POLITECNICO
MILANO 1863

CUDA implementation of Graph Convolutional Networks

High-Performance Data and Graph Analytics contest

Author: Gurrieri Davide

Professors: Ian Di Dio Lavore, Guido Walter Di Donato

June 2023

Abstract

This report presents the work done for the "High-Performance Data and Graph Analytics" contest at Polytechnic University of Milan.

After a brief introduction, Section 2 discusses a parallel CUDA implementation of Graph Convolutional Networks (GCNs), based on a provided C++ sequential version and developed from scratch without external libraries. The use of CUDA kernels on concurrent streams enhances training and inference speed, resulting in a substantial speedup up to 40.

Section 3 outlines the optimizations made to the model to maximize classification accuracy on popular datasets. The generalization of the number of layers, along with the selection of optimal hyperparameters using a validation dataset, has resulted in attaining satisfactory accuracy performance.

All the source code is available in the GitHub repository [1].

Contents

1 Introduction

1.1 Reference model

Graph Convolutional Networks (GCNs) are a class of deep learning models that extend traditional convolutional neural networks (CNNs) to operate on graph-structured data. GCNs can effectively capture and propagate information about the connectivity patterns in graphs, enabling tasks such as node classification, link prediction, and graph classification.

The contest focus on semi-supervised node classification, using the model introduced by Kipf and Welling in the article [3]. In summary, given a graph with N nodes and adjacency matrix A, the following steps are performed: firstly, the normalization $\bar{A} = A + I$ is applied; then, considering $\bar{D}$ as the diagonal degree matrix of $\bar{A}$, the calculation of $\hat{A} = \bar{D}^{-1/2}\bar{A}\bar{D}^{-1/2}$ follows. This matrix can also be expressed element-wise as $\hat{A}_{ij} = \frac{Aij+\delta_{ij}}{\sqrt{(d_i+1)(d_j+1)}}$. Additionally, it is assumed that each node in the graph has a feature vector X of length C and a label belonging to a set of size F. Labels are One-Hot encoded and denoted by Y_{if}. The original forward model for inference can be expressed as:

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

where $W^{(0)} \in \mathbb{R}^{C\times H}$ and $W^{(1)} \in \mathbb{R}^{H\times F}$ are weights matrices to be trained. The loss function is a cross-entropy error over all labeled examples:

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

where $\mathcal{Y}_L$ is the set of node indices that have labels.

The model is trained using backpropagation and the Adam optimization method.

1.2 Contest presentation

All the rules of the contest are detailed here.

The first part of the contest involves accelerating a sequential version of the model, written in C++, using GPUs and CUDA language. The aim is to enhance the performance of the sequential model, without making any change to the parameters, and to mantain the same accuracy. Further details regarding the sequential version can be found in the GitHub repository [5].

The second part of the contest builds upon the accelerated version of the previous part and aims to enhance the accuracy of the model. For this purpose, any type of modification is allowed.

1.3 Related work

Since article [3] is one of the most well-known papers in the context of graph machine learning, there have already been some attempts to accelerate and improve the original version of the model. In particular, the project [5] provides the sequential version in C++ used in the contest, as well as other experiments with parallelism techniques using OpenMP SIMD but results are limited due to the restricted parallelism achievable with CPUs. In project [4], CUDA code is written to parallelize the C++ version of the aforementioned work. However, the device memory is managed in an unsafe manner and the code contains several imprecisions that degrade accuracy. Moreover, the GPUs parallelism is not fully used, and the achieved speedup is not satisfactory. To overcome these limitations it was decided to develop this project in an independent manner.

2 GPU acceleration

2.1 General design choices

For the management of device memory, in order to have better control over resources, unified memory was not employed. However, this choice required additional effort to avoid allocation issues.

A general class template, `dev_shared_ptr`, was developed from scratch to adhere to the RAII (Resource Acquisition Is Initialization) principle of C++. It serves as a replica of the standard library's `std::shared_ptr` and is responsible for managing device resources: each pointer allocates and frees its own data. The class also provides various other methods that are useful for managing device memory. The implementation is available here and a brief example of the usage is reported in listing 1.

```
int main(int argc, char **argv)
{
    const unsigned N{42};
    std::vector<float> host_val(N, 0.5);
    dev_shared_ptr<float> dev_ptr(N); // Allocate N floats in the device memory
    dev_ptr.copy_to_device(host_val.data()); // Copy host values to device
    my_kernel<<<1,1>>>(dev_ptr.get()); // CUDA kernel that uses the raw device pointer
    return 0;
}
// No need to call cudaFree()!
```

Listing 1: Usage of `dev_shared_ptr`

Another possible approach could have been to rely on constructors and destructors of the classes containing raw pointers to device memory but this strategy is prone to many bugs. For example, an unintentionally created temporary copy of an object can lead to calling the destructor twice: once for the temporary copy and again when exiting the scope of the object. This could result in freeing the memory before object usage and subsequently attempting to free memory that has already been deallocated, causing significant issues.

