Graphite: Optimizing Graph Neural Networks on CPUs Through Cooperative Software-Hardware Techniques

Zhangxiaowen Gong
University of Illinois at Urbana-Champaign and Intel Labs
USA
zhangxiaowen.gong@intel.com

Houxiang Ji
University of Illinois at Urbana-Champaign
USA
hj14@illinois.edu

Yao Yao
University of Illinois at Urbana-Champaign
USA
yaoy4@illinois.edu

Christopher W. Fletcher
University of Illinois at Urbana-Champaign
USA
cwfletch@illinois.edu

Christopher J. Hughes
Intel Labs
USA
christopher.j.hughes@intel.com

Josep Torrellas
University of Illinois at Urbana-Champaign
USA
torrella@illinois.edu

ABSTRACT

Graph Neural Networks (GNNs) are becoming popular because they are effective at extracting information from graphs. To execute GNNs, CPUs are good platforms because of their high availability and terabyte-level memory capacity, which enables full-batch computation on large graphs. However, GNNs on CPUs are heavily memory bound, which limits their performance.

In this paper, we address this problem by alleviating the stress of GNNs on memory with cooperative software-hardware techniques. Our software techniques include: (i) layer fusion that overlaps the memory-intensive phase and the compute-intensive phase in a GNN layer, (ii) feature compression that reduces memory traffic by exploiting the sparsity in the vertex feature vectors, and (iii) an algorithm that changes the processing order of vertices to improve temporal locality. On top of the software techniques, we enhance the CPUs' direct memory access (DMA) engines with the capability to execute the GNNs' memory-intensive phase, so that the processor cores can focus on the compute-intensive phase. We call the combination of our software and hardware techniques *Graphite*.

We evaluate Graphite with popular GNN models on large graphs. The result is high-performance full-batch GNN training and inference on CPUs. Our software techniques outperform a state-of-the-art GNN layer implementation by 1.7-1.9x in inference and 1.6-2.6x in training. Our combined software and hardware techniques speed-up inference by 1.6-2.0x and training by 1.9-3.1x.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Multicore architectures**; *Single instruction, multiple data.*

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527403

KEYWORDS

Graph Neural Networks, hardware-software co-design, CPU, DMA

ACM Reference Format:
Zhangxiaowen Gong, Houxiang Ji, Yao Yao, Christopher W. Fletcher, Christopher J. Hughes, and Josep Torrellas. 2022. Graphite: Optimizing Graph Neural Networks on CPUs Through Cooperative Software-Hardware Techniques. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3470496.3527403

1 INTRODUCTION

Deep Neural Networks (DNNs) have attained state-of-the-art results across a range of tasks from image recognition [30] to speech recognition [2], scene generation [37], and game playing [44]. However, traditional DNNs such as Convolutional Neural Networks (CNNs) are only applicable to Euclidean data, such as a grid of pixels in an image. They lack the power to process non-Euclidean data, such as graphs [51].

Graphs model a set of objects (vertices) and their relationships (edges). Many important types of data are represented as graphs — e.g., a network of e-commerce products that are purchased together, the biologically meaningful associations between proteins, or a citation network of papers [23]. Graphs are often irregular. They have a set of unordered vertices, and each vertex may link to a different number of neighbors. Consequently, operations like convolutions are difficult to apply in the graph domain [51]. As a result, there is an increasing demand for a DNN model that can operate on graphs. Graph Neural Networks (GNNs) fill this need. They have been proven effective in social science [21, 29], physical systems [41], knowledge graphs [20], and other domains [12, 14].

Although the community often uses GPUs to run GNNs [26], and has proposed several accelerators for GNNs [15, 33, 54], running GNNs on CPUs has benefits. First, CPUs are ubiquitous. For example, data centers possess a high number of spare CPUs during off-peak hours that can be utilized to perform GNN tasks, lowering the Total Cost of Ownership (TCO) [1, 40, 46, 53]. Second, CPUs can be equipped with terabytes of memory [40, 53]. Memory capacity is a major limiting factor in GNN execution. Real-world graphs can have millions to billions of vertices [4, 6], and their footprint may occupy tens to hundreds of gigabytes.

Although techniques such as neighborhood sampling and vertex mini-batching are proposed to cope with GPUs' limited memory capacity [21], these workarounds have drawbacks. They induce vast under-utilization of the compute capacity [24, 31], may degrade the network accuracy [21, 26], and induce significant overhead through additional costly operations [47]. Instead, a CPU can work with full-batches without sampling for large graphs. It also enables wider and deeper network structures, which are trending [31, 32, 34]. In light of these benefits, the goal of this paper is to characterize and optimize the execution of GNNs on CPUs.

Efficiently executing GNNs on CPUs is non-trivial. While traditional DNN workloads are usually regular and compute-intensive, GNNs are irregular and often memory-intensive, which is due to the sparse connections in graphs. Therefore, GNNs pose distinct performance challenges compared to other DNNs. We profiled GNN workloads using the state-of-the-art DGL framework [49] on a CPU server and found that the executions are heavily DRAM-bandwidth bound — less than 10% of the pipeline slots are spent on useful computation. Therefore, alleviating the stress on memory is key to speeding-up GNN workloads on CPUs.

In this paper, we propose cooperative software-hardware techniques to tackle this memory problem. We call the combination of our techniques *Graphite*. We start with three software techniques. First, we note that a GNN layer is composed of a memory-intensive *aggregation phase*, where each vertex collects information from its neighbors, and a compute-intensive *update phase*, where a deep learning operator such as a fully-connected layer processes the collected information [56]. Hence, to speed-up the execution, we fuse the two phases, overlapping the memory accesses of the aggregation phase with the computation of the update phase. Second, we observe that the vertex features in the hidden GNN layers often contain a moderate number of zeros due to the use of the rectified linear unit (ReLU) and dropout. Hence, we reduce memory traffic by compressing the sparse features before writing to memory and decompressing them after reading from memory. Third, we develop an effective algorithm to improve temporal locality by rearranging the processing order of the vertices.

Although these three software optimizations are effective, there are still remaining issues to address. Specifically, processor cores stall while fetching low-locality data in the aggregation phase. Therefore, we propose a light-weight near-memory compute unit on top of the software techniques. We observe that modern Direct Memory Access (DMA) engines include the gather functionality [25, 42, 52]. Since the aggregation is generally a gather and a reduce, we propose to augment the DMA engines to offload the aggregation from the processor cores. Since the aggregation reuses many existing resources in the original DMA engines, the hardware extensions are limited.

The Graphite software and hardware techniques are synergistic in most cases. Their combined capabilities result in greatly reduced private cache accesses and miss rates, higher DRAM bandwidth utilization, and substantial compute-memory overlap.

Overall, in this paper, we make the following contributions. First, we develop effective software techniques to relieve the DRAM bandwidth pressure in GNN workloads running on CPUs. Second, we address the remaining memory issues by proposing a novel DMA engine that offloads the GNN aggregation. We apply these software-hardware optimizations to both inference and training. Third, we validate our techniques with full-batch computation on medium to large scale graphs. We evaluate our software techniques on a 28-core server, and our hardware technique on a simulator. Our software techniques outperform a state-of-the-art GNN layer implementation on popular GNN models by 1.7-1.9x in inference and 1.6-2.6x in training. Our combined software and hardware techniques speed-up inference by 1.6-2.0x and training by 1.9-3.1x.

2 BACKGROUND

2.1 Graph Neural Networks (GNNs)

GNNs have become popular tools to extract information from graphs, which is hard for other types of DNNs [51]. To introduce the general formulation of GNNs, we use the notations in Table 1.

Table 1: List of symbols.

	Description		Description
$\mathcal{G}$	graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$	$\mathcal{V}$	vertices of $\mathcal{G}$
$\mathcal{E}$	edges of $\mathcal{G}$	D_v	degree of vertex v
$\mathcal{N}(v)$	all neighbors of vertex v	$\mathcal{S}(v)$	sampled subset of $\mathcal{N}(v)$
$e_{u,v}$	edge between vertex u and v	$\mathbf{A}$	adjacency matrix
K	number of layers	H	vertex feature vector length
$\mathbf{h}$	feature matrix	$\mathbf{h}_v$	feature vector of vertex v
$\mathbf{a}$	aggregation feature matrix	$\mathbf{a}_v$	aggregation feature vector
$\mathbf{W}$	update weight matrix		of vertex v
$\mathbf{b}$	update bias vector	ψ	feature processing function

The inputs to GNNs are graphs. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ has vertices $\mathcal{V}$ and edges $\mathcal{E}$. Each vertex/edge can have a feature vector. Each element in a feature vector is a scalar feature. In this paper, we study the cases where vertices have features while edges do not.

A GNN has K layers. Each layer contains an *aggregation* phase and an *update* phase [56]. In the aggregation phase of layer k, each vertex v reduces its neighbors' as well as its own feature vectors at layer $k-1$ to create the aggregation feature vector $\mathbf{a}_v^k$ through an aggregation function:

$$\mathbf{a}_v^k = \text{AGGREGATE}(\mathbf{h}_u^{(k-1)} \mid \forall u \in \mathcal{N}(v) \cup \{v\}) \tag{1}$$

Next, in the update phase, the vertex computes its output feature vector $\mathbf{h}_v^k$ by applying an update function on $\mathbf{a}_v^k$:

$$\mathbf{h}_v^k = \text{UPDATE}(\mathbf{a}_v^k) \tag{2}$$

After K layers, each vertex's feature vector is a function of its neighbors up to K hops away.

To execute GNNs with large input graphs on memory-limited devices such as GPUs and accelerators, it is essential to first divide the graph into mini-batches of vertices. Then, one may perform a breadth-first search (BFS) to find the K-hop neighborhood of each vertex in a mini-batch. Finally, only the input feature vectors of the neighborhoods need to be transferred to the device memory.

However, depending on the graph structure and the depth of the network, the BFS can span the entire connected components that contain the vertices in the mini-batch, which can still be too large for the device memory. Hence, to confine each mini-batch's memory footprint, some networks *sample* each vertex's neighborhood before

the aggregation phase [21, 55], by randomly selecting up to some pre-determined number of neighbors for each vertex:

$$\mathcal{S}(v) = \text{SAMPLE}^k(\mathcal{N}(v)) \tag{3}$$

With a fixed sample size, the upper bound of the working set of each mini-batch is predetermined. The sampling process is generally performed on CPUs [10, 39].

Different GNN models may adopt various aggregation and update functions. Table 2 presents two popular GNN models: the Graph Convolutional Network (GCN) [29] and the GraphSAGE (SAGE) with the mean aggregator [21].

