15-418/618
Course Project: Parallelized Graph Convolutional Networks
Report

Liwei Cai (Andrew ID liweicai)
Chengze Fan (Andrew ID chengzef)

May 7, 2019

Abstract

We implemented Graph Convolutional Networks running on multi-core CPU, including a sequential version and a parallelized version using multithreading and SIMD. The experiment on a 16-core machine at AWS showed that our implementation achieves up to 43× speedup over the open source TensorFlow implementation and up to 10× speedup over sequential C++ program in model training, without any loss in accuracy. Our code is available at https://github.com/cai-lw/parallel-gcn.

1 Background

Analyzing and understanding graph data like social networks, citation networks and protein interaction networks are of great practical interest and thus become a popular research topic. In recent years, deep learning have achieved great success in learning from data with relatively simple structures, like tabular data, images and texts. However, designing the appropriate neural network for graph-structured data have long been a challenge.

Graph Convolutional Networks (GCN), proposed by Kipf and Welling (2016) [1], is one of the first successful attempts in applying deep neural networks to arbitrary graphs.

The original paper applied GCN to the task of *semi-supervised node classification*, where a graph and features of each node are given, but only a small portion of nodes are labeled with a class, and the remaining nodes are left for classification. Other algorithms for machine learning tasks on graphs, like label propagation (Zhu and Ghahramani, 2002) and DeepWalk (Perozzi et al., 2014), have been implemented as projects of this course in previous semesters.

Following the success of GCN, many variants of it have been proposed, and similar mathematical principles have been used in designing convolutional networks for manifolds. However, since the focus of this project is on parallel computing rather than machine learning, we decided to follow its original and simplest form.

[1] Also see the author's blog post (http://tkipf.github.io/graph-convolutional-networks) for a high-level explanation.

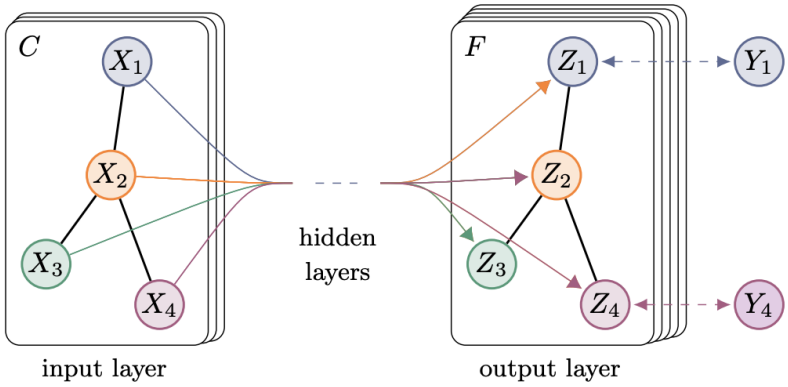

Figure 1: GCN architecture, taken from Figure 1(a) of Kipf and Welling (2016). Schematic depiction of multi-layer Graph Convolutional Network (GCN) for semi-supervised learning with C input channels and F feature maps in the output layer. The graph structure (edges shown as black lines) is shared over layers, labels are denoted by Y_i

2 Problem Statement

2.1 Computational Model of GCN

We extract the computational model of GCN from Kipf and Welling (2016). The whole process is also shown in Figure 1.

There exists a graph that has an adjacency matrix A. We first do the following normalization: $\bar{A} = A + I$, $\bar{D}$ is the diagonal degree matrix of $\bar{A}$, and $\hat{A} = \bar{D}^{-1/2}\bar{A}\bar{D}^{-1/2}$. It can also be expressed in the element-wise form as $\hat{A}_{ij} = \frac{A_{ij}+\delta_{ij}}{\sqrt{(d_i+1)(d_j+1)}}$.

We further assume that each of the nodes in a graph has a feature vector X of length C and output vector Z of length F. Y_i is the label of node i. The forward model for inference can be expressed as:

$$Z = f(X, A) = \text{softmax}(\hat{A}\ \text{ReLU}(\hat{A}XW^{(0)})W^{(1)}) \tag{1}$$

Here, $W^{(0)} \in \mathbb{R}^{C\times H}$ is an input-to-hidden matrix and $W^{(1)} \in \mathbb{R}^{H\times F}$ is an hidden-to-output matrix. These two matrices are the parameters we are supposed to train.

The loss is a cross-entropy error over all labeled examples and the function is shown as follows:

$$L = -\sum_{l\in X}\sum_{f=1}^{F} Y_{lf}\ln Z_{ln} \tag{2}$$

where X is the set of nodes that has labels.

To train the model, we minimize the loss function with

Each training iteration is performed over the labeled nodes and tries to minimize the loss using a gradient-based self-adaptive optimization method called Adam (Kingma and Ba, 2014).

2.2 Scope of the Problem

We implemented a stand-alone C++ program that trains a GCN with the structure shown in Figure 1. The input of the program is a graph, which is made of three files:

- A adjacency matrix, representing the connectivity of the graph
- Features and labels of every node
- Splitting of every node (training, evaluating or testing)

The output of the program is the loss and accuracy on the training set and evaluating set. At the end of training, the program also output weight matrices of the model.

