Aristotle's University of Thessaloniki
Electrical and Computer Engineering Department

Parallel and Distributed Systems

Third Project - Fast Graphlet Transform on GPU

Project of:

Ioannis Stefanidis
AEM: 9587

Panagiotis Syskakis
AEM: 10045

Isidoros Tsaousis-Seiras
AEM: 10042

Code Repository:
https://github.com/isidorostsa/cuda_fglt

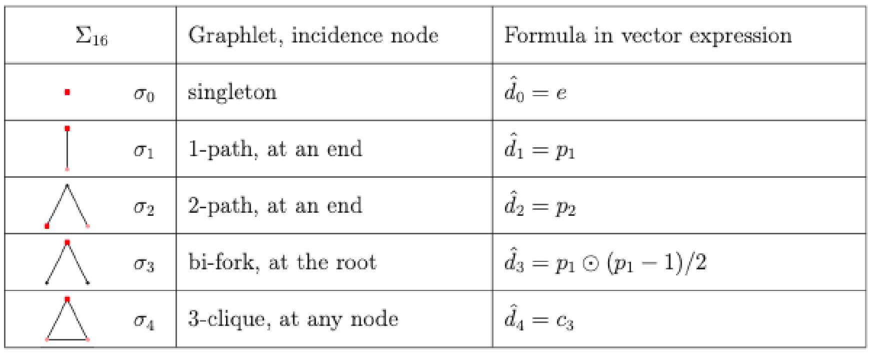
March 1, 2023

Introduction

In this assignment, we implement a subset of Fast Graphlet Transform (FGlT) calculations on the GPU.

FGlT ([1]) is a library for quantifying the prevalence of certain subraphs (graphlets) in large, sparse, undirected graphs. The frequency of these subgraphs encodes topological information about the graph, which may prove useful for further graph analysis and classification.

For the purposes of this assignment, calculations were restricted to graphlets σ_1, σ_2, σ_3 and σ_4, as illustrated in Fig. 1.



Σ_{16}	Graphlet, incidence node	Formula in vector expression
σ_0	singleton	$\hat{d}_0 = e$
σ_1	1-path, at an end	$\hat{d}_1 = p_1$
σ_2	2-path, at an end	$\hat{d}_2 = p_2$
σ_3	bi-fork, at the root	$\hat{d}_3 = p_1 \odot (p_1 - 1)/2$
σ_4	3-clique, at any node	$\hat{d}_4 = c_3$

Auxiliary Formulas
$p1 = Ae$
$p2 = Ap_1 - p_1$
$C_3 = A \odot A^2$
$c_3 = C_3 e/2$
$d_0 = \hat{d}_0$
$d_1 = \hat{d}_1$
$d_2 = \hat{d}_2 - 2\hat{d}_4$
$d_3 = \hat{d}_3 - \hat{d}_4$
$d_4 = \hat{d}_4$

Figure 1: Graphlets and auxiliary formulas

1 General Approach

1.1 Scope

The initial goal of this assignment was to explore, combine and compare several GPU libraries, such as CUDA, Thrust, cuSPARSE. We also experimented using Taskflow for asynchrous execution, and Julia to gauge the usability of it's CUDA.jl module. We also made parallel CPU implementations using OpenMP and OpenCilk for comparing CPU with GPU performance, benchmarked our implementations and compared them to the reference FGLT CPU implementation in [1].

1.2 Graph Representation

The need to process large graphs necessitates the use of a *compressed sparse row* (CSR) matrix representation. Additionally, even though calculations are presented in matrix form (Fig. 1), performance may be gained by leveraging problem-specific constraints instead, as will be explained in Section 3.

2 Implementation Specifics

2.1 Matrix Representation

A pair of structs was used to hold the data for the CSR matrices. Their rough structure was

```
struct h_csr{
    thrust::host_vector<int> offsets, positions, values;
    int rows, cols, nnz;
}
```

```
struct d_csr{
    thrust::device_vector<int> offsets, positions, values;
    int rows, cols, nnz;
}
```

Where `offsets` represents the compressed and `positions` the uncompressed row of the `CSR` Matrix. The difference between the two structs is that `d_csr` is stored in the device wheras `h_csr` is in the host.

2.2 The use of Thrust

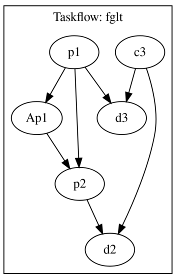
Thrust is an open source library developed by NVIDIA and contributors to add a level of abstraction on top of handling CUDA linear structures. It was used as the GPU equivalent of `std::vector`, holding all the arrays used to calculate the frequencies.

2.3 The use of cuSPARSE

cuSPARSE is a proprietary library developed by NVIDIA to execute sparse linear algebra algorithms in CUDA. We implemented a user-friendly wrapper for a CSR Matrix that can be manipulated by cuSPARSE. This was initially used for the calculation of A^2, however In the final implementation this calculation was not performed, so cuSPARSE was not used either.