Table 2: Example GNN models.

GNN	Aggregation	Update
GCN	$\sum(\mathbf{h}_u^{(k-1)}/\sqrt{D_v \cdot D_u}) \mid \forall u \in \mathcal{N}(v) \cup \{v\}$	$ReLU(\mathbf{W}^k a_v^k + \mathbf{b}^k)$
SAGE	$\sum \mathbf{h}_u^{(k-1)}/(D_v + 1) \mid \forall u \in \mathcal{N}(v) \cup \{v\}$	$ReLU(\mathbf{W}^k a_v^k + \mathbf{b}^k)$

Both models use the same update function: a fully-connected (FC) layer activated with the rectified linear unit (ReLU). Their difference lies in the aggregation function. In GCN, each vertex first normalizes each neighbor's and its own feature vectors. It then sums the normalized feature vectors. In GraphSAGE, each vertex takes the element-wise average of its neighbors' and its own feature vectors. Despite the difference, the aggregation functions of the two models both gather each vertex neighbors' feature vectors, process each gathered feature vector with a function ψ, and finally perform a reduction. Both models can adopt sampling by replacing $\mathcal{N}(v)$ with $\mathcal{S}(v)$ in the aggregation.

Training GNNs follows the general DNN training principles. The training process iteratively updates the trainable parameters (i.e., $\mathbf{W}$ and $\mathbf{b}$ in the two example models) with a loop of the forward pass and the backward pass. The forward pass computes the outputs with the current parameters, compares them with the ground truth, and produces errors. The backward pass propagates error gradients with the chain rule and updates the parameters accordingly.

Popular GNN frameworks include PyTorch Geometric (PyG) [11] and the Deep Graph Library (DGL) [49].

2.2 Sparsity in GNNs

Sparsity in a neural network is the presence of zeros in any of the input or output tensors. Zero-valued inputs result in useless multiplications and/or additions during the aggregation and update phases. Reading these zero-valued inputs from memory also wastes bandwidth. If we can efficiently "skip over" the zeros, we may save computation and/or memory bandwidth.

A GNN model can contain sparsity from various sources. First, a typical graph often has highly-sparse connections, i.e., each vertex is connected to a small subset of other vertices. When expressing the connections as an adjacency matrix, $\mathbf{A}$, the matrix is usually over 99% sparse. For example, the Amazon product co-purchasing network dataset (*products*) [23] has 2.4M vertices and 62M undirected edges, suggesting that $\mathbf{A}$ is 99.998% sparse. When uncompressed, the footprint of A is $O(|\mathcal{V}|^2)$, but when using a typical compressed format such as compressed sparse row (CSR), the footprint is only approximately $O(|\mathcal{E}| + |\mathcal{V}|)$. In addition, as the graph structure is fixed, so are the locations of the zeros. For these reasons, the adjacency matrix is often compressed. Frameworks such as DGL use the CSR format.

The second source of sparsity lies in the features. Some GNNs employ ReLU, which sets any negative scalar output feature to zero [51]. ReLU typically induces 40-90% sparsity [17]. In addition, *feature dropout* can also sparsify the features. It is widely adopted to reduce overfitting [45]. During training, a predefined fraction of the hidden features, often 50%, are randomly selected and set to zero. We profiled a 20-epoch training of a three-layer GraphSAGE on the *ogbn-products* dataset, and found that ReLU sparsifies the input features to the second layer by over 60%, and dropout further sparsifies them to over 80%. Furthermore, the sparsity of the input features to the third layer is even higher, reaching over 90%.

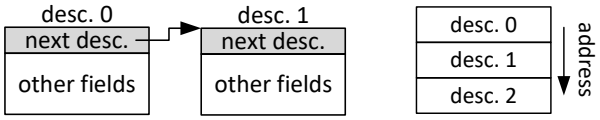
Sparsity in the features is hard to exploit. Both ReLU and dropout introduce a moderate amount of zeros. Also, the sparsity pattern changes dynamically, so frequent compression becomes costly. Therefore, feature sparsity is rarely exploited, and features are typically stored in a regular (dense) representation.

2.3 Scatter-Gather DMA

The aggregation phase of GNNs consists of each vertex gathering its neighbors' feature vectors and performing a simple reduction. Currently, machines execute this phase very inefficiently: processors spend substantial time fetching data from the lower levels of the cache hierarchy to only perform a simple computation, and are then unlikely to reuse the data from their caches because the data has poor locality.

One direction to improve efficiency is to offload the aggregation to a near-memory processor. However, these processors add significant hardware overhead and do not currently exist in commercial systems. On the other hand, we notice that Direct Memory Access (DMA) engines, such as various FPGA IPs [42, 52], do exist in current systems and are light weight. They support minimal functionality like a scatter-gather function. Hence, there is an opportunity to augment DMA engines to implement hardware-assisted aggregation with relatively low cost and minimal intrusiveness.

Existing DMA engines usually employ a descriptor-based programming interface. A descriptor encodes the source and destination addresses of the data block to be transferred, as well as its size. The scatter/gather operations supported are essentially batched data transfers. To describe a gather operation, the software needs to supply a chain of descriptors, where each one encodes the movement of a contiguous data block. The chain can be in the form of a linked list (e.g., Xilinx AXI [52] in Figure 1a) or an array (e.g., Intel DSA [25] in Figure 1b).



(a) Linked-list based. (b) Array based.

Figure 1: Scatter-gather DMA descriptor chain.

3 MOTIVATION: GNNS ON CPUS

GPUs and accelerators are popular platforms for various types of DNNs due to their high compute power. While this is true for GNNs too [15, 26, 33, 54], CPUs are also a viable platform for GNNs [36] for at least two reasons. First, CPUs are ubiquitous. For example, data centers possess a high number of spare CPUs during off-peak hours that can be utilized to perform GNN tasks, lowering the Total Cost of Ownership (TCO) [1, 40, 46, 53].

The second reason is that GNN execution requires large memory capacities. For example, real-world graphs can have millions to billions of vertices [4, 6], so the footprint of their feature matrices may occupy tens to hundreds of gigabytes. Traditionally, the memory devices used in GPUs and accelerators prioritize throughput over capacity. As a result, GNN memory footprints typically exceed GPUs and accelerators' memory capacity.

To run large input graphs on those devices, one may use sampling and mini-batching to confine the working sets. However, these techniques have drawbacks. First, the size of the K-hop neighborhood grows exponentially with the number of layers K. With enough layers, GPU users may be forced to use a tiny mini-batch size to fit the working sets in the device memory, which vastly under-utilizes the compute capacity [24, 31]. Second, sampling may degrade the network accuracy [21, 26]. Third, the techniques induce significant overhead [10, 39].

To examine the overhead of the sampling and mini-batching techniques, we profiled the training of a sampled GraphSAGE on the *products* dataset with DGL on a CPU-GPU heterogeneous platform. The GNN layers are computed on a Nvidia Titan V GPU while the sampling is performed on a 12-core CPU. We experimented with mini-batch sizes from 1024 to 4096. Figure 2 shows the breakdown of the training epoch time. The numbers in the figure are the time spent on: (i) sampling/mini-batching and (ii) GNN layer computation. We see that the sampling and mini-batching astoundingly contribute to over 80% of the total training time. Moreover, the training time increases significantly as the mini-batch size shrinks.

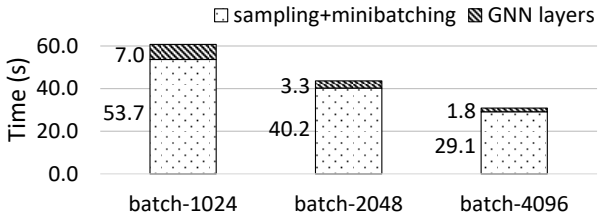


Figure 2: Training epoch time breakdown of a sampled GSC on a GPU with different mini-batch sizes.

While GPUs and accelerators are equipped with tens of gigabytes of memory, CPUs' memory capacity has reached the order of terabytes [40, 53]. This enables full-batch training without sampling for large graphs. It also benefits wider and deeper network structures, which are trending [31, 32, 34].

This paper aims to improve the performance of full-batch GNN training and inference on multi-core SIMD CPUs without sampling. To understand the performance bottlenecks of these workloads, we profiled GNN training on a 28-core Intel Cascade Lake CPU server with DGL using the Intel VTune profiler. Figure 3 is a breakdown of the pipeline slots either doing useful work or wasted on different bottlenecks during the training.

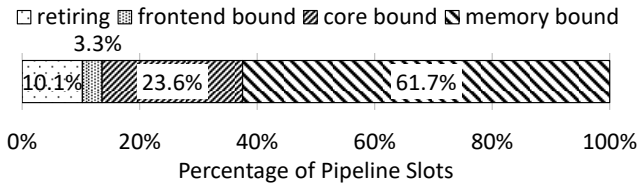


Figure 3: Breakdown of the pipeline slots spent on retiring micro-ops or stalled by different bottlenecks during a full-batch training of GraphSAGE on a CPU.

We observe that only 10% of the pipeline slots are attributable to useful work. Further, in 62% of the slots, the pipeline is stalled waiting for loads and stores. In addition, it can be shown that the L1 data cache line fill buffer is full almost 100% of the time, hinting that L1 misses are often satisfied from deep in the memory hierarchy, such as from main memory. This is because the aggregation phase typically constitutes over 80% of the execution time. Since the aggregation performs a simple reduction for each vertex after gathering its neighbors' feature vectors, the operation is highly memory-intensive. Therefore, reducing main memory bandwidth pressure appears to be key to optimize GNN workloads on CPUs.

In the rest of this paper, we present and evaluate new software and hardware techniques to accomplish this goal.

4 SOFTWARE OPTIMIZATION TECHNIQUES

This section presents our software techniques to optimize GNN execution on multi-core CPUs. The optimizations in Sections 4.1-4.3 benefit both training and inference, while the one in Section 4.4 further optimizes training.

4.1 Parallel Vectorized Aggregation

The aggregation phase dominates the execution time of a GNN layer. In the aggregation, each vertex v first gathers the feature vectors from $u \in \mathcal{N}(v) \cup \{v\}$, then performs an element-wise reduction, and finally updates its aggregation feature vector, $\mathbf{a}_v$. All working sets but $\mathbf{a}^k$, are read-only. Therefore, we output-parallelize the aggregation by letting threads compute different subsets of $\mathbf{a}^k$. This avoids race conditions and requires no synchronization among threads.

Algorithm 1 shows our parallel vectorized aggregation. In Line 1, we divide $\mathcal{V}$ into chunks of T vertices. Each parallel task operates on a chunk. The processing time of a chunk correlates with the degrees of the vertices in it. The degrees can vary significantly and sometimes follow a power law distribution [36]. To balance the load among threads, we schedule the parallel tasks with OpenMP's *dynamic* scheduler.