3 Approach

3.1 Sequential Implementation

According to Equation 1, we draw the computational diagram of GCN as in Figure 2. Note that although dropout is not included in the equation, it is still described as an necessary step in (Kipf and Welling, 2016), so we include it in our implementation. Dropout is a simple technique for regularizing neural network that, given a hyperparameter p, randomly zeroes each element with probability p and and scales the rest by $1/(1-p)$ during training, but does nothing in evaluation and prediction.

Following the design of general-purpose deep learning frameworks like TensorFlow and PyTorch, we decompose GCN into several modules where each of them performs a basic operation. Each module is a C++ class with two methods: `forward` for forward computation and `backward` for computing gradients with backward propagation. The network is a sequential concatenation of 8 modules with 6 distinct types. Note that the matrix multiplication in the first layer is a sparse-dense multiplication since input features are sparse in datasets we used.

All modules are implemented in the most straightforward way, following their mathematical definitions. For example, matrix multiplication is implemented as a triply-nested for loop, without using any third-party matrix computation library. That being said, some performance considerations do affect our sequential algorithm design:

- `ReLU` and `Dropout` are implemented in-place, in contrast to the default behavior in TensorFlow and PyTorch where results are returned in a new tensor. In-place operation has a smaller memory footprint and a better cache access pattern.
- `GraphSum` takes the original adjacency matrix without normalization, and compute normalization coefficients on the fly. Since all nonzero elements in the original adjacency matrix is 1, there is no need to store element values, also reducing memory footprint.

From the diagram, we can observe that most computations are independent among nodes, and are therefore trivially parallelizable over nodes. Although GraphSum mixes all nodes up, it can be parallelized over output

nodes just like sparse matrix multiplication. We will discuss in detail how we parallelize our program in the following parts of this section.

3.2 Parallelism Techniques

We used the shared-address model as in Assignment 3 as the starting point. As stated in Section 3.1, we divided the program into separate cascading modules. Because each module has different computational characteristics and dependency, each of them has to be parallelized in different ways. There are also barriers between them.

The big picture of the parallelism is two-fold, first we spawn a couple of threads taking care of independent part of the computation, and second, within each thread, we use SIMD to achieve data parallelism. We show this appoarch in Figure 3. Combining these two techniques, we should be able to utilize the computation power of the CPU to the max extent.

Due to the sparse and non-local nature of graphs, GCN does not benefit as much from GPU as other neural networks, and running on GPU may be less cost-efficient than on CPU in practice. In fact, we also evaluated the speedup of baseline code running on a GPU over a CPU is about 1.7× - 1.8×.

Therefore, we decided to only parallelize GCN on CPU.

Most of parallelism of multi-threading and SIMD is trivially done on the node level with OpenMP pragma:

1. `#pragma omp parallel for schedule(static)`
2. `#pragma omp simd`

However, there are some situations that need extra care. We show the challenges and our approach in the following sections.

3.3 Parallelism on Matrix Multiplication

The forward phase of matrix multiplication is trivial because the coordinate of results in the matrix do not overlap across threads. We just used triply nested loops and apply multi-threading to the outermost loop, as well as SIMD to the innermost loop.

However, during back-propagation, the program needs to accumulate to coordinates in the matrix that do overlap across threads. In order to avoid locks and atomic operations, each thread accumulates gradient in its own copy. After a barrier that ensures all threads finishes calculating local gradients, the program sums up local copies and get the final result.

Sparse matrix multiplication and GraphSum (which is just a specialized sparse matrix multiplication) is parallelized in the same way. However, work imbalance that affects speedup can occur when the distribution of nonzero elements among rows is highly imbalanced. This can happen in real-world datasets like social networks where a few celebrities have millions of connections while most people have few connections. Since we don't find speedup affected by work imbalance at all when testing on our datasets, we simply ignore this potential issue.

