Accelerate Graph Convolutional Network on GPU

Francesca Bettinelli

January 9, 2023

Introduction

The aim of this project is to parallelize a sequential C++ implementation of a Graph Convolutional Network (based on the model developed by Kipf et al.) with CUDA. The idea is to perform all the necessary data transfers from the CPU to the GPU in a preprocessing phase, and then parallelize the main functions responsible for the training of the GCN. The code is then tested against 4 different datasets representing citation networks in order to compute the achieved speed-up.

Hardware info

CPU — Intel(R) Core(TM) i3-7020U CPU @ 2.30GHz, 2304MHz, 2 Core(s), 4 Logical Processor(s).

GPU (provided by Google Colaboratory) — Tesla T4, with the following specifications:

- CUDA cores: 2560
- Global memory: 16 GB
- Shared memory per block: 49 KB
- L2 cache: 4 MB
- Streaming multiprocessors: 40
- Peak memory bandwidth: 320 GB/s

Assessment

Our workflow is structured following the first 3 phases of the APOD design cycle:

1. Assess the sequential code to identify the sections responsible for the bulk of execution time;
2. Parallelize the sequential code with CUDA;
3. Optimize the implementation to achieve better performance.

First, we profile the sequential code by running it on 4 datasets of different sizes:

Dataset	Nodes	Edges	Classes	Features
Cora	2708	5429	7	1433
Citeseer	3327	4732	6	3703
Pubmed	19717	44338	3	500
Reddit	232965	?	41	602

In the output of the profiler (`gprof`) we can see that the highest percentage of execution time is used by the functions declared in `module.h`:

Function	Cora	Citeseer	Pubmed	Reddit
Matmul::forward(bool)	6.85	5.13	3.92	2.14
Matmul::backward()	9.59	3.42	3.92	4.02
SparseMatmul::forward(bool)	9.59	18.80	19.28	26.35
SparseMatmul::backward()	2.74	11.11	8.70	12.77
GraphSum::forward(bool)	9.59	4.27	10.92	27.56
GraphSum::backward()	4.11	2.56	4.78	13.73
ReLU::forward(bool)	8.22	4.27	2.22	1.03
ReLU::backward()	1.37	1.71	1.02	0.47
Dropout::forward(bool)	6.85	6.84	9.39	2.81
Dropout::backward()	0.00	0.00	0.00	0.05
CrossEntropyLoss::forward(bool)	9.59	2.56	1.54	0.34
Total % time	**68.50**	**60.67**	**65.69**	**91.27**

We can notice that, as the size of the dataset increases, `GraphSum::forward(bool)`, `SparseMatmul::forward(bool)`, `GraphSum::backward()`, and `SparseMatmul::backward()` become the most computationally expensive functions.

Other functions worth mentioning outside `module.h` are `Adam::step()` (declared in `optim.h`), which takes up to 4% of execution time, and `xorshift128plus(unsigned long long*)` (declared in `rand.h`), which is used to generate random numbers for weights initialization and dropout and is called up to 4 billion times.

All these functions work on arrays or matrices and involve multiple (nested) loops, so they can be easily parallelized by assigning a chunk of data to each available processor.

Strong scaling is a measure of how, for a fixed problem size, the time to find a solution decreases as more processors are added to a system. Amdahl's law defines the maximum expected speed-up as:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

where P is the fraction of the total serial execution time taken by the portion of parallelizable code, and N is the number of processors. In our case, P can be estimated by the total percentage time computed in the table above, while $N = 2560$ (number of CUDA cores in the Tesla T4). Therefore, we get the following theoretical upper-bound for the speed-up:

	Cora	Citeseer	Pubmed	Reddit
Maximum speed-up	3.17	2.54	2.91	11.41

Parallelization

Data transfer

To limit data transfer overheads, the data should be kept on the GPU as long as possible; in particular, we should avoid transferring intermediate results of computations back and forth from host to device or vice-versa. In the sequential version of the code, the data passing from one GCN layer to the next is done via the private member `std::vector<Variable> variables` in the GCN class. Analogously, we can implement a parallel version with a private member `std::vector<CudaVariable> cuda_variables`, where `CudaVariable` is a class storing pointers to the GPU global memory, initialized via `CudaMalloc` when the constructor is called. This is also convenient for initializing the data and the gradients to zero in a parallel way; in addition, we can also modify the Glorot method for weights initialization by using cuRAND, the CUDA random number generation library.