A parallel task performs aggregation on each vertex v in the assigned chunk (Lines 4-7). Because the feature vector of a vertex often has hundreds of elements, we vectorize the feature gathering, processing, and reduction (Line 7).

Algorithm 1: Parallel vectorized aggregation.

input : graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, input feature matrix $\mathbf{h}^{k-1}$, reduction operator $\oplus$, feature processor ψ
output : aggregation feature matrix $\mathbf{a}^k$
constant : task size T, vector length V, prefetch distance D

1 **for** $i = 0$ **to** $|\mathcal{V}| - 1$ **step** T **in parallel do**
2 **for** $j = 0$ **to** T **do**
3 $v = \mathcal{V}_{i+j}$
4 $\mathbf{a}_v^k = \{0\}$
5 **for** $u \in \mathcal{N}(v) \cup \{v\}$ **do**
6 **for** $m = 0$ **to** $|\mathbf{a}_v^k|$ **step** V **do**
7 $\mathbf{a}_{v[m:m+V-1]} = \mathbf{a}_{v[m:m+V-1]} \oplus \psi(\mathbf{h}^{k-1}_{u[m:m+V-1]})$
8 $v' = \mathcal{V}_{i+j+D}$
9 PREFETCH($\mathbf{h}_{u'}^{(k-1)} \mid \forall u' \in \mathcal{N}(v') \cup \{v'\}$)

Algorithm 2: Fused aggregation and update.

input : graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, input feature matrix $\mathbf{h}^{k-1}$
output : aggregation feature matrix $\mathbf{a}^k$, output feature matrix $\mathbf{h}^k$
constant : block size B, blocks per task T, prefetch distance D

1 **for** $i = 0$ **to** $|\mathcal{V}| - 1$ **step** $T \cdot B$ **in parallel do**
2 **for** $j = 0$ **to** $T \cdot B - 1$ **step** B **do**
3 **for** $m = 0$ **to** $B - 1$ **do**
4 $v = \mathcal{V}_{i+j+m}$
5 $\mathbf{a}_v^k = \text{AGGREGATE}(\mathbf{h}_u^{(k-1)} \mid \forall u \in \mathcal{N}(v) \cup \{v\})$
6 $v' = \mathcal{V}_{i+j+m+D}$
7 PREFETCH($\mathbf{h}_{u'}^{(k-1)} \mid \forall u' \in \mathcal{N}(v') \cup \{v'\}$)
8 **for** $m = 0$ **to** $B - 1$ **do**
9 $v = \mathcal{V}_{i+j+m}$
10 $\mathbf{h}_v^k = \text{UPDATE}(\mathbf{a}_v^k)$

After the aggregation of each vertex, we prefetch the features needed by a later aggregation with a distance D (Line 9). Since the execution is mainly DRAM bandwidth bound, the L1D fill buffer is often full of pending misses. In such cases, adding excessive software prefetch can instead degrade the performance. Hence, although not shown in the algorithm, we empirically choose to prefetch only the first two cache lines of each feature vector.

We use a just-in-time (JIT) assembler to tailor the aggregation kernel to each layer's specification (e.g., the length of the feature vectors). Dynamically generated kernels use registers more efficiently by using layer-specific constants. They also avoid overhead such as unnecessary boundary checking. The kernels are generated only once during the entire training/inference session because the code is tailored to the model but not the data. Hence, the overhead of the dynamic code generation is amortized [16, 17].

4.2 Layer Fusion

The aggregation phase is irregular and memory-intensive, while the update phase is regular and compute-intensive. Conventionally, a GNN layer first aggregates all vertices, placing little pressure on compute hardware but a lot on the memory hierarchy, and then updates the vertices, flipping the hardware utilization. We propose to fuse the phases to overlap the memory movement with the compute.

Algorithm 2 describes the fusion. We parallelize it by partitioning the output working sets, which are both the aggregation feature matrix $\mathbf{a}^k$ and the output feature matrix $\mathbf{h}^k$. Each parallel task performs a fused aggregation-update on T blocks of B vertices per block in the j-loop (Lines 2-10). It iterates through the assigned vertices with a block size B (Line 2). In each j-loop iteration, it aggregates B vertices in Lines 3-7 and then updates them in Lines 8-10. The AGGREGATE function is equivalent to Lines 4-7 in Algorithm 1.

Memory and compute overlap in two ways. First, within a single thread, the prefetch operations in the j-loop iteration j' may prefetch for a future j-loop iteration $j' + n$. Therefore, during an update phase, the hardware prefetches features needed by a future aggregation phase. Second, when we consider all of the threads, aggregation and update operations may happen simultaneously for different threads.

Figure 4 shows how the two phases on three threads running on separate cores may overlap. Critically, we do not synchronize threads within the parallel loop; thus, we do not force threads to be out of phase with respect to each other, but expect this to happen naturally. In the figure, the aggregation phases on different cores take varied latency, so the subsequent updates start at different times and can overlap with the aggregation phases on other cores.

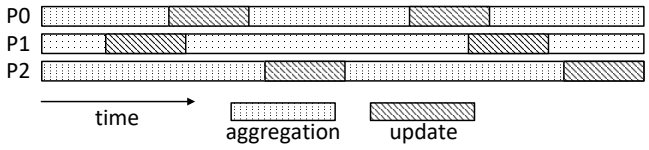
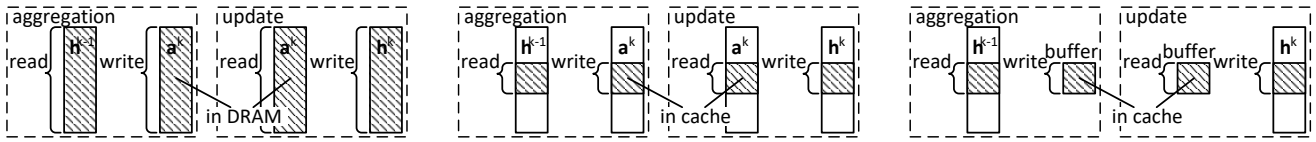


Figure 4: Aggregation and update on three cores P0-P2 can overlap without synchronization.

Besides inducing compute-memory overlap, layer fusion also reduces DRAM traffic and/or footprint, as shown in Figure 5. Without fusion, when $\mathbf{a}^k$ is much larger than the cache, the aggregation writes the entire $\mathbf{a}^k$ to DRAM, and the subsequent update fetches $\mathbf{a}^k$ from DRAM again (Figure 5a). In contrast, the fused implementation produces less DRAM traffic in both training and inference. In each j-loop iteration, the aggregation produces a block of $\mathbf{a}^k$, which is then consumed by the subsequent update. With a proper B, the $\mathbf{a}^k$ block resides in cache between the two phases (Figure 5b). Additionally, in inference, since there is no back-propagation, the fused implementation does not need to keep the entire $\mathbf{a}^k$ for all vertices. Instead, we only need a reusable buffer to hold the block of $\mathbf{a}^k$. We can discard the buffer's content after an update, and use it for the next $\mathbf{a}^k$ block (Figure 5c). In training, the entire $\mathbf{a}^k$ is needed for back-propagation, so this footprint reduction is inapplicable.

4.3 Feature Compression

The feature matrix $\mathbf{h}$ may be moderately sparse because of ReLU and/or feature dropout. Since aggregation uses so much DRAM bandwidth, we can improve performance by avoiding transferring zero-valued elements in the features. However, as discussed, using



(a) The basic implementation writes the whole a^k to DRAM in the aggregation and reads it back in the update.

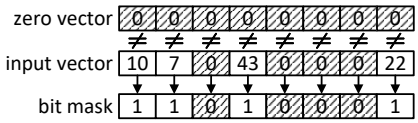
(b) The fused training kernel keeps the a^k working set in the cache between the aggregation and the subsequent update in the same j loop iteration in Algorithm 2.

(c) The fused inference kernel uses a single buffer to hold the a^k block used by the aggregation and the subsequent update in each j loop iteration in Algorithm 2.

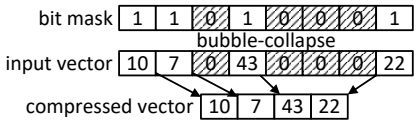
Figure 5: Layer fusion produces less main memory traffic and/or footprint than the basic implementation.

a traditional compressed format such as CSR would be counterproductive for **h**. Therefore, we need a more space/time-efficient compression technique for the purpose.

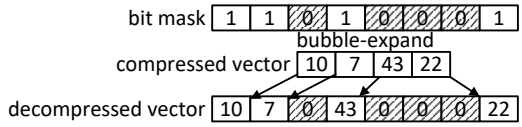
Modern CPU vector extensions such as x86's AVX-512 provide mask-based compression/decompression instructions. The compression instruction takes a vector and a bit mask as input. It uses the set-bits in the mask to select the non-zero elements from the source vector to compress into a contiguous destination vector. The decompression instruction uses a mask to expand elements from a contiguous input vector to a sparse destination vector. Figure 6 shows how we use the above instructions. The example uses a hardware vector length V=8.



(a) Compression step 1: produce a bit mask to mark non-zero elements using vector comparison.



(b) Compression step 2: bubble-collapse the input vector according to the bit mask.



(c) Decompression: bubble-expand the compressed vector according to the bit mask.

Figure 6: Examples of a mask-based (de)compression.

To compress a sparse vector, the first step is to compare it with a vector of zeros, using a vector compare instruction, to produce a mask (Figure 6a). Each of the set bits in the mask indicates the position of a non-zero element in the sparse vector. The second step executes the compression instruction with the mask (Figure 6b). It collapses all the bubbles in the original vector and compacts it to a dense vector. We save both the mask and the compacted vector. To restore the sparse vector from the compacted form, we execute the decompression instruction with the mask generated during compression (Figure 6c). This expands the dense vector back to the sparse vector.

The only meta data of the compression scheme is the mask. Each feature element requires one bit. Hence, if each feature has 32 bits, the space overhead is $1/32 = 3.125\%$ of the uncompressed feature matrix, regardless of the sparsity level. For moderate sparsity, as we expect, this overhead is small. For example, when 32-bit features are 50% sparse, the traffic from reading/writing the features is efficiently reduced by $50\% - 3.125\% = 46.875\%$.

While one could use the compression method to also reduce the memory footprint of **h**, we do *not* do this. It would make compressed feature vectors occupy variable spaces and consequently requires an additional indirection to access them. This harms fast random accesses to feature vectors, which are critical. Therefore, we maintain a constant-sized storage for each feature vector, and simply use only the fraction of it needed by the compressed data. Our purpose in compressing feature vectors is purely to save DRAM bandwidth when reading and writing them, which is achieved with this scheme.