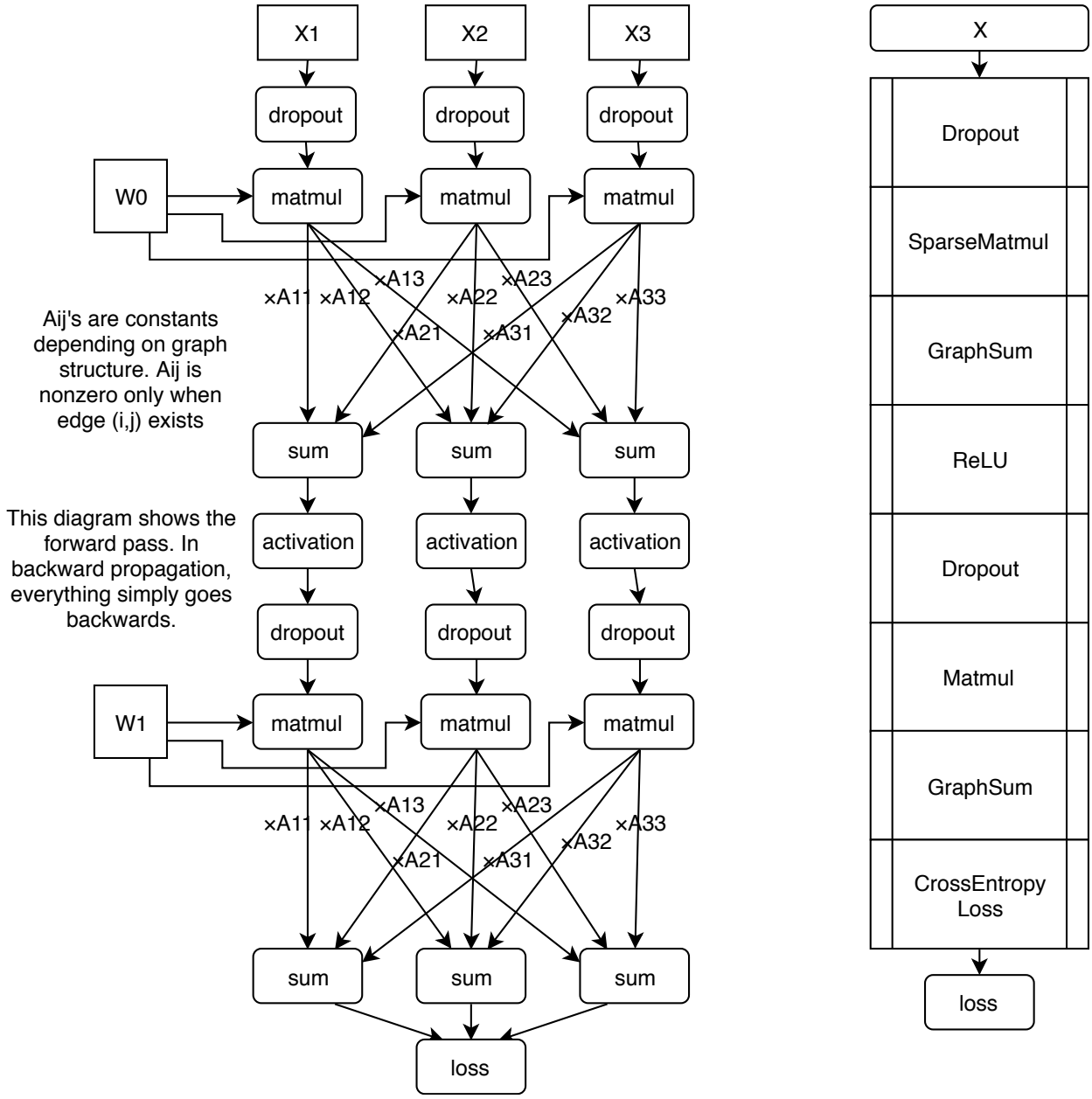

Figure 2: Computation steps of GCN, expressed as data-flow diagram (left) and stack of modules (right). Each vertical column represents a node.

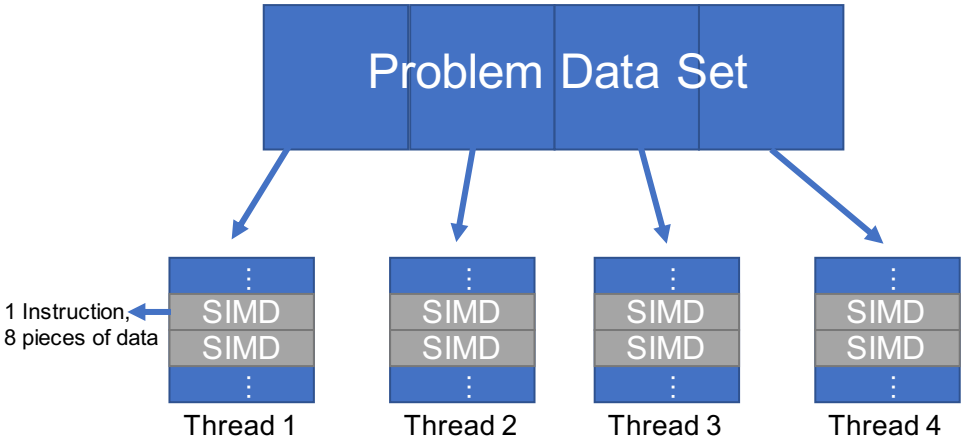

Figure 3: Abstraction of parallelism in each module

3.4 Parallelism on Dropout

The main part of the dropout layer is generating random numbers. We found that generating random numbers ended up as a non-trivial problem when we are trying to parallelize it. The `rand()` function, which uses linear congruential alrogithm, is both too poor in quality and too simple in computation which has an unfair advantage compared to baseline implementations. Therefore, we decided to use an adequately complex and strong algorithm called xorshift128+[2], which takes 128 bits as the seed, uses logical shifts and xor and returns a 64-bit unsigned integer. We takes the lower 31 bit and interpret it as the randomly generated value. We made all random number generator to be this one both in sequential and parallelized version.

We then rewrite dropout layer with SIMD intrinsic instructions. We used an open source SIMD random generator using xorshift128+ algorithm[3].