The same concept has also been used to handle pinned host memory with the class `pinned_host_ptr`, and to manage events and streams with the classes `smart_stream` and `smart_event`.

On the CPU side, polymorphism is extensively exploited in handling the various modules. This is made possible by using pointers to the base class `Module`, which are initialized with pointers to the derived classes. In this case as well, managing raw pointers to memory areas in the free store can be dangerous. Therefore, all the pointers in the sequential code have been replaced with the more modern smart pointers from the standard library.

Regarding the algorithm, it was decided to execute the entire training and evaluation pipeline on the device. This approach helps to avoid unnecessary memory transfers. When an object of the `GCN` class is created, all the variables required for the execution are appropriately allocated and initialized on the device. From that point onwards, the only transfers between the device and the host consist of 3 floats at the end of each forward pass: the loss, the L2 penalty of the loss, and the accuracy. These transfers are only used to print these values and are not strictly essential for the algorithm.

Another crucial decision involves random number generation. The algorithm necessitates random numbers to initialize the weights using the Glorot method. Subsequently, other random numbers are required for the dropout regularization layers in each training epoch. For weights initialization, generating random numbers on the host and then transfer them to the device would be sufficient, since it is an operation performed only once at the beginning. However, the dropout layers require new random numbers in each epoch. To avoid slow host-device transfers, it is necessary to generate random numbers in a parallel way on the device. Therefore, the cuRAND library was utilized. It is important to note that, in order to effectively utilize this library, CUDA directives suggest to assign a unique state (e.g., `curandState`) to each individual thread. This results in a significant memory overhead, as each `curandState` occupies 48 bytes but this approach is necessary to obtain independent, high-quality random numbers. In the implementation, `curandStatePhilox4_32_10_t` was employed as the state type, combined with the `curand_uniform4` function. Although this state is slightly heavier (64 bytes), it can generate 4 independent random numbers simultaneously. Significant memory savings are achieved by reducing the number of required states by a factor 4, leading to improved performance. To further optimize memory usage, instead of allocating a state vector for each variable (`Variable` class) involved in random number generation, a single static vector of sufficient size was allocated. This vector is shared among all the variables and it meets effectively all the necessary requirements.

Another minor but impactful choice was to use the GetPot library for parameter parsing. This approach enables saving the model parameters in text files, allowing easy modification without the need to recompile the code. This has greatly enhanced the agility of the development and testing phase of the code.

2.2 Preliminary optimizations

Several optimizations, which focus on areas unrelated to parallelization, were implemented. Referring to equation 1, it was observed that in the sequential algorithm, the multiplication of $\hat{A}$ with another matrix is performed 4 times in each epoch (2 times in forward and 2 times in backward) by the GraphSum module. The matrix A is stored in a sparse way using the CSR format, retaining only the indices and not the non-zero values since they are all 1. However, the values of the matrix $\hat{A}$ are calculated on-the-fly every time the multiplication is performed in GraphSum. This means that in a training of 100 epochs, the same calculations are repeated 400 times. To improve performance at the expense of higher memory consumption, the calculation of the values of $\hat{A}$ is performed only once at the beginning during parsing and the results are stored on the device.

Staying within the same context, the `CrossEntropyLoss::forward` function counts, each time it is executed, the number of nodes in the training, validation, and test datasets. These information can also be calculated and stored only once at the beginning.

Another optimization relates to the `get_accuracy` method. To ensure numerical stability, the `CrossEntropyLoss::forward` function subtracts the maximum value within each row from all elements in the output matrix. As a result, when calculating accuracy in the `get_accuracy` function, there is no need to iterate over all elements in each row. For further clarification, refer to Listing 2.

```
float truth_logit = output->data[i * params.output_dim + truth[i]];
// given version
for (int j = 0; j < params.output_dim; j++)
      if (output->data[i * params.output_dim + j] > truth_logit) {
        wrong++;
        break;
      }
// improved version
if (truth_logit < 0)
    wrong++;
```

Listing 2: Improvement of `get_accuracy`

Less significant optimizations have been omitted for brevity.