4.4 Temporal Locality Improvement

We also reduce DRAM bandwidth pressure by reducing the reuse distance of each feature vector. In aggregation, each vertex gathers its neighbors' features. If we process two vertices with a common neighbor close together in time, that common neighbor's features will have a small reuse distance, and will likely be cached for the second access. Thus, the processing order of vertices influences the temporal locality.

We access a given vertex's feature vector a number of times equal to the degree of the vertex plus one. Thus, to minimize overall reuse distance, we prioritize the temporal reuse of the features of high-radix vertices. Algorithm 3 describes a method to do this. In the algorithm, L is a collection of $|\mathcal{V}|$ sets. The algorithm builds the sets such that each vertex is placed in only one of these sets, and any vertex placed in set $\mathcal{L}_v$ reads vertex v's feature vector during aggregation. However, not all the vertices reading v's features are guaranteed to be in $\mathcal{L}_v$ because a vertex is only assigned to the set of its highest-degree neighbor. In this way, we build up the sets of high-degree vertices at the expense of those of low-degree vertices.

The algorithm works as follows. Each set is initially empty (Line 1). We populate the sets using a greedy algorithm (Lines 2-7). For each vertex v, we assign it to $\mathcal{L}_{u'}$, where u' is the vertex with the highest degree among $\mathcal{N}(v) \cup \{v\}$. After all vertices are assigned, we generate a processing order $\mathbf{M}$ of vertices in Lines 8-12. During aggregation, vertex $\mathbf{M}_{i+1}$ is processed immediately after $\mathbf{M}_i$.

Algorithm 3: Computing a processing order of vertices to improve the temporal locality in aggregation.

input : graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
output : processing order $\mathbf{M}$
variable : vertex groups $\mathcal{L}$

$\mathcal{L}_v = \emptyset \mid \forall v \in \mathcal{V}$
for $v \in \mathcal{V}$ **do**
  $u' = v$
  for $u \in \mathcal{N}(v)$ **do**
    if $D_u > D_{u'}$ **then**
      $u' = u$
  $\mathcal{L}_{u'} = \mathcal{L}_{u'} \cup \{v\}$
$i = 0$
for $v \in \mathcal{V}$ **do**
  for $u \in \mathcal{L}_v$ **do**
    $\mathbf{M}_i = u$
    $i = i + 1$

The time complexity of the algorithm is $O(|\mathcal{E}|+|\mathcal{V}|)$, which scales well with large graphs. For inference, the overhead can exceed the benefit. However, for training, we reuse graphs in each training iteration, so the cost of the algorithm is easily amortized. Therefore, we only apply this optimization in training.

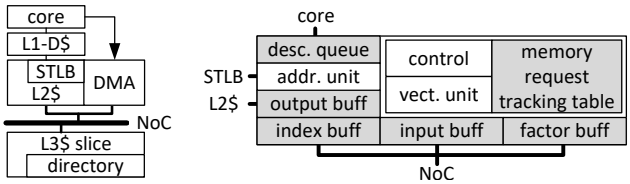
5 HARDWARE ASSISTED AGGREGATION

While the previous software techniques are effective at speeding-up GNN workloads, they still leave performance on the table: processor cores are often stalled waiting for data during aggregation. To address this problem, we propose to offload aggregations from the cores to DMA engines.

In our design, we augment DMA engines that already include gather functionality. Figure 7 shows a block diagram of the enhanced DMA engine. Figure 7a is the top level diagram. It shows that each core is equipped with a DMA engine attached to its L2 cache. The engine takes commands from the core and shares the port to the network on chip (NoC) with the L2. We choose to place the DMA engine near the L2 for several reasons. First, this enables the DMA engine to have easy access to the second-level TLB (STLB). The DMA engine uses virtual addresses and accesses the STLB for address translation. Second, the DMA engine can easily place the results of the aggregation into L2, so that the core can fetch them quickly in the subsequent update phase. Last but not least, the aggregation can benefit from the locality captured in the shared L3.

Our DMA engine works in user space. Modern DMAs such as Intel's Data Streaming Accelerator (DSA) [25] can work in this way.

Figure 7b shows the components in the DMA engine, with storage components shaded. The engine works as follows. The core issues a command by enqueuing a descriptor in user space with a dedicated instruction. To perform an aggregation, the control unit first fetches the indices of the inputs from memory to the index buffer. It then fetches the input data blocks to the input buffer, accordingly. It may also optionally fetch an array of factors to the factor buffer, as will be discussed later. It tracks all memory requests in the Memory Request Tracking Table. Next, it computes



(a) Top level. (b) The DMA engine.

Figure 7: Our enhanced DMA engine.

the reduction in a 4-lane vector unit, holding intermediate results in the output buffer. We choose the width of the vector unit such that the computation does not become a bottleneck. After all inputs are processed, the engine flushes the output buffer to the L2 cache. During the process, the engine performs address translation by looking up the STLB with an address unit.

We opt not to implement the feature compression in the DMA engine. This is because the compression hardware is expensive. Since only the models that use ReLU or dropout benefit from feature compression, the use case does not justify the hardware cost. Next, we discuss the descriptor and the aggregation operation in detail.

5.1 The Aggregation Descriptor

Existing DMA designs use a chain of descriptors to encode a gather operation (Section 2.3). Each descriptor in the chain describes a continuous block of data being gathered. This approach is suboptimal for typical GNN aggregations since the data blocks (in this case, the feature vectors) are relatively small. For example, a 256-element single precision feature vector is only 1KB. Furthermore, rather than describing a set of arbitrarily-sized blocks, we need to only describe a set of fixed-size blocks. Thus, we encode the entire aggregation operation with a single, new descriptor.

Figure 8 shows our proposed 64-byte descriptor and its fields. In the descriptor, *red_op* encodes the reduction operator. *bin_op* encodes the optional binary operator applied on the gathered feature vectors and the elements from a factor array. This is to support the feature processing function ψ described in Algorithm 1. We will discuss the actualization of ψ in Section 5.2. *idx_t* and *val_t* describe the data types of the index array elements and of the input/output elements, respectively.

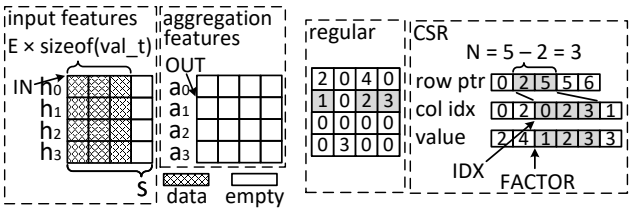
byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0	bytes
red_op	bin_op	idx_t	val_t	# of values in each data block (E)				0
padded size of each data block (S)				# of input data blocks (N)				8
index start address (IDX)								16
input base address (IN)								24
output start address (OUT)								32
factor start address (FACTOR)								40
completion record start address (STATUS)								48
reserved								56

Figure 8: Proposed descriptor for the aggregation operation.

Field E contains the number of elements in each gathered data block. Since the user may want to align data blocks, e.g., to cache line boundaries, the descriptor also includes the S field, which encodes

the padded size of each data block in bytes — which may include some padding bytes. Field N is the number of data blocks being gathered. *IDX* is the starting address of the index array. *IN* is the base address of the memory that contains all the data blocks being gathered. *OUT* is the starting address where the aggregation results are written to. *FACTOR* points to the optional factor array when performing a binary operation. Finally, the DMA engine writes the completion status of each operation to the completion record array *STATUS*. Note that all the addresses are virtual.

Figure 9a shows how the fields are set in an example aggregation. The example is for a graph with 4 vertices. Hence, the adjacency matrix **A** has a dimension of 4×4, and the input feature and the aggregation feature matrices each has 4 rows. In the example, *val_t* is one word. Assume that each vertex has 3 one-word features. Since we want to align each feature vector to 4-word cache lines, each feature vector is padded with one additional element. Figure 9a shows the input feature matrix and the aggregation feature matrix before the operation is performed. We see the values of E and S.



(a) Feature matrices. **(b) Adjacency matrix.**

Figure 9: Descriptor fields of an example aggregation.

Figure 9b shows the adjacency matrix **A** in both regular (left) and CSR (right) formats. It also highlights the data used in the aggregation for the second vertex. In this case, the number of data blocks is $N = 3$, which is the number of non-zeros in the second row. N is easily derived from the CSR row pointers. *IDX* is set to the starting index of the second row in the CSR column indices. If the elements in A are the edge weights used as factors, as will be discussed in Section 5.2, *FACTOR* points to the starting index of the second row in the CSR value array. Finally, as shown in Figure 9a, *IN* points to the starting address of the input feature matrix and *OUT* points to the second row of the aggregation matrix, where the results will be written to.

In this example, the aggregation for the second vertex will read rows h_0, h_2, and h_3 from the input feature matrix, and update row a_1 from the aggregation feature matrix.

5.2 The Aggregation Operation

Algorithm 4 describes the DMA-aggregation algorithm. In the algorithm, $\mathcal{B}$ is the output buffer and d is the aggregation descriptor. Arrays d.IN, d.OUT, d.FACTOR, and $\mathcal{B}$ are indexed in $d.val\_t$ units. For each of the N inputs, the algorithm calculates the index of each of its E elements (Line 4). Note that, for simplicity, the algorithm assumes that $d.S$ is a multiple of sizeof($d.val\_t$). The actual hardware does not have this constraint.

If a feature processing function ψ is present, the algorithm applies the binary operator to the input element and the corresponding

Algorithm 4: DMA-aggregation algorithm.

```
   input  : aggregation descriptor d
   output : output feature vector OUT
   variable : output buffer B
1  B_i = 0 | i ∈ [0..d.E − 1]
2  for i = 0 to d.N − 1 do
3      for j = 0 to d.E − 1 do
4          u = d.S/sizeof(d.val_t) · d.IDX_i + j
5          k = d.bin_op(d.IN_u, d.FACTOR_i)
6          B_j = d.red_op(B_j, k)
7      d.STATUS_i = GET_STATUS()
8  for i = 0 to d.E − 1 do
9      d.OUT_i = B_i
```

factor element (Line 5). This covers a wide range of ψ such as normalization. The host software is responsible for generating the factors. For example, if ψ is normalization, the host should compute the normalization factors and place them in the factor array, while the binary operator is set to "multiply". Finally, we apply the reduction operator to the processed input element and the corresponding buffer element (Line 6). After an input data block is processed, it writes the completion status to the completion record (Line 7). If the status indicates a failure, the remaining operations are aborted. The algorithm omits this case for simplicity. After all input data blocks are processed in the loop (Lines 2-7), the algorithm flushes the buffer to the output in Lines 8-9. Note that, when *red_op* is "sum" while *bin_op* is "multiply", the algorithm essentially performs a dense-matrix sparse-vector multiplication.