Second, each of threads needs its own seeds for its generator. Because each thread needs to overwrite the seed for every random number generation, there will be a false-sharing problem. The optimization we made is to pad the seed array to avoid false sharing across different threads.

4 Results

We benchmarked three versions of our C++ code by setting different rules on compilation and linking:

1. sequential: without any multi-threading and SIMD.
2. multi-threading only: with multi-threading and explicitly turing off vectorization optimization in compilers.
3. multi-threading + SIMD: with multi-threading and SIMD turned on.

[2] `http://prng.di.unimi.it/xoroshiro128plus.c`
[3] `https://github.com/lemire/SIMDxorshift`

We have two version of baseline implementation: original implementation in TensorFlow and Deep Graph Library (DGL)[4] with MXNet backend. Both of the baselines can directly run on CPU and GPU, empowered by the general-purpose back-end framework.

Recall that we aim to optimize the time of training. In all experiments, we define the speedup as how much wall-clock time is less than that of the TensorFlow CPU baseline.

4.1 Experiment setup

We evaluated on four datasets mainly on a dedicated c5.4xlarge instance (16 vCPUs, 8 cores with hyper-threading, 32GB memory) on AWS. We evaluated TensorFlow and MXNet GPU performance on a p3.2xlarge instance (1 GPU: Tesla V100, 8 vCPUs, 16 GB GPU memory, 61 GB main memory) on AWS. TensorFlow in our test machine comes from the Deep Learning AMI, which is specially optimized and built for AWS C5 instances.

All C++ programs are compiled with Intel C++ Compiler[5] to fully utilize the SIMD vectorization. We will discuss the choice of compilers in Section 5.2.

We evaluated using the three citation datasets used in the original paper and one additional larger Reddit post dataset. Dataset statistics are summarized in Table 1. According to Kipf and Welling (2016), in the citation network datasets Citeseer, Cora and Pubmed nodes are documents and edges are citation links, while in the Reddit dataset, nodes are posts and edges show if there is at least one user who commented on both posts.

Dataset	Type	Nodes	Edges	Classes	Features
Citeseer	Citation Network	3,327	4,732	6	3,703
Cora	Citation Network	2,708	5,429	7	1,433
Pubmed	Citation Network	19,717	44,338	3	500
Reddit	Online Post Graph	232,965	11,606,919	41	602

Table 1: Dataset Statistics

4.2 Accuracy

It worths to note that the reference implementation released by authors of the GCN paper was written in TensorFlow, which is both explicitly random (e.g. dropout and random initialization) and implicitly unpredictable (e.g. optimization in TensorFlow to accelerate computation and the nature of stochastic gradient descent). Even with exact model hyper-parameter, we cannot get the same model every time.

Meanwhile, some platform-specific and undocumented optimization of mathematical operations in TensorFlow will make our implementation harder to compete with it. This means we may need some optimization that is not related to parallel computing to beat the baseline.

Therefore, as a workaround, we measured the correctness of our program with two methods:

[4] `https://github.com/dmlc/dgl`

[5] `https://software.intel.com/en-us/c-compilers`

1. the convergence speed is similar to that of reference program.
2. the prediction accuracy is similar to that of reference program after convergence.

We show method 1 by comparing cross-entropy loss of both training set and validation set along the training process. The result is shown in Figure 4. We can see that convergence point and convergence speed are very similar to each other.

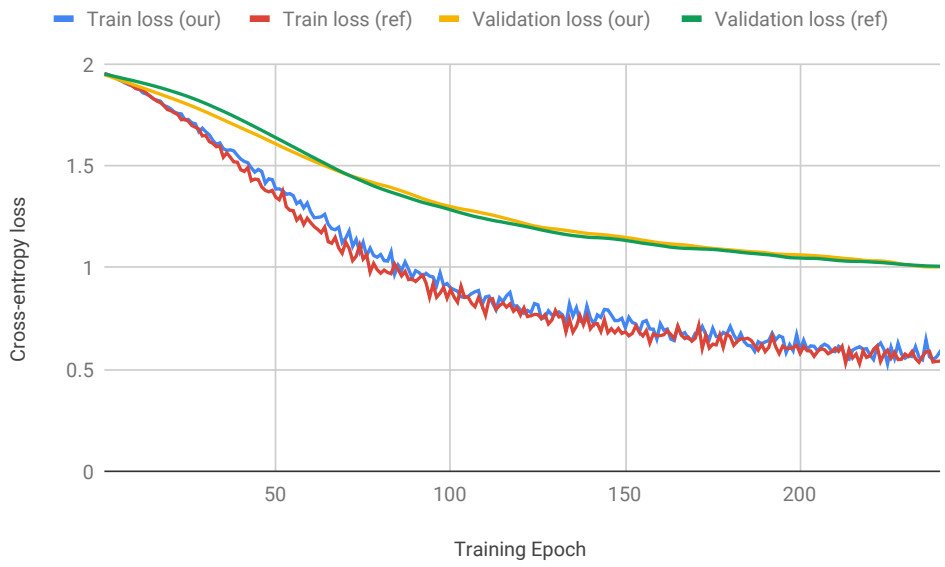

Figure 4: Loss comparison of first 250 training epochs between reference program and ours on Cora

Table 2 shows that our program can reproduce similar prediction accuracy to both reference programs and that reported in the papers.