2.3 CUDA implementation

As previously mentioned, the entire algorithm is executed on the device, so all functions from the sequential version have been rewritten as CUDA kernels. Providing a detailed description of each specific kernel is not the purpose of this report (refer to the source code in [1]). This section outlines some general guidelines that were followed during the code implementation.

In almost all kernels, grid-stride loops are used to avoid creating an excessive number of blocks, which can lead to performance slowdowns. The number of threads per block and the number of blocks are predetermined. Each kernel call follows the pattern shown in Listing 3, where `size` represents the data dimension on which the kernel operates.

```
// kernel definition
__global__ void my_kernel(..., const unsigned size)
{
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    for (unsigned i = id; i < size; i += blockDim.x * gridDim.x)
    {
        // do computations
    }
}
// kernel call
const unsigned n_blocks = std::min(CEIL(size, N_THREADS), N_BLOCKS);
```

```
my_kernel<<<n_blocks, N_THREADS>>>(..., size);
```

Listing 3: General kernels structure and call

Regarding the modules, the main operations performed are sparse-dense and dense-dense matrix multiplications, with possible transpositions.

Efforts were made to optimize sparse-dense matrix multiplication, since it is one of the most frequent operations within the algorithm. However, leveraging shared memory with the CSR format was challenging, due to the variable length of non-zero elements in each row of the sparse matrix. Initially, a simple version of the kernel without shared memory was implemented. Subsequently, applying the techniques described in the article [2] seemed promising, since it also provides the source code here. However, the results were not satisfactory and the algorithm only improved insignificantly compared to the simpler version. Consideration was also given to the modification of the storage format of the sparse matrix but it was determined that this change did not yield significant benefits. Therefore, the initial version was retained, since it still performs well overall. This kernel is used by the `SparseMatmul::forward`, `Graphsum::forward` and `Graphsum::backward` methods.

An important note should be added regarding the `SparseMatmul::backward` function. In this case, a multiplication between the transpose of a sparse matrix and a dense matrix needs to be performed. Due to the nature of the CSR format, iterating over the columns of the sparse matrix is not straightforward. Therefore, the algorithm remains the same as the non-transposed version, but the access pattern to the sparse matrix and to the result matrix changes. In this case, multiple threads may concurrently modify the same element in the result matrix, leading to race conditions. Consequently, the use of the `atomicAdd` function, as shown in Listing 4, is crucial.

```
__global__ void sparse_matmul_kernel_backward(const float *a, float *b, const float *c,
    const unsigned *indptr, const unsigned *indices, const unsigned m, const unsigned p)
{
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
#pragma unroll
    for (unsigned i = id; i < m * p; i += blockDim.x * gridDim.x)
    {
        const unsigned row = i / p;
        const unsigned col = i % p;
        const real val = c[i];
#pragma unroll
        for (unsigned jj = indptr[row]; jj < indptr[row + 1]; jj++)
        {
            unsigned j = indices[jj];
            atomicAdd(&b[j * p + col], a[jj] * val); // Critical
        }
    }
}
```

Listing 4: Transpose-Sparse-Dense matrix multiplication

On the other hand, the dense-dense matrix multiplication is a more widespread problem and easier to optimize. It was decided to use tiling, leveraging shared memory, which resulted in excellent performance. In this case, using grid-stride loops is more complicated and thus not applied. It's worth noting that in the backward version, the

Matmul::backward function performs two matrix multiplications: one to update the weights gradients and the other to continue the backward propagation. In the second case, despite requiring the transpose of the second matrix, the same tiling method as the forward version can be used. However, for the first case, applying the same technique is not as useful, since the output matrix only has $H \times F$ elements, with H being the hidden dimension of the two layers and $H, F << N$. Each thread would need to iterate over a very long cycle of N elements. An alternative approach can be used, similar to the sparse matrix multiplication shown in Listing 4, by parallelizing the inner loop of N elements and modifying the access pattern to the matrices. In this case, the benefits of tiling and shared memory are lost, but the GPU resources are utilized more effectively.

Another noteworthy aspect regarding the modules is the utilization of the warp_reduce kernel in the CrossEntropyLoss::forwad and get_l2_penalty_kernel methods. warp_reduce performs an efficient tree-based reduction to sum all the contributions of the loss function.

