Parallel & Distributed Computer Systems
Exercise 3 – *FGLT* with *CUDA*

Alexandros Athanasiadis – 10006

February 28, 2023

Abstract

In this report I will showcase my implementation of the Fast Graphlet Transform, as described in [1], using the *CUDA* programming model, running on a GPU. I will show the algotithms used to compute the various Graphlets, and then the choices for block/thread distribution and streaming in the GPU environment.

Source code at: `https://github.com/alex-unofficial/cuda-fglt`

1 The Problem

The Fast Graphlet Transform as a problem and its solution is described in [1] in detail. We were asked to implement the calculation of the raw and net frequencies $\hat{d}_k$ and d_k for the first 5 graphlets ($0 \leq k \leq 4$) using *CUDA*.

Considering the adjacency matrix A of a symmetric graph is a symmetric sparse matrix of either 0 or 1 at each position, we can create efficient algorithms for calculating the various frequencies.

Considering that A is stored using the CSC [1] sparse matrix format,

To calculate the raw frequencies:

- $\hat{d}_0$ is trivial.
- for $\hat{d}_1$, the result at each index i is equal to the sum of the elements of row i of A, and since all non-zero elements of A are equal to 1, the sum of the elements is equal to the number of non-zero elements in row i, or `col_ptr[i + 1] - col_ptr[i]` of the CSC format.
- for $\hat{d}_2$, for each row i, find the column indices j of each non-zero element in row i, and add the value of $p_1[j]$ to a sum. Finally subtract the value of $p_1[i]$ to get the result at index i.
- for $\hat{d}_3$, having calculated $p_1 = \hat{d}_1$, for each index i the result is $p_1[i] \cdot (p_1[i] - 1)/2$.
- for $\hat{d}_4$, for each row i, we find all non-zero elements at columns j. Then for each j we calculate the value of A^2_{ij} and add it to a sum. The result at index i will be equal to the sum divided by 2.

Then for the net frequencies, we can use $d_{0:4} = U_5^{-1}\hat{d}_{0:4}$ as is shown in [1].

2 Working with *CUDA*

2.1 Block and Thread distribution

In the *CUDA* programming model each kernel is given a grid of blocks that each contain threads. The problem of distributing these blocks and threads is of critical importance for parallelizing algorithms to run on a GPU.

For $\hat{d}_0$, $\hat{d}_1$ and $\hat{d}_3$ which are one-dimensional problems, we can index the arrays using the formula
`int tid = blockIdx.x * blockDim.x + threadIdx.x` which is standard when converting to one-dimensional index, then perform the required operation at index `tid`. Finally `tid` is updated: `tid += blockDim.x * gridDim.x`

As for $\hat{d}_2$ and $\hat{d}_4$ it is not that simple. Firstly, there are more dimensions to the problems, and this is further complicated by the fact that while threads can communicate with shared memory, blocks cannot, and so we must be careful not to require communication between blocks.

The process for these 2 calculations of the raw frequencies is given below.

[1]Since A is symmetric, it doesn't matter if we use CSC or CSR

2.1.1 For $\hat{d}_2 = Ap_1 - p_1$

for the index i of the result I use `int i = blockIdx.x`, meaning the rows are distributed between the blocks.

The threads of each block are distributed to the non-zero elements of row i, meaning `int j_ptr = threadIdx.x` and `int j = row_idx[j_ptr]`.

The thread then adds the value of p_1 at index j to a running sum, and is updated: `j_ptr += blockDim.x`

Each thread then adds the total sum to a shared memory array, which after all threads are finished is reduced to the total sum of all the elements, meaning this is the result of Ap_1 at index i. Finally the head thread writes the result to the output array, subtracting $p_1[i]$.

Then i is updated: `i += gridDim.x` and the process repeats.

2.1.2 For $\hat{d}_4 = (A \odot A^2) \cdot \mathrm{e}/2$

The distribution of blocks and threads is similar for $\hat{d}_4$.

Each row i is distributed between blocks: `int i = blockIdx.x`

Then the non-zero elements of the row i are distributed between threads
`int j_ptr = threadIdx.x` and `int j = row_idx[j_ptr]`

Each thread must then add to the sum the value of A^2 and index (i, j)

The rest of the process is similar to the one above.

2.2 Streaming

There is a significant overhead when transferring data from the CPU to the GPU and vice versa. for this reason we can use streaming in an attempt to hide the data transfer costs.

For example we might launch a kernel to do some operation while at the same type copying some unrelated data to the GPU.

In this program there is some potential for concurrency. for example, $\hat{d}_0$ is always equal to 1, and so it depends on none of the data, and so we can execute the kernel that "computes" it concurrent to the data transfer.