Statistics	Dataset	Ours	Reference (run)	Reference (reported)
Validation set accuracy	Cora	81.2	80.9	81.5
	Citeseer	71.3	71.5	70.3
	Pubmed	79.0	79.2	79.0
	Reddit	93.3	-	93.0∗

Table 2: Accuracy obtained by running the reference implementation and reported in the original paper are slightly different, so they are both listed.
- Reference implementation cannot read the format of Reddit dataset directly.
∗ Reported in Chen et al. (2018) with batched training

4.3 Overall Performance

The performance speedup of all versions over the baseline of TensorFlow CPU is shown in Figure 5. How the performance scales with different number of threads is shown in Figure 6.

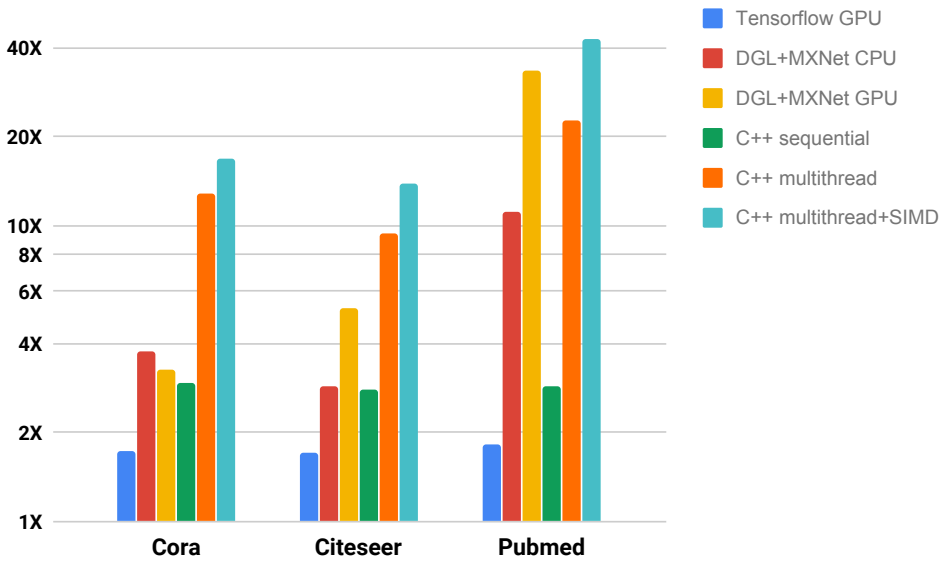

Figure 5: Speedup relative to TensorFlow CPU (baseline). The y-axis is in logarithmic scale. Lacking the Reddit dataset because this graph is not supported by baselines.

we will dig more about the performance in the following sections. In summary, our observations here are shown below.

1. Our program outperforms all general-purpose deep learning frameworks we tried.
2. Our program scales well with more threads, and better with larger datasets, but not well with hyper-threading.

4.4 Scalability of Problem Size

As shown by the yellow line (multi-threading only) in Figure 6, the size of problem can significantly affect the scalability.

For small datasets, the scaling is sub-linear with ≤ 8 threads and the performance becomes worse with 16 threads than 8 threads.

The reason for this is that each module only takes schedules OpenMP multi-threading seperately. OpenMP has a overhead of 100 microseconds or something[6] to spawn threads. According to our micro-benchmark

[6]https://software.intel.com/en-us/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code