The DMA engine fetches the indices, inputs, and optionally the factors from memory. For each address, it sends a request to the home directory of the address. That directory finds the data and replies to the engine. These fetches are parallelized. The number of entries in the index buffer, input buffer, factor buffer, and memory request tracking table determine the maximum number of fetch requests in flight. Besides structural limits, requests also obey dependences. Specifically, we need the indices first, to calculate the addresses of the inputs.

Figure 10 is an example that illustrates how the DMA hardware performs concurrent fetches, interleaves index and input data fetches, and gives priority to indices to make progress. The figure shows a timeline of the occupancy of a 2-entry index buffer (top) and a 4-entry tracking table (bottom). The entries in the index buffer contain fetched indices (idx[i]). They are in Reserved state when they are waiting for the indices to come from memory; they are in Occupied state when the indices have arrived but have not all been used to issue the corresponding input data fetches. Each entry in the tracking table is an outstanding request: when an address is first placed in an entry, the engine issues a request to memory, and when the data returns, the entry is freed. Of course, to calculate the address of an input i, one needs to first fetch idx[i]. The figure assumes that each requested line contains either two index elements or half of a data block.

At time t0, the DMA engine allocates tracking table entries 0 and 1 to fetch indices idx[0:1] and idx[2:3], and reserves index buffer entries 0 and 1 for when they are received. When idx[0:1] is received

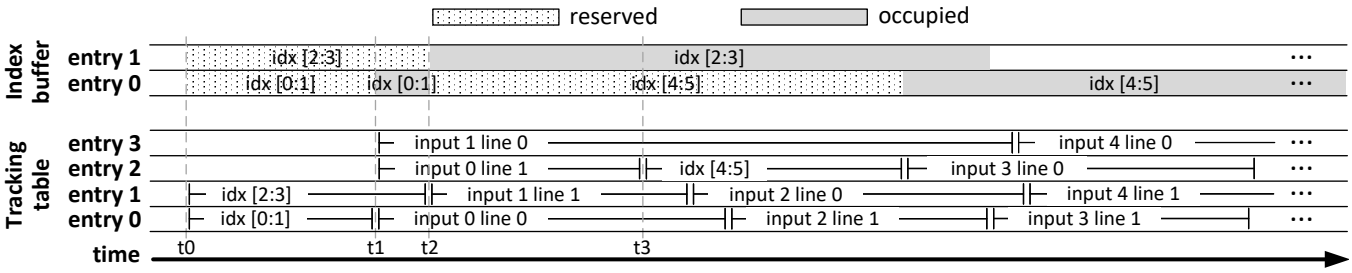


Figure 10: Example timeline of DMA requests.

at t1, the tracking table allocates entries for and fetches line 0 and 1 of input 0, and line 0 of input 1. There is no space for line 1 of input 1 until idx[2:3] arrives at t2. At that time, the tracking table allocates an entry for and fetches line 1 of input 1, and the index buffer frees-up idx[0:1] and reserves an entry for idx[4:5]. As soon as a tracking table entry is freed-up at t3, the table gives priority to allocate an entry for and fetch idx[4:5] over input data. The rest of the timeline proceeds as described.

It is possible that, because of dependences, some tracking table entries are temporarily unused. Rather than underutilizing the memory bandwidth, the DMA engine simultaneously processes a second descriptor.

The output buffer that holds the intermediate results has a limited size. If the size of a feature vector is higher than this limit, the software can break down the aggregation to fit. For example, if the output buffer can fit 256 elements while each feature vector is 400 elements, the software first issues a DMA-aggregation to produce the first 256 elements and then a second one to compute the remaining 144 elements.

The aggregation feature vectors produced by this DMA operation are the input to the next phase: update. To facilitate the pipelining of the two phases, we opt to write the results of the aggregation to L2. When an aggregation begins, the DMA engine prefetches the output lines to L2 in Exclusive mode. After the aggregation results are produced, the engine writes to these lines. If they have not been evicted from L2, we save the latency of the writes missing in L2.

Traditional DMA does not maintain coherence with caches. It is up to the programmer to guarantee data coherence. In our DMA-aggregation, it is not a problem that the DMA engine never stores the fetched inputs in the caches and, therefore, the inputs are unable to observe external invalidations. The reason is that the inputs are read-only by design. Therefore, there are no coherence hazards.

5.3 Software Algorithm Running on the Processor Core

DMA-aggregation is incompatible with feature compression. However, it is synergistic with layer fusion and orthogonal to the locality optimization. Importantly, during a DMA-aggregation, the processor core can work on the update phase, creating a perfect overlap.

Algorithm 5 shows the fused DMA-aggregation and update algorithm that runs on the processor core. The core offloads the aggregation to the DMA engine and performs the update itself. The structure of the algorithm is like the software fusion in Algorithm 2. Each thread dynamically takes a task a time, which processes T blocks of B vertices per block.

Algorithm 5: Pipelined fused DMA-aggregation and update algorithm running on the processor core.

input : graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, input feature matrix $\mathbf{h}^{k-1}$
output : output feature matrix $\mathbf{h}^{k}$
constant : block size B, blocks per task T, number of threads P
variable : descriptors $\mathcal{D}$, ping-pong states $\mathcal{Q}$ and $\mathcal{Q}'$, vertex block offsets $\mathcal{R}$

```
1  Q_t = 0, Q'_t = −1 | t ∈ [0..P − 1]
2  for i = 0 to |V| − 1 step T · B in parallel do
3      t = ThreadID()
4      for j = 0 to T · B − 1 step B do
5          for m = 0 to B − 1 do
6              v = V_{i+j+m}
7              BUILD_AND_ISSUE(D_{t[Q_t·B+m]}, v)
8          if Q'_t ≠ −1 then
9              for m = 0 to B − 1 do
10                 WAIT(D_{t[Q'_t·B+m]})
11             for m = 0 to B − 1 do
12                 v = V_{R_t+m}
13                 h^k_v = UPDATE(a^k_v)
14         R_t = i + j, Q'_t = Q_t, Q_t = (Q_t + 1) mod 2
15 for t = 0 to P − 1 in parallel do
16     for m = 0 to B − 1 do
17         WAIT(D_{t[Q'_t·B+m]})
18     for m = 0 to B − 1 do
19         v = V_{R_t+m}
20         h^k_v = UPDATE(a^k_v)
```

In a given task, the thread alternates between sending the descriptors for the aggregation of B vertices to the DMA engine, and updating B vertices. This is done in a pipeline with ping-pong buffers of descriptors. The ping-pong buffers in thread t, $\mathcal{D}_t$, consist of two batches of B descriptors per batch. In one j-loop iteration (Lines 4-14), the core first builds one batch of B descriptors and issues them to the DMA engine for aggregation (Lines 5-7). The core then waits for the DMA engine to finish the aggregations from a previous j-loop iteration with the other batch of B descriptors (Lines 9-10), and subsequently executes the update with the outputs

from those aggregations (Lines 11-13). In the next j-loop iteration, the roles of the two descriptor batches switch.

To pipeline the operation, each thread keeps track of two ping-pong states: $\mathcal{Q}_t$ and $\mathcal{Q}'_t$. The descriptors indexed by $\mathcal{Q}_t$ are used by the DMA-aggregation, and the ones indexed by $\mathcal{Q}'_t$ are used by the core update. The states of $\mathcal{Q}_t$ and $\mathcal{Q}'_t$ can be either 0 or 1, but $\mathcal{Q}'_t$ is initialized to an invalid value (-1). This is to let the algorithm identify the first j-loop iteration in the first task assigned to the thread. If this is the case (Line 8), the algorithm skips the update phase since there is no aggregation prior to it on this thread. In addition, after all tasks finish, each thread performs the trailing update (Lines 15-20).

To perform the update, a thread needs to know the indices of the vertices processed in the previous batch of DMA-aggregations issued by the same thread. However, such indices cannot always be statically inferred since tasks are dynamically assigned to threads. Therefore, each thread bookkeeps in $\mathcal{R}_t$ the offset of the vertex block $(i + j)$ processed in the aggregation in the current step, to be used as the offset of the vertex block to update in the next step.

6 EXPERIMENTAL SETUP

To evaluate Graphite, we implement the aggregation kernels that incorporates our optimizations with the xbyak JIT assembler [43]. For the update phase, we use GEMM libraries. Specifically, we use Intel MKL for the implementation without layer fusion. With layer fusion, we use libxsmm [22], which is optimized for small matrix multiplications.

Our baseline uses the state-of-the-art DistGNN [36] for the aggregation and MKL's GEMM for the update. DistGNN has recently been incorporated into DGL. We refer to this baseline as *DistGNN*. Because the aggregation can be computed with sparse-dense matrix multiplication (SpMM), we also compare with an approach that uses MKL's SpMM for the aggregation and MKL's GEMM for the update. We call this approach *MKL*.

The software evaluation platform is a 28-core Intel Cascade Lake CPU with AVX-512 vector extensions. Each core has a 32KB L1 data cache, a 1MB L2 cache, and a 1.375MB slice of a non-inclusive shared L3 cache. The core frequency is fixed at 2.7GHz. The DRAM bandwidth is 140.8GB/s. We disable simultaneous multithreading (SMT) and run 28 threads.

We simulate the baseline hardware and the enhanced DMA engine with the Sniper multi-core simulator [8]. The configuration of the simulated machine matches the one used in the software evaluation. On average, the simulated execution time deviates from the native execution time by less than 10%.

We configure the DMA engine per core to have a 2KB output buffer, a 2KB input buffer, a 128B factor buffer, a 128B index buffer, a 32-entry memory request tracking table, and a 32-entry descriptor queue. The DMA engine's storage is 4.5KB. According to Cacti [3] at 22nm, this storage uses $0.051mm^2$ and consumes $2.8mW$ of leakage power. The dynamic energy of a read/write access to the 2KB input or output buffers is $0.011nJ$.

We evaluate Graphite with the full-batch, non-sampled training and inference of the GNN layers from GCN and GraphSAGE. The software evaluation includes 4 medium to large input graphs. Table 3 lists the details of the graphs, including the number of vertices, number of edges, input feature size (H_{input}), as well as the average ($\overline{D_v}$), max ($max(D_v)$), and variance ($\sigma^2(D_v)$) of the vertex degrees. The hardware evaluation is limited to *products* and *wikipedia* due to very long simulation times.