Therefore, it is possible to transfer all the data from the CPU to the GPU in a preprocessing phase (by calling `GCN::set_cuda_input()` at the beginning of `GCN::train_epoch()`) and then perform all computations needed for training, validation, and testing directly on GPU, provided that the dataset fits into the GPU memory.

The CPU code is already based on single-precision (float) values, which is convenient from a memory usage perspective. Considering the Tesla T4, the global memory (16 GB) is big enough to store the Reddit dataset (~ 3 GB); however, we must be careful if we want to use shared memory (49 KB per block).

`Matmul::forward(bool)` and `Matmul::backward()`

These methods perform (dense) matrix multiplication $A \cdot B = C$; this is a standard problem well-suited for parallelization. The simplest approach consists in assigning to each thread the computation of an element of C. To do so, we can call the CUDA kernel with a sufficient number of 2D blocks; we can try different block sizes, being aware that, for the Tesla T4, 1024 is the maximum number of threads per block, and that it is better to choose a block size multiple of the warp size, which is 32 (this facilitates coalesced access in memory).

However, this approach is not optimal, since it requires redundant accesses to the elements of A and B stored in the global memory; moreover, the access to the elements of B is uncoalesced. Instead, we can assign the computation of a tile of C to each block by loading the corresponding tiles of A and B in the shared memory.

`SparseMatmul::forward(bool)` and `GraphSum::forward(bool)`

These methods perform a sparse-dense matrix multiplication $A_{sp} \cdot B = C$ (A_{sp} is stored in CSR format). The access to the sparse matrix is coalesced, while the access to the dense matrix via sparse indices is random. Like in the dense-dense matrix multiplication case, we can define 2D blocks and assign to each thread the computation of an element of C. However, in this case, we cannot switch easily from a global memory to a shared memory implementation based on tiling because of the randomness of the accesses to B.

`SparseMatmul::backward()` and `GraphSum::backward()`

These methods update gradients, represented by dense matrices, by accessing their elements in random order via a sparse index. This makes parallelization more difficult, as it is required to use `atomicAdd` in order to avoid conflicts among threads in the same block or in different blocks.

`ReLU::forward(bool)`, `ReLU::backward()`, `Dropout::forward(bool)`, `Dropout::backward()`, `Adam::step()`

All these functions contain a simple for loop which can be parallelized by using 1D blocks so that each thread accesses exactly one element. In the CUDA kernel called by `Dropout::forward(bool)`, we use the CuBLAS random number generator to generate a threshold.

`CrossEntropyLoss::forward(bool)`

This function is based on a for loop which can be parallelized in a naive way with 1D blocks. One relevant property we can notice is the presence of several math functions (`fmaxf`, `expf`, `logf`). Therefore, we can try to compile the progam with the `--use_fast_math` flag, which coerces hardware-level functions calls and lowers the precision of the division operation; this can improve performance, but might affect accuracy.

Optimization

After implementing both the global memory and the shared memory versions of the kernel for `Matmul::forward(bool)` and `Matmul::backward()`, we find out that they have comparable execution times. This quite surprising fact might be justified by the existence of efficient L1 and L2 cache management systems in the GPU. Since the usage of shared memory is not convenient, we choose the simplest implementation relying on global memory.

To evaluate the performance of `GraphSum::forward(bool)`, `SparseMatmul::forward(bool)`, `GraphSum::backward()`, and `SparseMatmul::backward()`, we try two different 2D block sizes, 32×32 (reaching the maximum number of threads per block for the Tesla T4) and 16×16. For all the other kernels relying on 1D blocks, we directly use the maximum number of threads per block.

To measure the average execution time (with a fixed number of training epochs equal to 100), we use CPU timers placed before and after each kernel call and we run the code 3-5 times to minimize the effects due external overheads (other programs running on the local CPU, other users accessing the remote Google Colaboratory GPU, etc.).