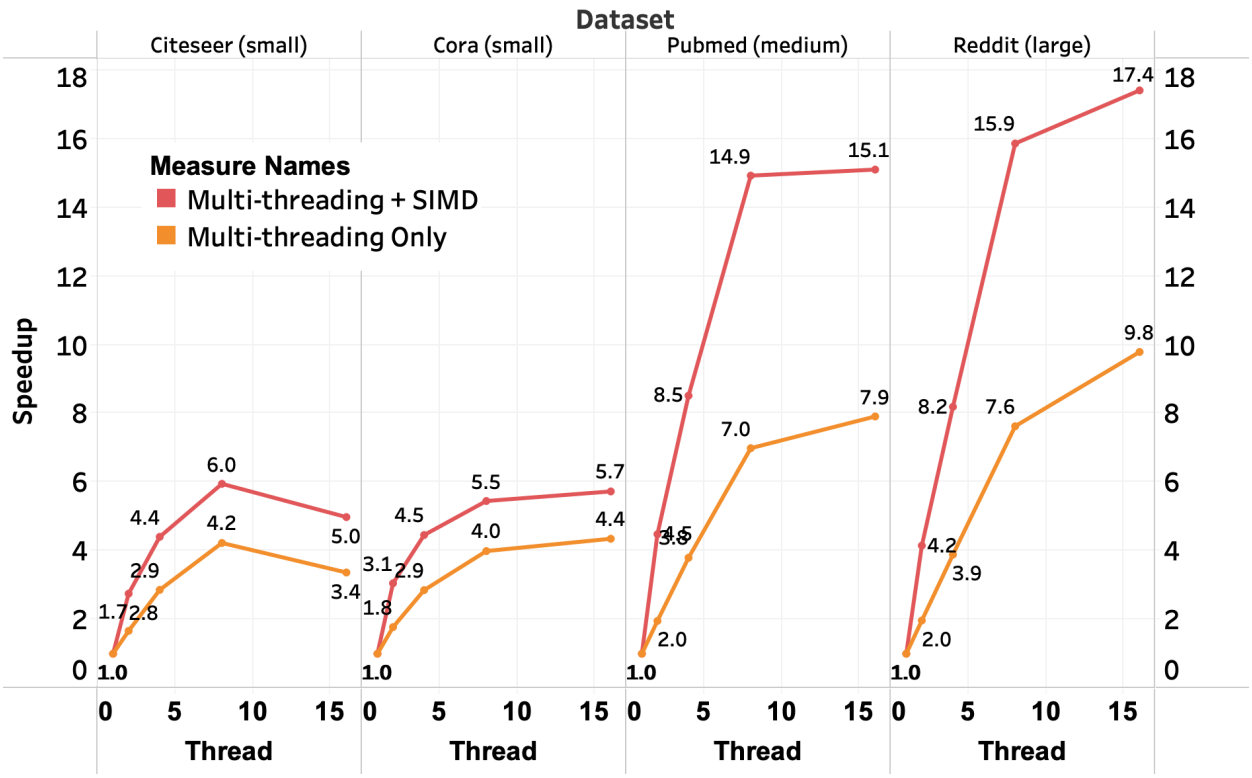

Figure 6: Speedup with different threads in different datasets, relative to the sequential version (without SIMD)

of different modules in Section 4.6, each module takes less than a millisecond for smaller datasets. The Amdahl's Law makes the smaller dataset sub-linear.

For medium to large datasets, the scaling performance is much better than smaller ones, also thanks to the Amdahl's Law.

Hyper-threading does not bring a good speedup here. We speculate the reason is that for smaller dataset, Amdahl's Law dominates while for larger dataset, the program is throughput bound and arithmetic intensity is very high.

4.5 SIMD Performance

Figure 6 already shows that adding SIMD makes the program significantly faster. We further study the speedup brought by SIMD itself (excluding the effect of multithreading) as illustrated in Figure 7. The figure shows that the SIMD speedup drops as number of threads increases. We suggest that the issue here is still Amdahl's law: when the parllelizable part is already fast, making it even faster has less improvement in overall speed, since the sequential overhead remains the same.

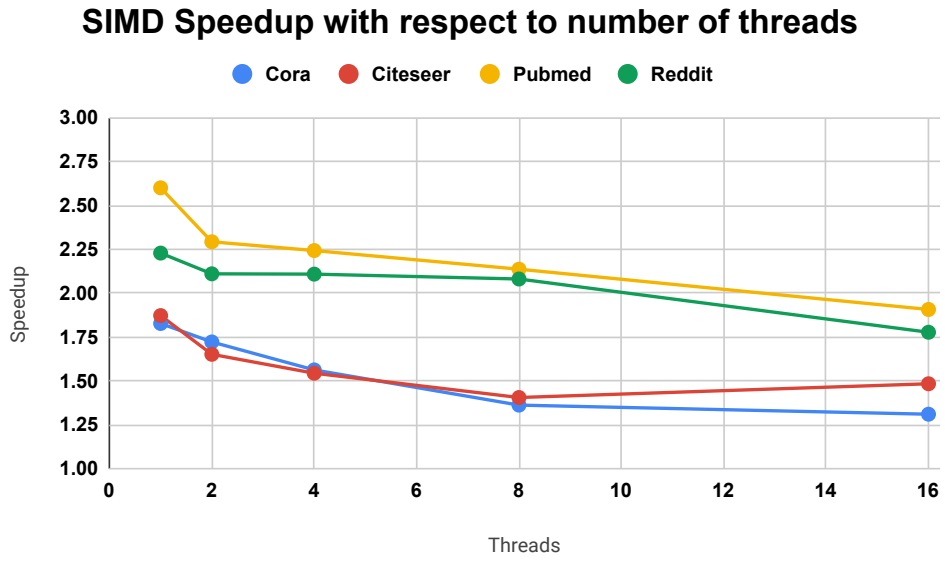

Figure 7: Speedup from multithreading only to multithreading with SIMD, with different number of threads.

Another limiting factor is the difference in resources per CPU core. Since there are less SIMD units per core than scalar units, SIMD program benefits less from hyperthreading because less resource can be shared by multiple threads on the same core. This is reflected in the almost flat line from 8 to 16 threads in Figure 6.

We also measure the SIMD speed when increasing the dimension of the hidden layer, as shown in Figure 8. As the Reddit dataset is trained with 128 hidden units in Chen et al. (2018), larger hidden layers is a realistic workload. Increasing hidden layer size increases the computation of any dataset by a same factor, but the speedup for the two smaller datasets remains almost unchanged, while for the two larger datasets it significantly increases. This confirms our assumption that the major limiting factor is sequential overhead, since for smaller datasets even the increased workload cannot outweigh the overhead.

