Parallel Programming Report of the 3rd Assignment

Φωτεινή Κοκκώνη

Instructions

Firstly we will provide the proper way of running everything

Step1

Make all the necessary imports for our CUDA and Kernels by running the coding block:
import Pkg
Pkg.add("KernelAbstractions")
Pkg.add("CUDA")

Step2

Import any .mtx graph and name it "graph.mtx", move it to the sample_data folder and run it. The first time it will take a LONG time. That is because it is making some other files

Step3

Run it again with the new files created and see it run like a lighting!

Approach Analysis

With the use of Julia, Cuda Kernels and different code structure I was able to achieve a new record time for the graph of Friendster, much faster than any other implementation. I still have the typical loading and cc connection functions but I managed to sky-rocket the loading time through a different approach which reads the dimensions of the graph and converts each string to Int32 (used to optimize our code).
Why use Int32? That is because it has a smaller memory footprint, allows the information to be faster on GPU and it matches perfectly with CUDA providing a friendly indexing. The graph is undirected therefore each edge(u.v) becomes (u->v),(v->u) which leads to the worst-case edges being 2* number of non zeros.
Next we store the indexes of the u and v edges, with the use of undef we achieve faster allocation cause the values will be overwritten anyway. We add an edge_count to count the edges added this far, the following loop reads line by line(no full file in memory), removes the whitespace, avoids blank lines and skips comment lines inside the file. Then we "extract" the vertex IDS, convert them from strings to integers while updating our edge_count. We have a code condition for reverse edges which adds reverse edges,makes sure that (u->v) should not be duplicated and ensures that our adj lists are symmetric.

CSR Build

After all edges are read, the arrays are resized to match the actual number of stored edges, reclaiming unused memory. Then we continue to the CSR building where we use a cache-friendly and GPU-friendly, compact format. This allows fast access to all the neighbors of a vertex, contiguous memory access patterns and efficient execution on parallel architectures(GPUS). Specifically rowptr[v] stores the starting index of vertex v's adjacency list in colind, rowptr[v+1] stores the ending index(exclusive) and the colind stores ALL adjacency lists concatenated into a single array!

CSR Storage

The CSR arrays are written directly to disk in binary format. This avoids expensive MTX parsing in future runs! Then the Binary files are read and reinterpreted as Int32 arrays, enabling fast, zero-copy loading

GPU CC Algorithm&Kernel Logic

Each GPU thread processes 1 vortex. The kernel performs label propagation, where each vertex adopts the minimum label among itself and its neighbors.

In the Kernel each vertex starts with its own label, the kernel scans all neighbors using CSR and if a smaller label is found the vertex updates its label. Meanwhile a global changed_flag variable is set to indicate convergence is NOT yet reached.

When it comes to the GPU Driver Function we firstly transfer CSR data to the GPU, then we initialize the vertex labels which will allow us to repeatedly launch the kernel until no labels change. In the end our algorithm copies the final labels back to the host and the loop terminates ONLY when the variable changed_flag remains zero after an iteration.

Main Program Flow

Firstly our int main checks for the binary CSR files, if they exist it loads them otherwise it goes ahead to Load .mtx, build the CSR and save CSR in binary form. Then when everything is loaded it runs the Connected Components on GPU while measuring and report execution times

Performance Characteristics

CSR Construction O(V+E)
Connected Components: O(E * iterations)
Memory Efficiency: Uses compact Int32 arrays
Scalability: Suitable for large sparse graphs
Number of Iterations:Fixed 10, it depends on the algorithm NOT the threads

Observations(V1)

Firstly we rename the com-Friendster.mtx to graph.mtx, then we run the import code block. The first run is VERY slow, that is because we read and build our binary files that were previously mentioned.

```
Binary CSR not found. Loading MTX and building CSR...
Loaded MTX in 10.611 s
Saving CSR binary...
Running Connected Components on GPU...
Iterations: 7
CC time: 21.393 s
Total time: 37.729 s
Connected components: 64059426
First 10 labels: Int32[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

After the second run we get:

```
Loading CSR from binary files...
Loaded CSR in 0.184 s
Running Connected Components on GPU...
Iterations: 7
CC time: 1.291 s
Total time: 1.476 s
Connected components: 64059425
First 10 labels: Int32[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Which is AS LOT faster than any other iterations of ours, specifically this run time is ~56.9% faster than our best run time achieved on Aristotelis from assignment2 and has a 2.31 speedup! Compared to our best run from the first assignment we got a 73% time improvement and a 3.8 speedup. Phenomenal work

We delete the binary files and proceed to do the same for other graphs.

Com-Orkut

Our first run:

```
Binary CSR not found. Loading MTX and building CSR...
Loaded MTX in 43.279 s
Saving CSR binary...
Running Connected Components on GPU...
Iterations: 10
CC time: 0.96 s
Total time: 45.203 s
Connected components: 13
First 10 labels: Int32[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

our second run thou:

```
Loading CSR from binary files...
Loaded CSR in 0.15 s
Running Connected Components on GPU...
Iterations: 10
CC time: 0.685 s
Total time: 0.836 s
Connected components: 13
First 10 labels: Int32[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Amazing results in comparison with our first assignment, we got the graph being processed 85.6% faster which also means we achieve a ~7 speedup! Now we are talking about real speed!

Remember to delete the binary files!

Com-LiveJournal

First run:

```
Binary CSR not found. Loading MTX and building CSR...
Loaded MTX in 53.439 s
Saving CSR binary...
Running Connected Components on GPU...
Iterations: 10
CC time: 0.554 s
Total time: 55.116 s
Connected components: 14860
First 10 labels: Int32[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Our second run thou:

```
Loading CSR from binary files...
Loaded CSR in 0.241 s
Running Connected Components on GPU...
Iterations: 10
CC time: 0.839 s
Total time: 1.083 s
Connected components: 14860
First 10 labels: Int32[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Great results for one more time, according to the statistics of the first assignment we have a code that is 87.4% faster than our fastest run and has a 7.95 speedup!

Number of Threads Observation!

On our original code we add the benchmark_thread_configurations function(+stride) and make the appropriate changes to allow it to run so that we can see the difference on Iterations and CC time when we do not use our max parallelism(1 thread per vertex)

Friedster Analysis

You can see the statistics on the console as well
For each thread having 1 vertice per thread = 11seconds

Some Statistics from the console:

Threads →	1k	2k	4k	8k	16k	32k	64k	128k	256k
Total Time	11.3s	10.95s	11s	10.284s	10.744s	10.8s	10.25s	10.7s	10.6s

Observations

From the console's final report we notice that we achieve the best mean time with 518k threads 10.25s,we can obviously see that the increase of threads does NOT mean decrease of the Total Time of our implementation.
Why is that?
That is because our algorithm is memory bound and NOT not compute-bound , our GPU e.g has a maximum memory bandwidth of 400GB/s, our algorithm needs 10.7GB of data.
Minimum possible Time= Data/Bandwidth
That is why when we have f.e 1M threads we DO NOT get a better Total time that when we got 8k, our "operations' WAIT for the available memory.