Concerning the GCN class, all methods that access resources on the device have been parallelized. It is important to highlight a performance-oriented decision made at the expense of memory usage. In each epoch, a forward training pass is performed, followed by a backward pass to update the weights, and finally another forward evaluation pass to get the validation accuracy. During the forward training pass, the Dropout module modifies the input variable by randomly setting some values to zero and scaling the remaining ones. However, the forward evaluation pass require the original input. One alternative would have been to copy the entire input matrix from the host to the device in each epoch. This would result in significant time loss, particularly if the matrix is large. Instead, a copy of the input matrix is kept on the device, remaining unchanged throughout the execution of the algorithm. Thus, before the forward evaluation pass, a kernel is launched to quickly reset the input matrix. The downside is that the copy occupies more memory on the device.

As a general guideline, the use of synchronization between the host and device has been minimized, favoring the use of cudaEvent events.

2.4 Streams

After parallelizing the entire algorithm, the performance was already satisfactory. However, the GPU resources were not fully utilized, especially for relatively small datasets like Cora and Citeseer. Therefore, the dependencies and temporal constraints of the algorithm were studied in order to use multiple concurrent streams.

As mentioned earlier, the algorithm involves two forward phases and one backward phase in each epoch. The forward phases are strictly sequential but can be executed on different streams. The most interesting part is the backward one: in this phase, the algorithm is responsible for weights updates and subsequent optimization steps. As soon as the algorithm is capable of computing one weight matrix, it can do so independently of the other matrix. Thus, it was decided to use 4 streams: the first for the forward training phase, the second for the main backward phase, the third for the update of the second weight matrix during the backward pass and the last one for the forward evaluation phase.

Smart events were extensively used to ensure all temporal constraints were met. Figure 1 provides a diagram illustrating how synchronization was managed.

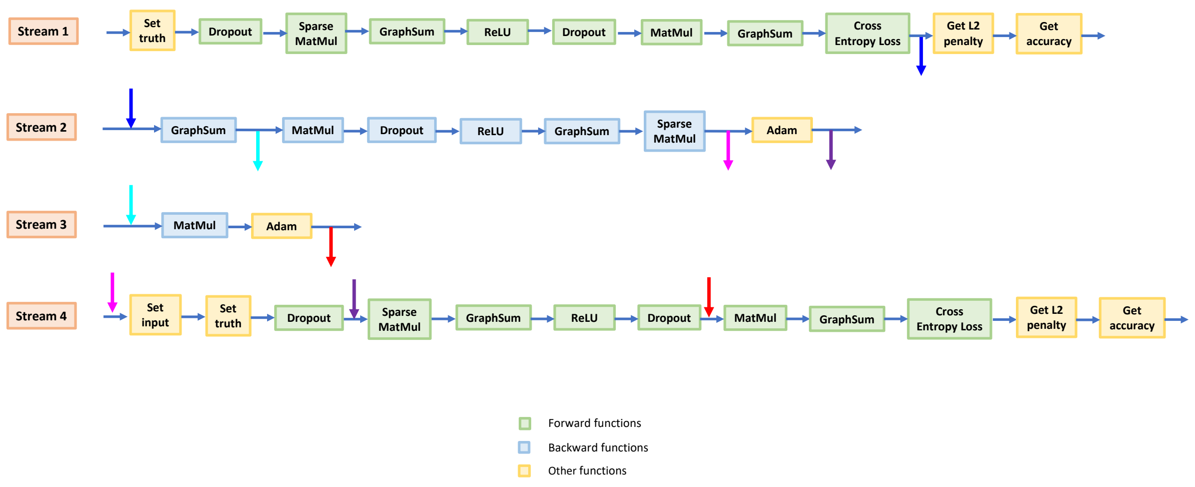

Figure 1: Synchronism diagram of cudaStreams. The large colored arrows indicate the generation (outgoing arrow) and waiting (incoming arrow) of cudaEvents.

After the forward training phase, the main backward phase can begin on stream 2. Immediately after the first GraphSum, the computation of the gradient for the second weight matrix ($W^{(1)}$ in Equation 1) can start in parallel on stream 3. As mentioned earlier, this computation can be computationally intensive, so performing it in parallel with stream 2 fully exploits the GPU. SparseMatMul utilizes the input data modified by dropout. Therefore, right after it completes, the forward evaluation phase can start on stream 4. In this step, it is also necessary to wait for both weight gradients to be updated.

The use of cudaStreams significantly improved performance, especially for smaller datasets. Table 1 presents the performance before and after utilizing streams.