It is somewhat disappointed that SIMD only brings a speedup of less than $2\times$ when it can in theory process 8 `float` numbers in parallel. All code we apply SIMD are simple arithmetic with no conditional branch. The reason remains unknown to us.

4.6 Scalability at Each Module

To understand the parallelism in different kinds of computation, we measure the time spent on each module, using the largest Reddit dataset as the workload. The time includes both forward and backward computation. "Other" includes all computation not belong to any module, such as updating parameters, copying input data, and calculating accuracy. Note that "other" computations are also parallelized with multithreading and SIMD whenever possible.

Figure 9 shows the percentage of time spent on each module. The ratio remains fairly constant with different number of threads. When using SIMD, `SparseMatmul` and `Dropout` cost significantly less percentage

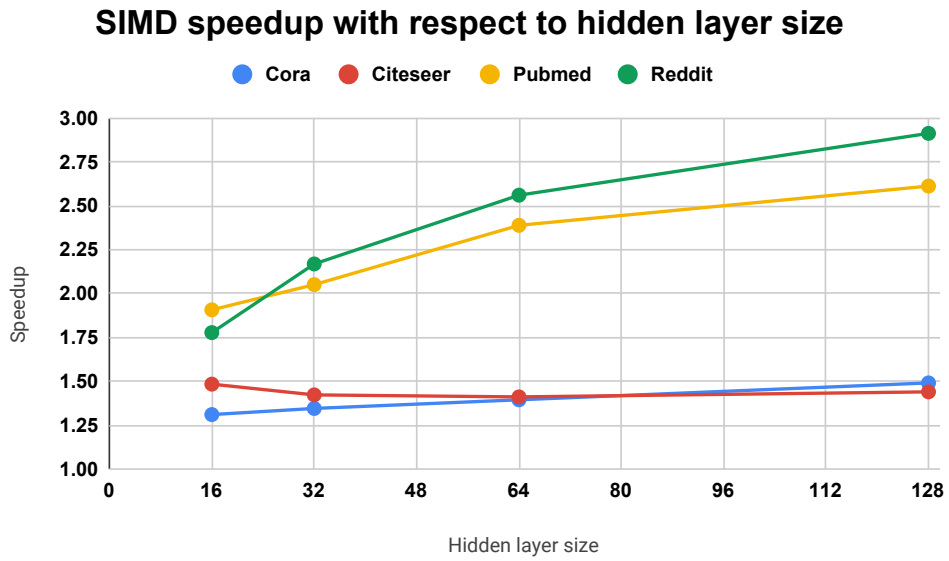

Figure 8: Speedup from multithreading only to multithreading with SIMD, with different hidden layer sizes. All runs use 16 threads.

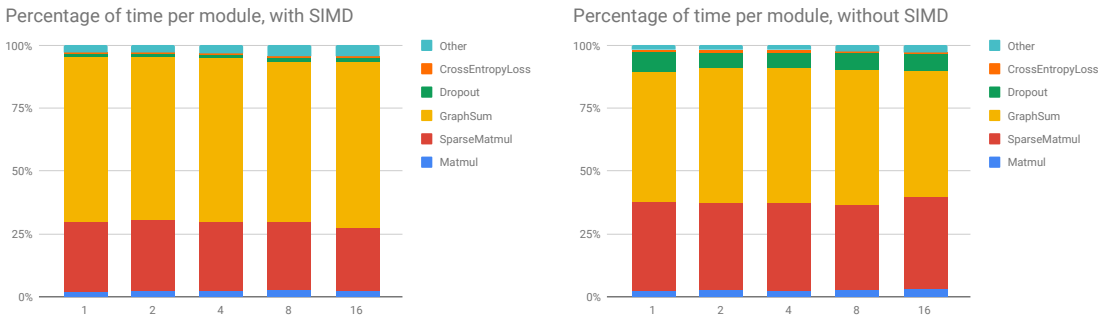

Figure 9: Percentage of time spent on each module, with different number of threads

of time, while `GraphSum` costs more percentage of time.