Table 3: List of dataset configurations.

Graph	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\overline{D_v}$	$max(D_v)$	$\sigma^2(D_v)$	H_{input}
products [23]	2.45M	124M	50.5	17.5K	9.20K	100
wikipedia [9]	3.57M	45.0M	12.6	7.06K	1.09K	128
papers [23]	111M	1.62B	14.5	26.7K	927	256
twitter [4]	61.6M	1.47B	23.8	3.00M	3.96M	256

All graphs except *products* are directed. For *products*, the number of edges in the table is twice of its undirected edges. *products* and *papers* have a predefined H_{input}. *wikipedia* and *twitter* only provide their graph structures but not features, so we synthetically set their H_{input} to 256. For all datasets, we use a hidden feature size of 256. We populate the input features with synthetic values. When evaluating the feature compression technique, we randomly set the features to zeros with predefined rates.

7 EVALUATION

7.1 Overall Performance

7.1.1 Software Techniques. We first present the performance of our software optimizations. Figure 11 shows the speedup of our implementations and *MKL* over the *DistGNN* baseline for inference and training. The variations of our implementations are: (i) our algorithm from Section 4.1 for aggregation and the MKL's GEMM for update (*basic*), (ii) the layer-fused implementation from Section 4.2 (*fusion*), (iii) *basic* plus feature compression from Section 4.3 (*compression*), and (iv) the combination of both *fusion* and *compression* (*combined*). For training only, we use *combined* plus the locality optimization from Section 4.4 (*combined+locality* or *c-locality*).

compression, *combined*, and *c-locality* incorporate feature compression. Figure 11 shows their performance when operating on 50% sparse features. This is conservative since, as discussed in Section 2.2, realistic sparsity is often over 80%. Therefore, feature compression may improve the performance more than reported.

Our implementations outperform the baseline significantly. Figure 11a shows the speedup in inference. Performance is determined primarily by memory behavior, which is the same for the two GNNs, so we see similar performance on them. *MKL* is slightly slower than the baseline. *basic* already outperforms the baseline on all datasets. The other variations of our implementation are faster than *basic* on all datasets. *wikipedia* benefits more from layer fusion than from feature compression, while the other datasets benefit more from feature compression than from layer fusion. We will explain reasons behind the difference in Section 7.2.2. *combined* always performs the best. Compared with the baseline of GCN and GraphSAGE, *combined* is 1.72-1.90x and 1.74-1.94x faster, respectively.

Figure 11b presents the speedup in training. The execution time of each implementation includes both forward and backward propagation. Forward propagation is similar to inference, except that when layer fusion is applied, we do not reduce the footprint of $\mathbf{a}^k$

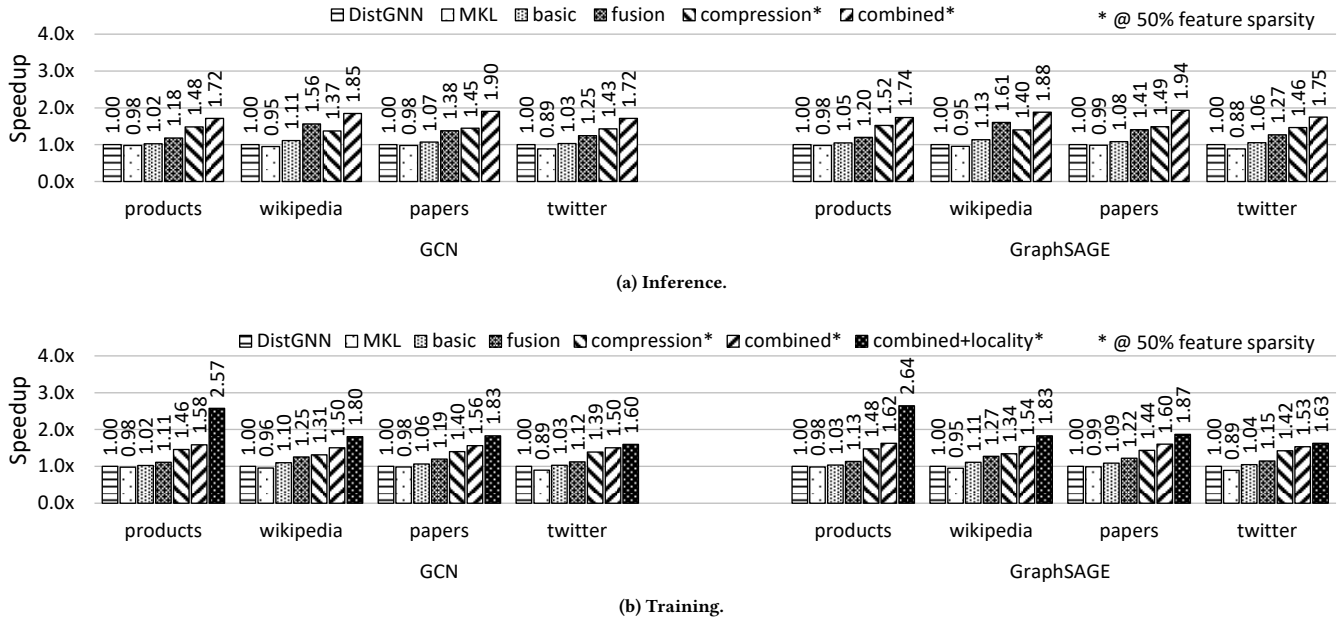


Figure 11: Speedup of MKL and our implementations with different software techniques over *DistGNN*.

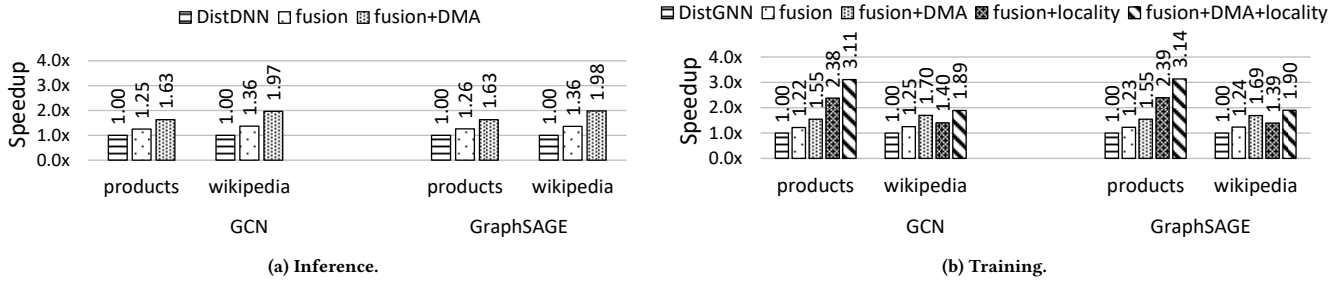


Figure 12: Simulated speedup of different software techniques and the *DMA* support over *DistGNN*.

as discussed in Section 4.2. Backward propagation computes the gradients of $\mathbf{h}^{k-1}$, $\mathbf{a}^k$, $\mathbf{W}^k$, and $\mathbf{b}^k$. It has one more GEMM than the forward propagation. In training, *MKL* is again slightly slower than the baseline. *basic* outperforms the baseline on all datasets. Layer fusion is less effective in training than in inference because in training, it does not shrink the memory footprint of the aggregation feature vectors, and therefore is less effective at reducing memory traffic. Nevertheless, both *fusion* and *compression* perform better than *basic* in all cases. By combining layer fusion and feature compression, *combined* outperforms the baseline in GCN and GraphSAGE training by 1.50x-1.58x and 1.53x-1.62x, respectively. Finally, with the locality optimization, *c-locality* outperforms the baseline by a substantial 1.60x-2.57x and 1.63x-2.64x.

7.1.2 DMA Engine. Now we present the additional impact of adding the DMA engine. Figure 12a shows the simulated speedups of *fusion* and *fusion+DMA* over *DistGNN* for inference. *fusion+DMA* (which we call *DMA*) is the DMA-assisted fusion from Section 5.3. For training, Figure 12b also shows *fusion+locality* (*f-locality*) and *fusion+DMA+locality* (which we call *DMA-locality*), which include the locality optimization.

Figure 12a shows that, in inference, *DMA* outperforms the baseline by 1.63-1.97x and 1.63-1.98x on GCN and GraphSAGE, respectively. In addition, *DMA* is 1.31-1.45x faster than the software-only *fusion*. Figure 12b shows that, in training, *DMA* outperforms the baseline by 1.55-1.70x and 1.55-1.69x on GCN and GraphSAGE, respectively. With the locality optimization, *DMA-locality* outperforms the baseline by a *substantial* 1.89-3.11x and 1.90-3.14x. Further, it is 1.31-1.36x faster than *f-locality*.

Since the performance of GCN and GraphSAGE are similar, we focus on GCN in the rest of the evaluation for simplicity.

7.2 Evaluation of the Software Techniques

7.2.1 Memory Performance Characterization. GNN workloads are often memory bound. We achieve speedup mainly by improving

Table 4: Performance characterization of GCN training. *combined* and *c-locality* are profiled with 50% feature sparsity.

Input Graph	Implementation	Pipeline Slots On		Cycles Bound By				Cycles When
		Retiring	Memory Bound	L2	L3	DRAM Bandwidth	DRAM Latency	L1 Fill Buffer Full
products	DistGNN	9.8%	75.2%	1.5%	2.4%	78.8%	5.3%	100%
	MKL	11.2%	71.8%	0.0%	0.5%	74.4%	5.2%	100%
	combined	18.8%	58.1%	0.8%	1.9%	62.8%	13.4%	100%
	c-locality	28.7%	39.3%	2.7%	4.7%	40.8%	19.1%	31.3%
wikipedia	DistGNN	23.2%	49.0%	2.4%	3.5%	47.9%	8.5%	100%
	MKL	23.1%	47.7%	0.1%	1.2%	45.4%	10.0%	100%
	combined	33.9%	30.6%	1.5%	2.9%	29.8%	12.6%	42.7%
	c-locality	34.1%	30.3%	1.5%	1.9%	28.3%	9.6%	39.1%
papers	DistGNN	13.5%	75.7%	1.5%	3.5%	77.1%	7.2%	100%
	MKL	13.4%	76.7%	0.0%	0.8%	77.1%	7.0%	100%
	combined	24.5%	58.9%	1.0%	1.8%	60.6%	13.1%	100%
	c-locality	28.9%	52.0%	1.3%	3.2%	53.4%	15.3%	93.6%
twitter	DistGNN	12.4%	77.2%	2.4%	3.9%	79.1%	7.5%	100%
	MKL	12.3%	78.8%	0.0%	0.9%	79.2%	8.5%	100%
	combined	19.2%	64.3%	1.1%	2.7%	67.3%	16.7%	100%
	c-locality	22.6%	60.1%	1.4%	3.4%	62.4%	14.9%	100%

memory performance. Table 4 quantitatively demonstrates the memory performance enhancement from our techniques. The table lists key metrics collected with the Intel VTune Profiler that contribute to the performance difference among the *DistGNN* baseline, *MKL*, *combined*, and *c-locality*, in GCN training. The characteristics in inference are similar.