2.4 The use of Taskflow

Taskflow is an open source library used to execute tasks while keeping track of execution dependencies. We used it to execute the calculations in a graph-like fashion. Although Taskflow is commonly used to parallelize tasks, the gpu was fully utilized throughout the duration of the algorithm, so no performance gains where observed. It would be interesting to see a hybrid task-based CPU/GPU implementation.



2.5 The use of Julia

Julia is an efficient high-level open source computational library. We implemented a Julia solution quite quickly, thanks to Julia's ease of use. The structs we used were already present in CUDA.jl, which is the Julia interface for CUDA.

3 Graphlet Algorithms

For any node *i*, FGlT algorithms essentially count how many subgraphs exist where *i* is the incidence node (Fig. 1).

In general, the calculations closely follow the notation in Fig. 1, taking into account that A is a CSR matrix. We implemented CPU, CUDA and Thrust versions of all calculations. A few selected examples are demonstrated below.

SPMV using CUDA kernel

```
__global__ void spmv_kernel(int *A_offsets, int *A_positions,
                            int *x, int *y, int n){
    for (int i = blockDim.x * blockIdx.x + threadIdx.x; i < n;
        i += blockDim.x * gridDim.x){
        int sum = 0;
        for (int j = A_offsets[i]; j < A_offsets[i + 1]; j++){
            sum += x[A_positions[j]];
        }
        y[i] = sum;
    }
}
```

σ_1 (1-path) using Thrust

```
thrust::transform(
    d_A.offsets.begin() + 1, d_A.offsets.end(),
    d_A.offsets.begin(), d_p1.begin(),
    thrust::minus<int>()
);
```

σ_4 (3-clique, at any node) on CPU

```
void c3(int n_vertices, int* A_offsets, int* A_positions, int *c3) {
// for every node i
  for(i = 0; i < A; i++) {
    int i_nb_start = A_offsets[i], i_nb_end = A_offsets[i + 1];

    for (int i_nb_idx = i_nb_start; i_nb_idx < i_nb_end; i_nb_idx++) {
      int j = A_positions[i_nb_idx];

      if(i<=j) break;
      // j are all neighbors of i, with i>j
      // Find common neighbors of i, j

      int j_nb_start = A_offsets[j], j_nb_end = A_offsets[j + 1];

      int _i_nb_idx = i_nb_start, _j_nb_idx = j_nb_start;

      while (_i_nb_idx < i_nb_end && _j_nb_idx < j_nb_end){
        // If a neighbor of i or j has larger node id, break
        if ((A_positions[_i_nb_idx] > i) ||
            (A_positions[_j_nb_idx] > j)){
            break;
        }
        // If common neighbor, count triangle
        else if (A_positions[_i_nb_idx] == A_positions[_j_nb_idx])
        {
          c3[j]++, c3[i]++, c3[A_positions[_i_nb_idx]]++;
          _i_nb_idx++, _j_nb_idx++;
        }
        // Else increment lowest index
        else if (A_positions[_i_nb_idx] < A_positions[_j_nb_idx]){
          _i_nb_idx++;
        }
        else _j_nb_idx++;
      }
    }
  }
}
```

The calculation of σ_4 (3-clique) proved of significant interest. In provided notation, intermediate matrix $C_3 = A \odot A^2$ is calculated, and $c_3 = C_3 e/2$ is calculated afterwards, which is in essence the row-wise summation of C_3.

In practice, it would be inefficient to actually perform these calculations and store the intermediate results. Due to the Hadamart product, we only need to calculate elements of A^2 that also exist in A. In addition, the row-wise summation may happen in-place, giving us a single unified algorithm for calculating c3.

Our algorithm can be conceptualized by viewing c3[i] as a count of the number of triangles (3-length loops) that are originating from some node i: Starting from node i, we iterate through neighbors j.

Having the neighbors of i and j as two sorted arrays (located at A->unc[i] and A->unc[j]), we find their common elements.

Finally, the calculations are further optimized by taking into account that the CSR representation of *A* contains all elements in a sorted fashion. Again, conceptualizing the problem as triangle-count, we choose to traverse the graph only for increasing node-id, thus counting each triangle only once.

4 GPU Parallelism

4.1 Approaches

A trivial GPU parallelization of the Graphlet for-loops is easy to implement in CUDA or Thrust. We also used the Grid-Stride Loops technique to split the workload into chunks. This method uses less threads than the totality of the elements that need to be processed, assigning multiple elements to each thread.

4.2 Limitations of GPU parallelism

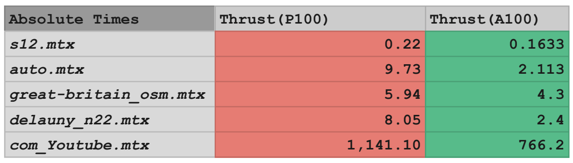
In the calculation of c3 (which accounts for the majority of computation time), the issue of efficient parallel sparse matrix multiplication [2] (equivalent to sparse graph traversal) becomes apparent as a challenging and performance-limiting factor. The two main issues are:

- **Inefficient memory access pattern** when using CSR representation.
- **Thread divergence**, meaning that threads in the same GPU block have an unequal amount of work, leading to the threads that finish early to stay idle until the last thread in that block finishes.