Table 1: Performance before and after using cudaStreams. Times are expressed in milliseconds and refer to the average epoch training time over 100 epochs and 200 execution (20 for Reddit). Used GPU: Colab Tesla T40.

Dataset	**Time Before**	**Time After**	**Gain**
Citeseer	0.4897	0.3127	36.2%
Cora	0.4980	0.2753	44.7%
Pubmed	1.6321	1.4468	11.3%
Reddit	240.217	238.415	0.75%

It is interesting to interpret the results using the statistics in Table 2, which provide some insights into the considered datasets.

The greatest benefits are observed for small datasets like Cora and Citeseer. In these cases, the full computational capacity of the GPU was not used. Now, with streams, the various operations can be overlapped and parallelized. On the other hand, for massive datasets like Reddit, the difference is negligible. Note that the data in Table 1 was obtained by running `performance-gpu` in the `test/` folder of [1].

Table 2: Dataset Statistics

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

Attempts were made to further exploit streams by parallelizing the forward evaluation phase with the subsequent forward training phase. However, the GPU is already fully utilized in the presented version and the new modifications did not lead to further improvements. Moreover, to avoid race conditions on shared resources, the management of synchronizations became significantly complex, since the forward evaluation must always be one step ahead of the training one. The experimental attempts were discarded, but can be found in the "highly_optimized_version" branch of [1].

2.5 CUDA parameters tuning

Another aspect was the selection of optimal CUDA parameters, namely the number of threads per block (`N_THREADS`) and the number of blocks (`N_BLOCKS`). As for the second parameter, it was decided to set it indirectly, leveraging the information of the number of streaming multiprocessors (SM) on the GPU where the program is executed. Thus, `N_BLOCKS` was set as `k * N_SM` and the multiplicative factor `k` was used instead of directly setting `N_BLOCKS`. In this way the choice remains independent of the architecture. To achieve the best performance for each dataset, a test was conducted to explore different combinations of parameter values. It can be replicated by running the `script/tune_cuda_parameters.sh` script. The results for the selected values are presented in Figure 2.

It can be observed that setting a value too small for the multiplicative factor of the number of SM generally degrades performance. The Cora dataset, being the smallest, exhibits more variable behaviors, while other datasets show more stable trends. Interestingly, for the two smallest datasets, a value of 512 threads corresponds to significantly worse performance. Based on these results, the best values for each dataset were selected and are listed in the `parameters/` folder of [1]. The previously reported results in Table 1 were obtained using the parameters selected from this test.

2.6 Performance evaluation

In this final section of the first part are presented the results obtained in terms of performance improvement compared to the sequential version. To ensure a fair test, 50 executions (5 in the case of Reddit) of both the sequential version and the parallel one were performed. The test was conducted using Colab, which integrates a Tesla