The table shows *Retiring* and *Memory Bound* pipeline slots. *Retiring* is the fraction of pipeline slots utilized by useful work. Increasing it results in higher instructions-per-cycle (IPC). *Memory Bound* is the fraction of pipeline slots stalled due to incomplete memory requests. For our workloads, this is primarily from stalls on loads that miss in cache. In the table, *L2*, *L3*, *DRAM Bandwidth*, and *DRAM Latency* are the fraction of cycles where execution is impacted by L1 miss/L2 hit, L2 miss/L3 hit, DRAM bandwidth limit, and DRAM latency, respectively. *L1 Fill Buffer Full* is an estimate of how often the L1D fill buffers (i.e., the miss status holding registers or MSHR) are fully occupied. This scenario prevents additional L1D-missing requests from being issued. A high value is a symptom that the core is starved for data from memory.

DistGNN and *MKL* exhibit similar traits. They are heavily memory bound on *products*, *papers*, and *twitter*, spending only 9.8-13.5% of the pipeline slots doing useful work, and over 70% of the pipeline slots memory bound. They suffer mainly from the DRAM bandwidth limit. The executions are DRAM bandwidth bound in over 75% of the cycles. On *wikipedia*, the stress on the memory subsystem is lessened. Still, L1 fill buffers are always full on all datasets.

combined incorporates both layer-fusion and feature compression. It is much less memory bound than *DistGNN* and *MKL* on all datasets. The fraction of memory bound pipeline slots is reduced to 30.6-64.3%, and the fraction of useful pipeline slots increases to 18.8-33.9%. Overall, *combined* is less DRAM bandwidth bound than *DistGNN* and *MKL* but more DRAM latency bound than them.

c-locality further optimizes the memory performance. It lowers the fraction of memory bound pipeline slots to 30.3-60.1% and raises the fraction of retiring pipeline slots to 22.6-34.1%. Compared with *combined*, generally more data reuse is captured in the cache, and fewer cycles are stalled by DRAM bandwidth.

As discussed in Sections 4.1 and 4.2, we use software prefetches to exploit the spare L1D fill buffer entries. In general, this approach can leverage all the fill buffers on the large scale graphs (*papers* and *twitter*). However, the fill buffers are still underutilized on the medium scale graphs (*products* and *wikipedia*), where the memory access characteristics are significantly improved such that our software prefetches do not saturate DRAM bandwidth. Although *c-locality* has already achieved impressive performance, free fill buffer entries suggest that adding more aggressive software prefetches may yield additional speedup.

In summary, our techniques are effective and significantly improve the memory performance of GNN workloads.

7.2.2 Layer Fusion. We now investigate the effectiveness of layer fusion. Figure 13 compares *basic*, *fusion* in inference, and *fusion* in forward training on GCN's hidden layers. Hidden layers have the same input and output feature vector length, which is important for this evaluation. The execution time in the figure is normalized to *basic*. The execution time of *basic* is separated into aggregation and update times.

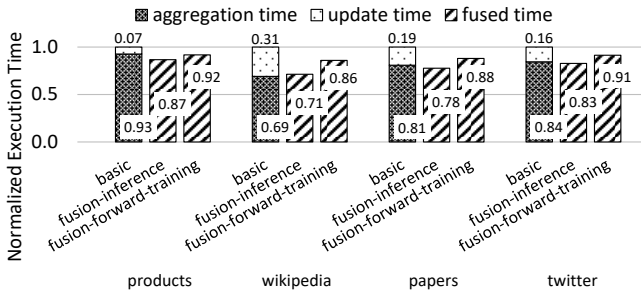


Figure 13: Execution time breakdown of *basic* and *fusion* in inference and forward training on GCN's hidden layers.

Layer fusion provides more benefit when a larger fraction of *basic*'s execution time is spent on update because there is more time to overlap with the aggregation. Since update is 31% of *basic*'s

execution time on *wikipedia*, *fusion* is able to reduce the execution time by 29% in inference. In contrast, *basic* on *products* only spends 7% of the time on update, so *fusion* merely reduces the execution time by 13% in inference.

The amount of DRAM traffic for *fusion* in inference and *basic*'s aggregation is similar — they have the same input and output sizes in our evaluation, and although *fusion* accesses additional data for update, those data are expected to be cache-resident. This gives us the opportunity to assess the effectiveness of the compute-memory overlap by comparing the execution time of *fusion* in inference with *basic*'s aggregation. We see that on all datasets, *fusion* in inference takes similar time as *basic*'s aggregation. This implies that for *fusion*, the compute in update is practically fully hidden.

The difference between *fusion* in inference and *fusion* in forward training is that the latter keeps $\mathbf{a}^k$. The performance difference between the two is from the traffic of writing to $\mathbf{a}^k$.

7.2.3 Feature Compression. We perform a sensitivity study on the performance of feature compression with different levels of feature sparsity. Figure 14 shows the speedup from *compression* over *basic* in GCN at feature sparsities ranging from 10% to 90%.

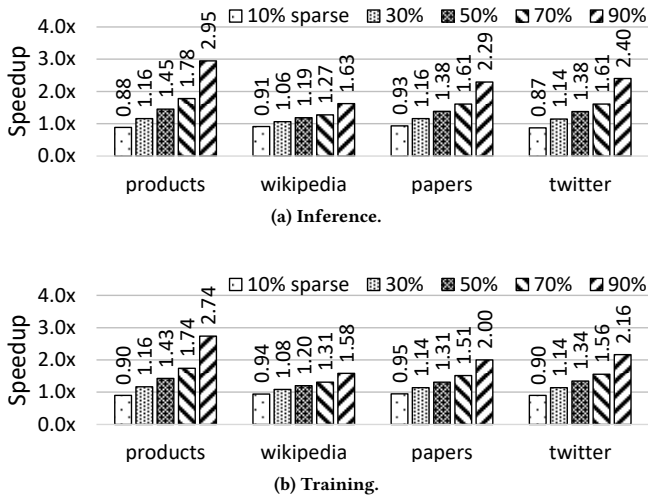


Figure 14: Speedup from *compression* over *basic* at different feature sparsity levels in GCN.

At 10% sparsity, *compression* is slower than *basic*, but at 30% sparsity, it surpasses *basic*. At 90% sparsity, which does appear in training, *compression* is 1.63x-2.95x and 1.58x-2.74x faster than *basic* in inference and training, respectively. Thus, feature compression can be a major source of speedup.

7.2.4 Locality Of Input Graphs. Our temporal locality optimization speeds up the computation on *products* much more than on other datasets (Figure 11b). This phenomenon has two possible explanations. First, the average degree of *products* is 50.5, which is much higher than the average degrees of the other datasets, which range from 12.6 to 23.8. A higher average degree typically implies a higher chance for the vertices to share common neighbors. Because our locality optimization relies on reusing the features of shared neighbors, it performs better on *products*.

Second, some of the input graphs may already embed locality optimization from their sources. We design an experiment to test the hypothesis. First, for each graph, we generate 5 different randomized vertex processing orders. Randomizing the processing order breaks any locality optimization if it exists. Then, we run *combined* with each processing order. Finally, we take the average execution time of the 5 runs and denote it as *randomized*. Figure 15 presents the speedup from *combined* and *c-locality* over *randomized*.

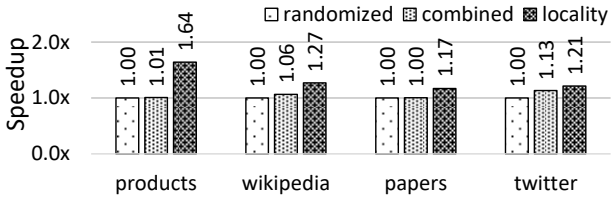


Figure 15: The speedup from *compression* and *c-locality* over *randomized* on GCN training.

randomized performs on par with *combined* on *products* and *paper*, suggesting that the two graphs have not been optimized for locality. On the other hand, *combined* outperforms *randomized* on *wikipedia* and *twitter*. This means that the two graphs possess better-than-average locality already, possibly from pre-processing.

We can view *randomized* as the performance on the graphs with average locality. This experiment shows that our locality optimization improves the locality in all datasets.

7.3 Evaluation of the DMA Engine

7.3.1 Private Cache Access Reduction. One motivation for offloading the aggregation phase to the DMA engine is to reduce the number of low-locality accesses to the private caches — since the input data of the DMA operations bypasses the private caches. In this section, we measure the number of private cache accesses with and without the DMA use. We consider two scenarios: performing only the aggregation or performing the fused aggregation-update. The baseline in the aggregation-only case is *basic*'s aggregation phase. The baseline in the fused aggregation-update case is *fusion*.

Table 5 lists the fraction of private cache accesses eliminated by utilizing the DMA in the two scenarios. In aggregation only, on both datasets, the DMA-aggregation reduces the L1 accesses by over 97%. The core only accesses L1 for building the descriptors or checking for completion. DMA-aggregation also reduces the L2 accesses significantly. The remaining L2 accesses are mainly from the DMA writing the aggregation results. Without DMA, the core issues many L2 accesses to gather the input features, whose proportion in the total L2 accesses correlates with the average degree of the graph. Because *wikipedia* has a lower average degree than *products*, the L2 access reduction on *wikipedia* is lower.

The fused aggregation-update has additional private cache accesses from the update, even with the DMA operation. Consequently, the overall reduction in the number of accesses is lower. Nevertheless, the DMA-assisted implementation still significantly reduces the L1-D/L2 accesses on *products*. The amount reduced on

Table 5: Reduction in private cache accesses thanks to using the DMA engine.

Graph	Aggregation only		Fused aggregation-update	
	L1-D	L2	L1-D	L2
products	98%	97%	43%	36%
wikipedia	97%	89%	19%	12%

wikipedia is lower. This again is due to this graph's low average degree: the average number of memory accesses in the aggregation of a vertex is lower in *wikipedia* than in *products*, while the number of accesses in the update of a vertex is the same in both datasets. Therefore, the ratio between the accesses in the aggregation and in the update is lower in *wikipedia* than in *products*.