These limitations could be addressed by using more complex data structures, or by distributing the matrix rows such that rows with similar amount of expected work (ie elements) end up in the same block, which could be scheduled using CUDA Dynamic Parallelism. However, even without these optimizations, a GPU may still improve performance, as we see in the next section.

5 Results

The performance of our implementations was tested on the Aristotelis HPC Cluster. The CPU versions were tested on a 2-socket AMD EPYC 7302 node (32 CPU cores and 64 CPU threads in total). The GPU versions were tested on a NVIDIA Tesla P100 GPU, including memory allocations, host-to-device, kernel execution and device-to-host time in the measurement. The Thrust method was also tested on the NVIDIA Ampere A100 GPU.



Absolute Times	Thrust(P100)	Thrust(A100)
s12.mtx	0.22	0.1633
auto.mtx	9.73	2.113
great-britain_osm.mtx	5.94	4.3
delauny_n22.mtx	8.05	2.4
com_Youtube.mtx	1,141.10	766.2

Figure 2: Absolute execution times in milliseconds

Due to our optimizations, all implementations showed significant speedup compared to [1] for all sparse graphs tested, as seen in Fig. 3.

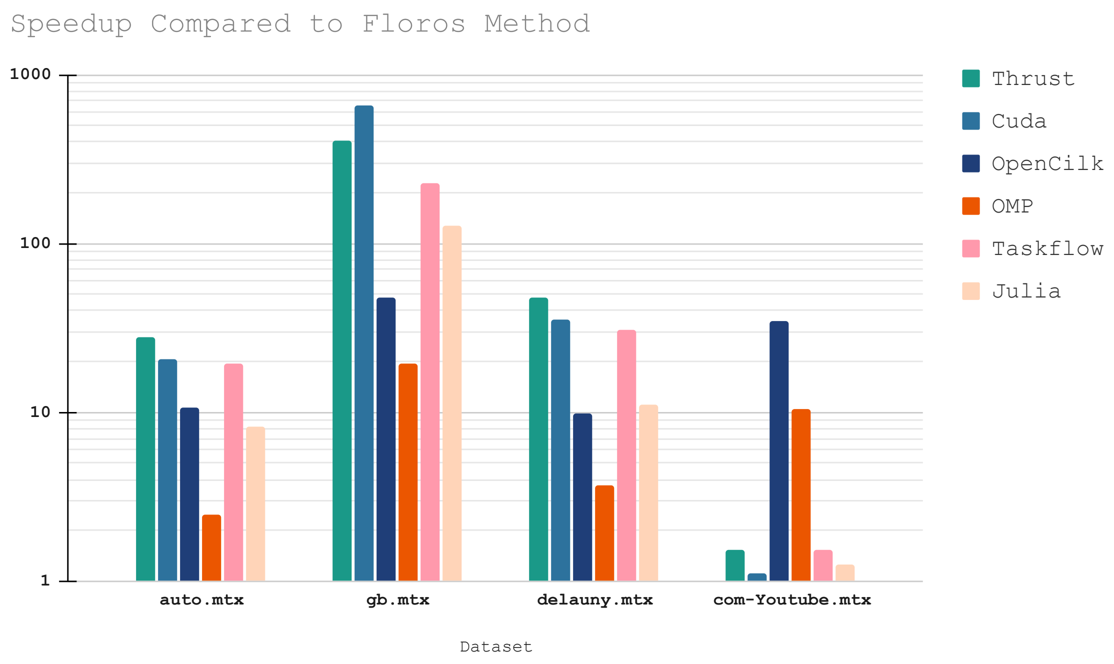


Figure 3: Speedup Compared to the reference method

5.1 GPU parallelism gone wrong

Looking at the results, we see that the GPU implementation is faster for all tested graphs, except for *com-Youtube*. Not only the GPU implementation was slower for *com-Youtube*, but it performed worse than *great-britain* and *delaunay*, despite those being much larger graphs. Our theory is that this is a result of how the GPU parallelization works, as discussed in Section 4.2. Examining the variance of the frequency σ_4 for the 3 graphs *com-Youtube*, *great-britain* and *delaunay* (77275, 0.004, 1.89 respectively), we found that in *com-Youtube* graph the amount of unequal work per block is enormous.

We also notice that the *OpenCilk* is much faster than other implementations in *com-Youtube*, which may be attributed to its Work-Stealing Scheduler[3].

References

[1] Dimitris Floros, Nikos Pitsianis, and Xiaobai Sun. *Fast Graphlet Transform of Sparse Graphs*.

[2] Nathan Bell and Michael Garland. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. https://www.nvidia.com/docs/io/66889/nvr-2008-004.pdf.

[3] *Cilk's Work-Stealing Scheduler*. https://www.opencilk.org/img/opencilk-pact-2021.pdf. 2021.