Furthermore, $\hat{d}_1$ and $\hat{d}_3$ only depend on the `col_ptr` array, so after this has transferred to the GPU, we can start the transfer of the `row_idx` array while concurrently running the kernels to compute $\hat{d}_1$ and $\hat{d}_3$.

Some additional logic is added using events to ensure that no kernel is launched without the data it needs having first been completed.

3 Results

Using the Aristotle HPC Cluster I was able to test the performance of the implementation on the 'gpu' partition, which has an *Nvidia Tesla P100*.

I tested the result on 4 matrices from the *SuiteSparse Matrix Collection*,

Matrix Name	Columns	Non-zero Elements
`auto`	448695	6629222
`great-britain_osm`	7733822	16313034
`delaunay_n22`	4194304	25165738
`com-Youtube`	1134890	5975248

3.1 Comparison to CPU

One way to test the performance of the program is to compare the total execution time to that when ran on the CPU.

Table 1a. shows the relationship between the Num. of *CUDA* Blocks used and the execution time of the program, while Table 1b. does the same for the Num. of threads per *CUDA* Block. The CPU time for each matrix is also given as a comparison metric.

From these results it is evident that a significant speedup is possible using a GPU to compute the *FGLT*, depending significantly on the grid and block size. In general it seems that increasing the num. of blocks in the grid increases performance, while increasing the number of threads in each block past 32 actually decreases performance – the exception seemingly being `com-Youtube`.

Table 1: The execution times (in msec) of the program for each of the matrices. The line labeled CPU is the execution time of the Serial Implementation and is given as a comparison.

Blocks	`auto`	`great..`	`delaun..`	`com-You..`
CPU	833 ms	835 ms	1799 ms	23781 ms
64	132 ms	908 ms	663 ms	5579 ms
128	78 ms	580 ms	429 ms	4686 ms
256	50 ms	405 ms	308 ms	3962 ms
512	34 ms	321 ms	242 ms	3467 ms
1024	27 ms	291 ms	223 ms	3301 ms
2048	29 ms	286 ms	213 ms	3257 ms
4096	27 ms	276 ms	202 ms	3177 ms

(a) Adjusting Num. Blocks while keeping the Threads/Block constant and equal to 32

Threads	`auto`	`great..`	`delaun..`	`com-You..`
CPU	828 ms	834 ms	1782 ms	23748 ms
32	47 ms	416 ms	298 ms	3966 ms
64	49 ms	435 ms	307 ms	2581 ms
128	51 ms	508 ms	338 ms	1926 ms
256	63 ms	813 ms	493 ms	1709 ms
512	118 ms	1416 ms	855 ms	1793 ms

(b) Adjusting the Threads/Block while keeping Num. Blocks constant and equal to 256

4 Feedback

In the last assignment, I said I regret leaving it to the last moment and rushing it, and that I would try to not do that for the next one (meaning this one). As it turns out, that did not work out as planned[1].

Nevertheless, here's some feedback on this assignment.

4.1 CUDA

First of all CUDA was not as difficult as I expected it to be, and getting it to work correctly and somewhat efficiently was quite straightforward. That being said, I do not believe my implementation is entirely optimized.

I was unable to do theoretical bandwidth/performance percentage analysis, mainly due to the difficulty in evaluating the number of read/write memory operations correctly, but by some estimates I don't believe it's very high.

4.2 Testing

One might notice the lack of graphics in this Report, and that is on purpose, because I do not believe that the results obrained would fit well in plotted format. For this reason I attached the results in tabular format.

I will say I had some difficulty choosing which parameters to even test in my testing. I chose the number of blocks and threads since that was somewhat obvious but I cannot help but feel that there was more I could have tested.

Additionally, the results obtained are somewhat inexplicable. For example `com-Youtube` needing more than 10 times more execution time than the other matrices, even though it is smaller both in rows and in non-zero elements.

4.3 Report

The report is shorter than the previous ones, even one page less than the limit[2]. This is partially explained by the fact that [1] was given as a resource which was pretty complete when it came to explaining the problem and methods, and partially due to the apparently low complexity of the implementation.

Regardless, I'd rather have more to say than less, and so I'm a little dissapointed. The issue here I think again is that I did not have enough time.

That being said, I'm hoping that the following and last assignment, having a few months to work on it [3], will turn out more well-rounded and complete.

References

[1] Dimitris Floros, Nikos Pitsianis, and Xiaobai Sun. "Fast Graphlet Transform of Sparse Graphs". In: *IEEE High Performance Extreme Computing Conference*. 2020.

[1] Being in the exam period I feel it is somewhat justified
[2] In the last report I went over the limit
[3] Fingers crossed