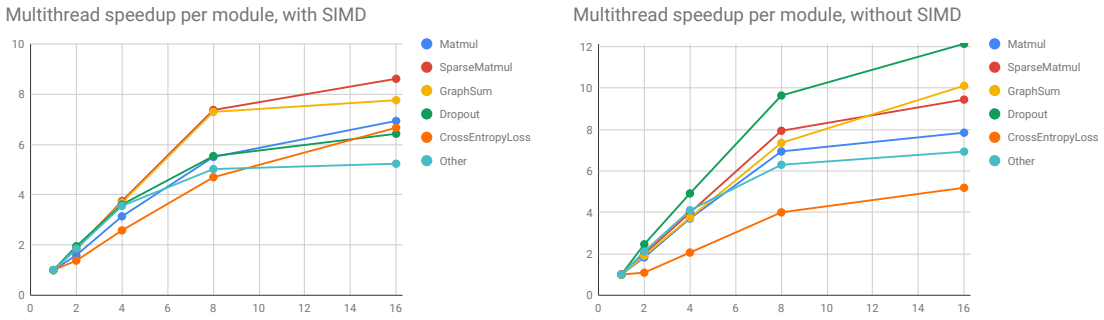

Figure 10: Multithread speedup in each module

Figure 10 shows the multithread scaling behavior of each module. The more computationally intensive module `SparseMatmul` and `GraphSum` achieves almost perfect linear speedup (except for hyperthreading), while simple but less intensive `Matmul` has less speedup due to a relatively larger overhead. `CrossEntropyLoss` has low speedup because of workload imbalance: nodes not in the current split are skipped (e.g. only the training split of nodes are used during training). `Dropout` has normal scaling behavior with SIMD as a less intensive module, but strangely achieves superlinear speedup without SIMD, which we cannot explain.

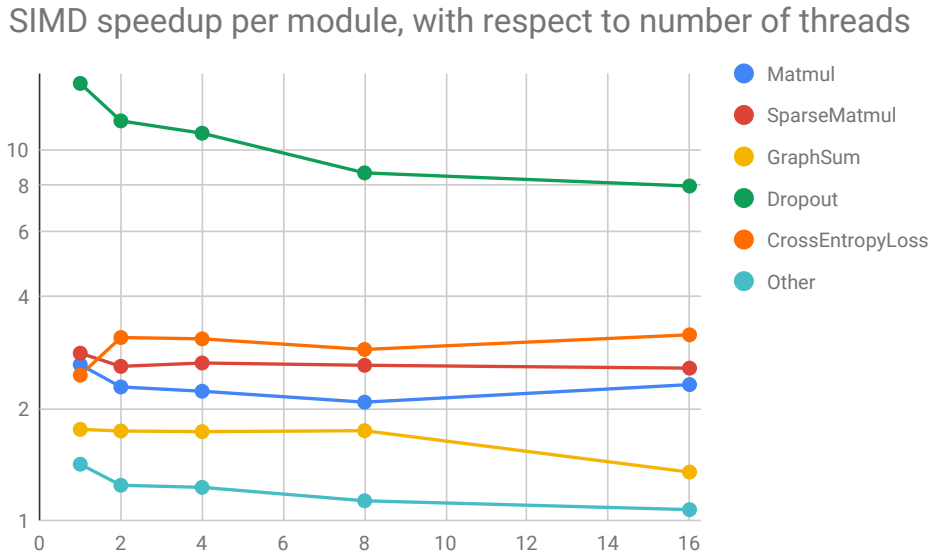

Figure 11: SIMD speedup in each module

Figure 11 shows the SIMD speedup in each module. The astonishingly high SIMD speedup of `Dropout` may be attributed to our manually coded SIMD algorithm. The low speedup in `GraphSum` is likely due to the sparsity of the graph, causing frequent cache miss and large amount of time penalty, which SIMD

cannot help at all. “Other” speeds up the least simply because many “other” functions are not SIMD-ized. The change of SIMD speedup with different number of threads has similar reasons as explained in Section 4.5, that is, limited by sequential overhead and availability of SIMD units.

5 Miscellaneous

5.1 Relationship between Other Graph Computing Systems

Graph computing has become a hot topic in system research. There exists lots of graph computing framework out there. To name a few, GraphLab from Low et al. (2014), Distributed GraphLab from Low et al. (2012), PowerGraph from Gonzalez et al. (2012), Pregel from Malewicz et al. (2010).

The focus of these framework is to provide a domain specific framework and primitives for graph processing. They provide a set of APIs for programmers to build general purpose graph processing algorithms. They also focus a lot on fault tolerance and scalability. Our project just implement a single algorithm, i.e. GCN, and don’t support algorithm modification. Moreover, we don’t put efforts on fault tolerance and scalability, but fine-tuned performance instead.

5.2 Compiler Matters

We originally uses GNU GCC as the compiler and finds out that we have virtually no gain after adding SIMD support. We then tried to use Intel C++ Compile (ICC) to compile our code and found that the sequential and multi-threaded version is a little bit slower than that compiled with GNU GCC, but SIMD is taking exciting effect with the help of ICC, as shown in Table 3.

Compiler	Sequntial	Multi-threading	Multi-threading + SIMD
Intel C++ Compiler	66.6ms	8.64ms	4.41ms
GCC	44.8ms	7.39ms	7.58ms

Table 3: Training time of 1 epoch with different compilers on the Pubmed dataset

In comparison to GCC, ICC has a weaker optimization in scalar codes, but has better integration with OpenMP SIMD and better support for Intel’s intrinsic instructions for vector calculation.

We did not analyze deeper about how different compilers optimize our codes, but we learned at least one lesson that definitely try another compiler when you want your code to run faster.

Division of Work

Liwei Cai studied related papers and open source code, then implemented the sequential algorithm. Chengze Fan parallelized the program and did the majority of writings (including proposal, checkpoint and report). Generally speaking, both project members contributed equally.

References

Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. 2002.