7.3.2 Overall Memory System Performance. The use of the DMA engine improves the memory system performance. Specifically, we measure that the L2 miss rate decreases from 20.5% to 2.8% in *products* and from 45.5% to 2.8% in *wikipedia*. Moreover, the processor stall time due to waiting for the memory system decreases from 58.1% to 42.8% in *products* and from 30.6% to 25.7% in *wikipedia*. Note that the memory stall time with the DMA engine includes the time cores spend waiting for the DMA-aggregation to finish. (Algorithm 5 Lines 9-10). Overall, the DMA engine is very beneficial.

7.3.3 Memory Request Parallelism. The maximum number of in-flight memory requests is mainly determined by the size of the Memory Request Tracking Table. We empirically choose the size to maximize the memory/network bandwidth utilization. Figure 16 shows the execution time of the DMA-aggregation on *wikipedia* with different numbers of table entries, normalized to the execution time with 8 entries. The figure shows that the execution time decreases significantly from 8 to 32 entries. After that, we get marginal improvements. Hence, we use 32 entries.

Figure 16: DMA-aggregation time on *wikipedia* with different numbers of Memory Request Tracking Table entries.

8 RELATED WORKS

The alternating update and aggregation phases in GNN processing are challenging to both DNN libraries and graph processing frameworks [19, 50]. DNN libraries target regular computations, like the update phase, but perform poorly on the aggregation phase. Graph processing frameworks, on the other hand, manage irregular memory accesses well but lack support for optimized heavy compute operations in updates. Consequently, GNN-specific software frameworks are emerging. Deep Graph Library (DGL) [49] and PyTorch Geometric (PyG) [11] are two of the prevailing libraries used by researchers. Other frameworks include NeuGraph [35] and AliGraph [57]. Our single socket software optimizations can be incorporated into the compute kernels in these frameworks.

Software algorithms have been proposed to optimize GNN executions on CPUs. FusedMM [38] fuses the sampled dense-dense matrix multiplication (SDDMM) for computing edge messages with the SpMM for computing vertex messages. Their SpMM part performs comparably to MKL. DistGNN [36] is the state-of-the-art in full-batch GNN training on CPU clusters. It also provides single socket optimizations, which we use as our baseline. Dorylus [47] and FlexGraph [48] also focus on distributed training on CPUs, but they do not include single-socket optimizations.

Huang et al. [24] studies the performance gaps in existing GNN frameworks and proposes various software optimizations on GPUs. Their locality-aware task scheduling tackles the same issue as our locality optimization does, albeit for GPUs. Furthermore, ROC [26], P^3 [13], and DGCL [7] focus on distributed training on GPUs.

The community has also explored custom hardware for GNNs. HyGCN [54] uses separate engines for the aggregation and update due to their different behaviors. EnGN [33] treats a GNN as a concatenated matrix multiplication of features, adjacency matrices, and weights. AWB-GCN[15] alleviates the load imbalance due to vertex degree variations. GRIP [28] leverages the abstraction of GReTA [27] to develop a general accelerator for any GNN variant.

SparseTrain [17] and SAVE [18] exploit feature sparsity in software and hardware, respectively, to reduce the compute intensity of DNNs on CPUs. They do not alleviate memory stress nor apply to GNNs. ZCOMP [1] (de)compresses sparse features on CPUs by introducing dedicated instructions.

G-DMA [5] uses scatter-gather DMAs to fetch data in traditional graph workloads, which have different characteristics than GNNs. It uses the descriptor chain model. Our new descriptor design is more efficient for GNNs. In addition, our DMA engine is enhanced with compute capability.

9 CONCLUSION

CPUs are viable platforms for GNNs because of their high availability and high memory capacity. With potentially terabytes of memory, one can perform full-batch GNN computation on real-world large graphs on CPUs, which is impractical on memory-limited devices such as GPUs. However, GNN workloads on CPUs are often highly memory bound, which limits their performance. We propose Graphite: a software-hardware cooperative approach that optimizes full-batch GNN training and inference on CPUs by tackling the memory problem. In software, we fuse layers, compress features, and improve temporal locality. In hardware, we augment the DMA engine to support the aggregation operation. We evaluate Graphite with GCN and GraphSAGE on large graphs. Graphite is effective. Its software outperforms a state-of-the-art GNN implementation by 1.7-1.9x in inference and by 1.6-2.6x in training. Its combined software and hardware speeds up the baseline by 1.6-2.0x in inference and by 1.9-3.1x in training.

ACKNOWLEDGMENTS

This work was supported in part by a gift from Intel Corporation and by NSF grants CCF 1725734, CNS 1909999, CNS 1942888, and CCF 2028861.

REFERENCES

[1] Berkin Akin, Zeshan A Chishti, and Alaa R Alameldeen. 2019. ZCOMP: Reducing DNN Cross-Layer Memory Footprint Using Vector Extensions. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 126–138.

[2] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. 2015. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. arXiv:1512.02595 [cs.CL]

[3] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (June 2017), 25 pages. https://doi.org/10.1145/3085572

[4] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[5] Andrew Bean, Nachiket Kapre, and Peter Cheung. 2015. G-DMA: improving memory access performance for hardware accelerated sparse graph computation. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–6. https://doi.org/10.1109/ReConFig.2015.7393317

[6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[7] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: An Efficient Communication Library for Distributed GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 130–144. https://doi.org/10.1145/3447786.3456233

[8] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 52:1–52:12.

[9] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[10] Jialin Dong, Da Zheng, Lin F Yang, and Geroge Karypis. 2021. Global Neighbor Sampling for Mixed CPU-GPU Training on Giant Graphs. *arXiv preprint arXiv:2106.06150* (2021).

[11] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[12] Alex M Fout. 2017. *Protein interface prediction using graph convolutional networks*. Ph. D. Dissertation. Colorado State University.

[13] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 551–568. https://www.usenix.org/conference/osdi21/presentation/gandhi

[14] Victor Garcia and Joan Bruna. 2017. Few-shot learning with graph neural networks. *arXiv preprint arXiv:1711.04043* (2017).

[15] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. 2020. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 922–936.

[16] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 830–841.

[17] Zhangxiaowen Gong, Houxiang Ji, Christopher Fletcher, Christopher Hughes, and Josep Torrellas. 2020. SparseTrain: Leveraging Dynamic Sparsity in Software for Training DNNs on General-Purpose SIMD Processors. In *Proceedings of the 29th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[18] Zhangxiaowen Gong, Houxiang Ji, Christopher W. Fletcher, Christopher J. Hughes, Sara Baghsorkhi, and Josep Torrellas. 2020. SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[19] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[20] Takuo Hamaguchi, Hidekazu Oiwa, Masashi Shimbo, and Yuji Matsumoto. 2017. Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach. *arXiv preprint arXiv:1706.05674* (2017).

[21] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[22] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 981–991. https://doi.org/10.1109/SC.2016.83

[23] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687* (2020).

[24] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and Bridging the Gaps in Current GNN Performance Optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 119–132.

[25] Intel. 2019. Intel Data Streaming Accelerator Architecture Specification.

[26] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[27] Kevin Kiningham, Philip Levis, and Christopher Ré. 2020. GReTA: Hardware Optimized Graph Processing for GNNs. In *Proceedings of the Workshop on Resource-Constrained Machine Learning (ReCoML 2020)*.

[28] Kevin Kiningham, Christopher Re, and Philip Levis. 2020. GRIP: A Graph Neural Network Accelerator Architecture. *arXiv preprint arXiv:2007.13828* (2020).

[29] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*.

[31] Guohao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. 2019. DeepGCNs: Can GCNs Go as Deep as CNNs?. In *Proceedings of the IEEE/CVF international conference on computer vision*. 9267–9276.

[32] Guohao Li, Chenxin Xiong, Ali Thabet, and Bernard Ghanem. 2020. DeeperGCN: All You Need to Train Deeper GCNs. *arXiv preprint arXiv:2006.07739* (2020).

[33] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. 2020. EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks. *IEEE Trans. Comput.* (2020).

[34] Meng Liu, Hongyang Gao, and Shuiwang Ji. 2020. Towards Deeper Graph Neural Networks. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 338–348.

[35] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 443–458.

[36] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. 2021. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[37] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *arXiv preprint arXiv:1511.06434* (2015).

[38] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. 2021. FusedMM: A Unified SDDMM-SpMM Kernel for Graph Embedding and Graph Neural Networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 256–266.

[39] Morteza Ramezani, Weilin Cong, Mehrdad Mahdavi, Anand Sivasubramaniam, and Mahmut Kandemir. 2020. GCN Meets GPU: Decoupling "When to Sample" from "How to Sample". *Advances in Neural Information Processing Systems* 33 (2020), 18482–18492.

[40] Andres Rodriguez, Wei Li, Jason Dai, Frank Zhang, Jiong Gong, and Chong Yu. 2017. Intel Processors for Deep Learning Training. https://software.intel.com/content/www/us/en/develop/articles/intel-processors-for-deep-learning-training.html

[41] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. 2018. Graph Networks as Learnable Physics Engines for Inference and Control. In *International Conference on Machine Learning*. PMLR, 4470–4479.

[42] Lattice Semiconductor. 2015. Scatter-Gather Direct Memory Access Controller IP Core User Guide.

[43] Mitsunari Shigeo. 2021. Xbyak: JIT assembler for x86(IA32), x64(AMD64, x86-64) by C++. https://github.com/herumi/xbyak.

[44] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (Jan 2016), 484–489. https://doi.org/10.1038/nature16961

[45] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[46] Dean Takahashi. 2018. Gadi Singer interview - How Intel designs processors in the AI era. https://venturebeat.com/2018/09/09/gadi-singer-interview-how-intel-designs-processors-in-the-ai-era/

[47] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 495–514. https://www.usenix.org/conference/osdi21/presentation/thorpe

[48] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. 2021. FlexGraph: A Flexible and Efficient Distributed Framework for GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3447786.3456229

[49] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[50] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.

[51] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[52] Xilinx. 2019. AXI DMA v7.1 LogiCORE IP Product Guide.

[53] Koichi Yamada, Wei Li, and Pradeep Dubey. 2020. Intel's MLPerf Results Show Robust CPU-Based Training Performance For a Range of Workloads. https://www.intel.com/content/www/us/en/artificial-intelligence/posts/intels-mlperf-results.html

[54] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN Accelerator with Hybrid Architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 15–29. https://doi.org/10.1109/HPCA47549.2020.00012

[55] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 974–983.

[56] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81.

[57] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *arXiv preprint arXiv:1902.08730* (2019).