.

[31] J. Chen, T. Ma, and C. Xiao, "Fastgcn: fast learning with graph convolutional networks via importance sampling," *arXiv preprint arXiv:1801.10247*, 2018.

[32] Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang, "FeatGraph: A flexible and efficient backend for graph neural network systems," *Proceedings of SC*, 2020.