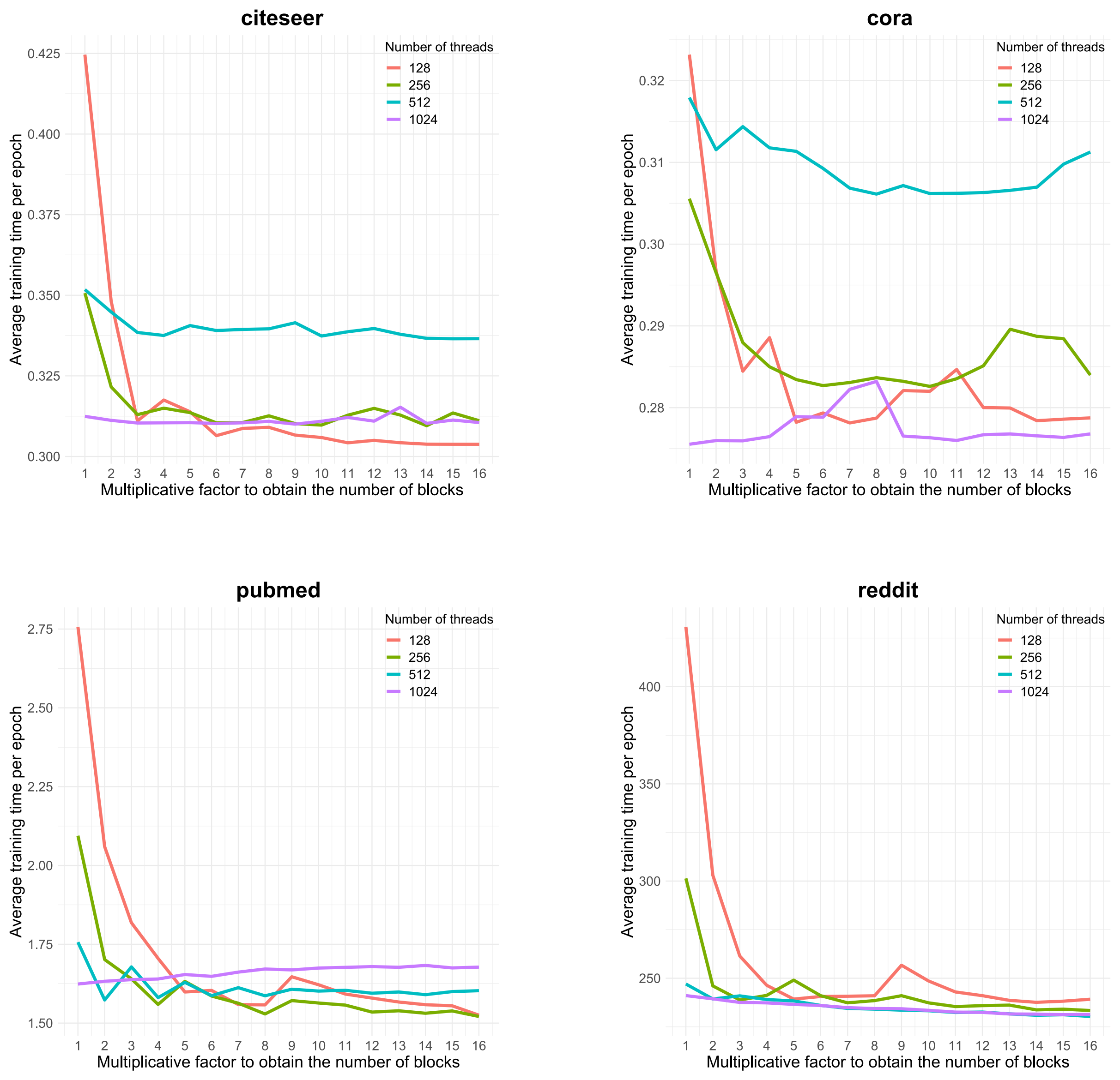

Figure 2: CUDA parameters tuning. On the y-axis the average epoch training time over 100 epochs and 100 execution (5 execution only for Reddit). Times are expressed in milliseconds. Used GPU: Colab Tesla T40.

T40 GPU and an Intel Xeon CPU. To maintain consistency with the other conducted tests, the average epoch training time [ms] over 100 epochs is reported, which is equivalent to considering the total time, as the operations outside the training of an epoch are minimal. The results are presented in Table 3 and were obtained by running the `script/performance_comparison.sh` script in [1].

Table 3: Performance comparison between CPU and GPU

Dataset	**CPU time**	**GPU time**	**Speedup**
Citeseer	7.291	0.325	22.4
Cora	5.002	0.277	18.0
Pubmed	49.827	1.418	35.1
Reddit	9826.111	231.518	42.4

These results are satisfactory and the speedup is directly proportional to the dataset's size. A seed was set for the reproducibility of accuracy results and there was no deterioration compared to the original sequential version.

3 Model optimization

This section presents the modifications made to the initial model to increase its accuracy. After analyzing the available literature, it was decided to keep the same model. The GCN presented in [3], with appropriate modifications, achieves good performance in terms of accuracy. After some preliminary considerations, the structure of the model was generalized, allowing for the insertion of a generic number of layers. Finally, various values of hyperparameters were explored, including the number of layers, so as to find the best combination.

3.1 Preliminary considerations

Initially, an in-depth study of the available datasets was conducted. Citeseer, Cora, and Pubmed are citation networks. Given a dictionary of words for each dataset, each node corresponds to a paper and the features of each node indicate whether a word from the dictionary is present in the corresponding paper: they are binary information. The provided data has undergone normalization: each row of the matrix has been divided by the number of non-zero elements in that row. This normalization introduces additional information. If two nodes contain the word "good," but the first one has more terms than the second one, the word "good" in the first node has a lower weight. Nothing was found in the literature regarding this normalization. In the paper [3], a similar normalization is applied to the adjacency matrix A, but in that case, the purpose is to preserve the feature scale. This normalization did not seem a smart choice and it was decided to use the original binary features.

This had an incredible effect. By running the algorithm with the same parameters as the sequential version, the convergence speed to the minimum point of the loss function increased by a factor ranging from 5 to 6. The accuracy values achieved after 100 epochs in the original version were reached in approximately 15/20 epochs.