1. Sequential code

Function	Cora	Citeseer	Pubmed	Reddit
Matmul::forward(bool)	0.931	0.842	4.139	808
Matmul::backward()	1.118	0.894	3.732	1527
SparseMatmul::forward(bool)	1.369	2.789	17.541	9871
SparseMatmul::backward()	0.587	1.264	7.360	4809
GraphSum::forward(bool)	1.460	1.188	8.871	10330
GraphSum::backward()	0.747	0.620	4.301	5195
ReLU::forward(bool)	0.332	0.318	1.920	387
ReLU::backward()	0.249	0.153	1.013	177
Dropout::forward(bool)	1.654	2.347	29.472	3297
Dropout::backward()	0.086	0.037	0.256	17.471
CrossEntropyLoss::forward(bool)	1.800	2.114	7.511	919
Total training time (ms)	**10.901**	**13.746**	**87.971**	**37517**
Training accuracy	0.75	0.72	0.83	0.18
Validation accuracy	0.78	0.78	0.87	0.21
Test accuracy	0.77	0.77	0.86	0.21

2. Parallel code — Results with 2D blocks of size 32×32

Function	Cora	Citeseer	Pubmed	Reddit
Matmul::forward(bool)	0.049	0.050	0.113	9.505
Matmul::backward()	0.251	0.277	0.637	59.044
SparseMatmul::forward(bool)	0.069	0.108	0.446	161
SparseMatmul::backward()	0.057	0.089	0.366	172
GraphSum::forward(bool)	0.569	0.369	1.110	659
GraphSum::backward()	0.303	0.202	0.559	323
ReLU::forward(bool)	0.022	0.023	0.029	2.100
ReLU::backward()	0.011	0.011	0.015	1.062
Dropout::forward(bool)	0.089	0.121	0.608	75.245
Dropout::backward()	0.012	0.012	0.018	1.380
CrossEntropyLoss::forward(bool)	0.144	0.149	0.177	11.684
Total training time (ms)	**2.030**	**2.259**	**6.026**	**1715**
Training accuracy	0.79	0.73	0.82	0.18
Validation accuracy	0.82	0.79	0.88	0.21
Test accuracy	0.80	0.77	0.85	0.21

3. Parallel code — Results with 2D blocks of size 16×16

Function	Cora	Citeseer	Pubmed	Reddit
Matmul::forward(bool)	0.032	0.035	0.065	11.367
Matmul::backward()	0.253	0.277	0.703	66.862
SparseMatmul::forward(bool)	0.055	0.080	0.326	185
SparseMatmul::backward()	0.042	0.065	0.268	217
GraphSum::forward(bool)	0.522	0.358	0.609	391
GraphSum::backward()	0.279	0.198	0.300	191
ReLU::forward(bool)	0.023	0.023	0.031	2.074
ReLU::backward()	0.011	0.011	0.016	1.062
Dropout::forward(bool)	0.089	0.122	0.597	75.068
Dropout::backward()	0.012	0.012	0.019	1.376
CrossEntropyLoss::forward(bool)	0.149	0.155	0.182	11.564
Total training time (ms)	**1.930**	**2.186**	**5.058**	**1387**
Training accuracy	0.79	0.73	0.82	0.18
Validation accuracy	0.82	0.79	0.88	0.21
Test accuracy	0.80	0.77	0.85	0.21

Both parallel codes are much faster than the sequential code. Between the two parallel codes, the one with blocks of size 16×16 provides better execution times for all datasets.

Furthermore, when testing the code with the `--use_fast_math` flag, it turns out that this apparently negligible optimization strongly improves the execution time of `GraphSum::forward(bool)` and `GraphSum::backward()`. The reason might the presence of the following line of code computing the reciprocal of a square root: `float coef = 1.0 / sqrtf((indptr[src + 1] - indptr[src]) * (indptr[dst + 1] - indptr[dst]));`. Plus, accuracy is not affected.

4. Parallel code — Results with 2D blocks of size 16×16 and fast math

Function	Cora	Citeseer	Pubmed	Reddit
Matmul::forward(bool)	0.032	0.034	0.070	14.200
Matmul::backward()	0.252	0.277	0.758	71.347
SparseMatmul::forward(bool)	0.051	0.080	0.355	185
SparseMatmul::backward()	0.041	0.065	0.300	220
GraphSum::forward(bool)	0.203	0.131	0.251	230
GraphSum::backward()	0.108	0.084	0.139	114
ReLU::forward(bool)	0.023	0.023	0.032	2.065
ReLU::backward()	0.011	0.011	0.017	1.062
Dropout::forward(bool)	0.091	0.121	0.595	74.780
Dropout::backward()	0.012	0.012	0.020	1.378
CrossEntropyLoss::forward(bool)	0.149	0.154	0.183	12.390
Total training time (ms)	**1.427**	**1.825**	**4.655**	**1162**
Train accuracy	0.79	0.73	0.82	0.18
Validation accuracy	0.82	0.79	0.88	0.21
Test accuracy	0.80	0.77	0.85	0.21

Final remarks

We can compute the achieved speed-up with respect to the sequential code as:

$$S = \frac{T_{seq}}{T_{par}}$$

Code version	Cora	Citeseer	Pubmed	Reddit
1.	-	-	-	-
2.	5.37	6.08	14.60	21.88
3.	5.65	6.29	17.39	27.05
4.	7.64	7.53	18.90	32.29

We can notice that the speed-up increases with the size of the dataset (number of nodes of the graph); this is coherent with Gustavson's law, which states that one tends to increase the problem size to fully exploit the available computational resources. Also, the experimental speed-up values are higher than the theoretical ones estimated through Ahmdal's law, probably because we underestimated the percentage of parallelizable code.

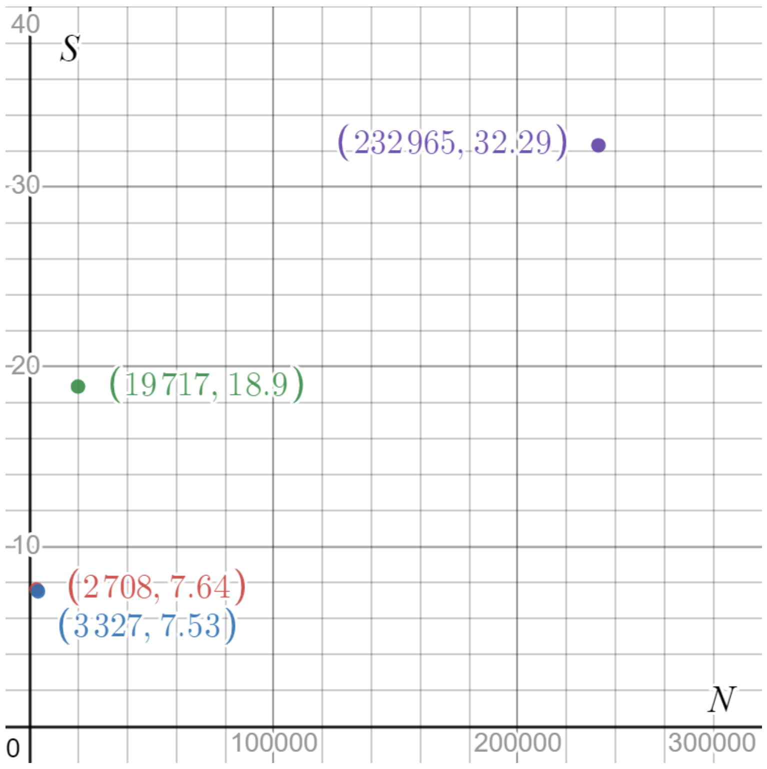

Figure 1: Speed-up vs number of nodes

Further steps for analysis may include benchmarking the parallel code against built-in libraries like cuBLAS or cuSPARSE. Moreover, to avoid the usage of raw pointers and the risk of memory leaks (which was prevented by checking the code with the `compute-sanitizer` tool), we may use libraries such as Thrust, which provides containers for storing data on the GPU.

References

Paper by Kipf et al.: `https://arxiv.org/pdf/1609.02907.pdf`
Tesla T4 specifications: `https://www.techpowerup.com/gpu-specs/tesla-t4.c3316`
APOD design cycle and CUDA best practices:
`https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`