It should also be noted that this choice allows for significant memory savings, as it is sufficient to store only the indices and not the feature values. It was concluded that the normalization of features was not appropriate as it introduces information that misleads the model, slowing down its convergence.

This result was included at this point because both versions will be considered in the subsequent steps to study their effects on accuracy.

On the other hand, Reddit is an online post graph, and its features are not binary. Therefore, this modification is not applicable.

Other optimization attempts were inspired by the article [6]. In particular, the use of global clustering coefficients within the matrix $\hat{A}$ was implemented. However, after this modification, the accuracy remained unchanged without any improvements, contrary to the reported findings in the cited paper.

3.2 Model generalization

To avoid being constrained to a fixed number of 2 layers, it was decided to generalize the model, in order to insert any number of layers. The main code modification was the definition of the `GCN::insert_layer` method, which allocates all the necessary resources for a layer during the construction of a GCN object. A generic layer consists of four modules: Dropout, MatMul, GraphSum, and ReLU. Once all the resources are initialized, generalization was straightforward thanks to the polymorphism of the `forward` and `backward` methods. The only part that required careful attention was the generalization of event synchronization. It was decided to maintain the same number of streams, as the GPU resources are already saturated. The main changes were made to backward streams 2 and 3, keeping the main backward flow on stream 2 and performing weights gradients calculations and optimization steps on stream 3. Stream 4 was also modified to receive the new events generated by stream 3. Figure 3 shows an example of the modifications made to streams 2 and 3 for a 3-layer model. The new parts compared to Figure 1 are highlighted in red.

Parameters can be chosen without limitations. For example, if a model with 5 layers is desired, with hidden dimensions of 16, 32, 40, and 18, and dropout parameters of 0.1, 0.2, 0.6, 0.2, 0.1, the following values need to be set in `parameters/` (no spaces between commas):

```
n_layers = 5
hidden_dims = 16,32,40,18
dropouts = 0.1,0.2,0.6,0.2,0.1
```

3.3 Hyper-parameters tuning

After generalizing the model, it was decided to explore different parameter combinations to achieve the best accuracy results. This analysis was conducted using the Citeseer, Cora, and Pubmed datasets since the Reddit dataset would require a much more powerful hardware.

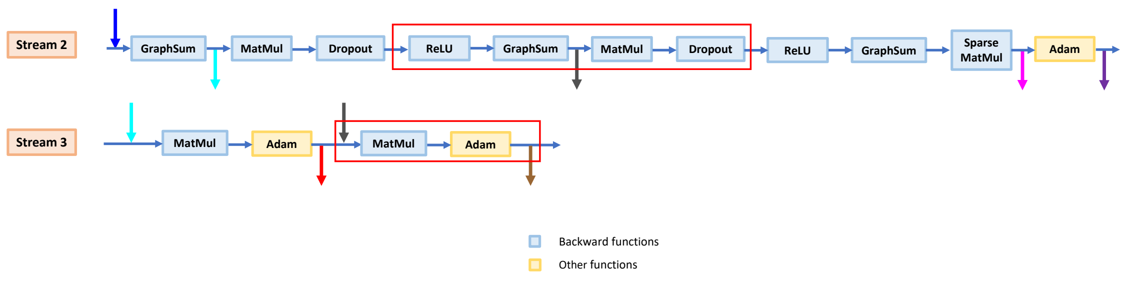

Figure 3: Generalization of the synchronism diagram for streams 2 and 3.

The validation dataset was used to evaluate the accuracy, reserving the test dataset only for evaluating the final selected model.

The random component plays a significant role in accuracy evaluation. In the case of this project, the division between training, validation, and test datasets is assigned a priori and it was decided not to modify it. However, weight matrix initialization and the action of dropout modules are random processes. To present the results and achieve greater reproducibility, a seed was set but the selection of hyperparameters was made leaving randomness and averaging multiple simulations. In this way, the choice was not influenced by particularly lucky or unlucky cases.

To get an idea of which parameters to explore, Appendix B of [3] was referred to. In cited article, the accuracy trend in relation to the number of layers was analyzed, keeping the other parameters fixed. It is observed that an excessive number of layers leads to degraded performance. Therefore, the values 2, 3, and 4 were chosen for exploration. To avoid an explosion of combinations, it was decided to use the same `hidden_dim` value in each layer. As for dropout, the first parameter was left independent of the others since it directly affects the input. The explored values are as follows:

```
n_layers: {2, 3, 4}
dropout: {0.0, 0.2, 0.4, 0.6}
hidden_dim: {8, 16, 32, 64}
early_stopping: = {10}
epochs = 1000
weight_decay: {5e-5, 5e-4, 5e-3}
```

The parameters of the Adam optimization method were left unchanged. The algorithm was run 20 times for each combination of the reported parameters and were calculated the arithmetic mean and standard deviation of the accuracy values. At each iteration, a seed was randomly sampled using the Mersenne Twister generator from the standard library.

This was done for both the original version with normalized input and the binary input version presented at the beginning of this section. Subsequently, a second simulation was conducted, exploring additional parameters based on the results obtained from the first one.

3.4 Accuracy results

In this section, the results of the previous analysis are presented.

The fast-converging version using non-normalized features is able to achieve slightly better results in the case of the Cora and Pubmed datasets, while slightly worse results in the case of Citeseer. Therefore, it was chosen to use it for the first two cases and the original version for Citeseer.

The parameter selection was based on the average validation accuracy, measured at the last epoch before the early stopping mechanism. Variance was also taken into consideration: slightly worse accuracy results but with lower variance were preferred. Additionally, when performance was comparable, simpler models were favored. Table 4 shows the parameter values of the selected models along with their average accuracy and relative standard deviation.

Table 4: Parameters and average validation accuracy over 20 executions of the selected models. L represents the number of layers, H denotes the hidden dimension, $D1$ and $D2$ represent the dropout parameters of the input and the other layers, respectively.

Dataset	**L**	**H**	**Weight decay**	**D1**	**D2**	**Avg. accuracy**	**Sd**
Citeseer	2	16	5e-04	0.6	0.6	81.46	0.495
Cora	2	72	5e-05	0.4	0.2	88.42	0.297
Pubmed	2	8	5e-03	0	0.2	90.01	0.314

The generalization of the model to multiple layers did not lead to improvements in accuracy. With 3 layers, results equal to those in Table 4 can be obtained, but simpler models were preferred. From 4 layers onwards, performance starts to deteriorate, consistent with the findings of [3].

Regarding Citeseer, the best-performing model utilizes dropout normalization in a massive way. This suggests that this dataset is more prone to overfitting. It is interesting to note that the best model for the Pubmed dataset requires fewer parameters. This may seem counterintuitive, considering that Pubmed is the largest dataset among the three. However, this can be justified by the fact that Pubmed has only 3 classes, while the other datasets have 6 and 7 classes and therefore the classification task is much simpler in the case of Pubmed.

A much more powerful GPU would be required for the Reddit dataset but this exceeds the available resources. Therefore, some parameters were manually tested to observe their effect on accuracy. It was observed that even in this case, the model performs better with 2 layers. Regarding the hidden dimension, higher values work better in this case. The selected values are listed in the `parameter/` folder of [1].

Once the best parameters for each graph were selected, the accuracy was independently evaluated using the test dataset. A seed was set to ensure result reproducibility. The table 5 compares the accuracy of the optimized parallel model to that of the initial sequential model, both obtained using the test dataset.

Table 5: Comparison of the original vs improved test accuracy.

Dataset	**Original**	**Improved**
Citeseer	77.0	80.0
Cora	81.9	87.9
Pubmed	85.4	89.6
Reddit	18.4	26.4

To reproduce exactly these same results, the following commands can be executed:

```
make gcn-par-improvement
make run-citeseer
make run-cora
make run-pubmed
make run-reddit
```

The improvement in terms of accuracy was moderate; however, the result can be considered satisfactory. Better results can be found for Reddit, but as mentioned earlier, this is made possible by superior hardware and specific techniques for extremely large datasets. Although it is difficult to make a direct comparison due to the different divisions of the training, evaluation and test sets, the best results reported in the literature for Citeseer, Cora and Pubmed are very similar to those obtained in this project.

References

[1] Davide Gurrieri. parallel-gcn. https://github.com/davide-gurrieri/parallel-GCN, 2023.

[2] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2020.

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

[4] Zongze Li, Yuanhao Jia, Hengda Shi, and Jintao Jiang. cuda-gcn. https://github.com/hengdashi/cuda_gcn, 2019.

[5] Cai Liwei and Fan Chengze. parallel-gcn. https://github.com/cai-lw/parallel-gcn, 2019.

[6] Ihsan Ullah, Mario Manzo, Mitul Shah, and Michael G Madden. Graph convolutional networks: analysis, improvements and results. *Applied Intelligence*, pages 1–12, 2019.