Contents

1 Problem description **2**

2 Solution **3**

2.1 MPI 3

2.2 CUDA 3

2.3 MPI & CUDA 4

3 Experimental setup **5**

3.1 Hardware 5

3.2 Memory Device 6

3.3 Software 7

4 Performance and SpeedUp **8**

4.1 Performance 8

4.2 SpeedUp 11

5 Conclusions **13**

Problem description

The requirement was to provide a parallel version of the Hopcroft-Karp algorithm for finding a maximum-cardinality matching in a Bipartite Graph. This implementation is a hybrid of message passing and shared memory paradigms, leveraging MPI and CUDA technologies. There are no constraints on the graph structure or memory allocation type, allowing for complete flexibility in data management.

Solution

In the proposed solution, the bipartite graph is generated randomly. Specifically, it is generated based on the number of nodes in each of the two parts. As a matter of simplicity and to better handle testing, given a pair of m (number of nodes in set X) and n (number of nodes in set Y), the generated graph is always the same.

2.1 MPI

The bipartite graph is generated. Each process is responsible for computing a local matching within the graph. This parallel execution enables the algorithm to leverage the computational resources of multiple processors simultaneously, thereby potentially speeding up the overall computation.

After computing the local matchings, the results need to be aggregated to determine the global maximum matching. MPI facilitates this aggregation through collective communication operations. Specifically, the local matchings from all processes are gathered to a designated root process (often process with rank 0). This root process then analyzes the collected data to identify the global maximum matching.

2.2 CUDA

The core of the Hopcroft-Karp algorithm is the breadth-first search (BFS) traversal, which is crucial for finding augmenting paths in the bipartite graph.

In CUDA, the BFS traversal is parallelized by assigning each vertex in set X of the bipartite graph to a separate CUDA thread.

This parallelization enables multiple BFS traversals to occur concurrently, with each CUDA thread exploring a different part of the graph simultaneously.

As a result, the computational workload is distributed across thousands of CUDA cores available on the GPU, leading to significant speedup compared to sequential execution on the CPU.

2.3 MPI & CUDA

By combining the distributed memory parallelism of MPI with the GPU acceleration of CUDA, the hybrid approach achieves a synergistic effect, enabling efficient parallelization of the Hopcroft-Karp algorithm on large-scale bipartite graphs. This approach leverages the scalability of MPI and the massive parallelism of GPUs, resulting in significant performance gains and scalability across heterogeneous computing platforms.

Experimental setup

3.1 Hardware

```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         43 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  4
  On-line CPU(s) list:   0-3
Vendor ID:               AuthenticAMD
  Model name:            AMD Ryzen 3 3250U with Radeon
     Graphics
    CPU family:          23
    Model:               24
    Thread(s) per core:  2
    Core(s) per socket:  2
    Socket(s):           1
    Stepping:            1
    Frequency boost:     enabled
    CPU(s) scaling MHz:  60%
    CPU max MHz:         2600,0000
    CPU min MHz:         1400,0000
    BogoMIPS:            5190,28
    Flags:               fpu vme de pse tsc msr pae mce cx8
       apic sep mtrr pge mca cmov pat pse36 clflush
                         mmx fxsr sse sse2 ht syscall nx
                            mmxext fxsr_opt pdpe1gb rdtscp lm
                             constant_tsc rep_good
                         nopl nonstop_tsc cpuid extd_apicid
                            aperfmperf rapl pni pclmulqdq
```

```
                                monitor ssse3 fma cx16
                            sse4_1 sse4_2 movbe popcnt aes xsave
                                avx f16c rdrand lahf_lm
                                cmp_legacy svm extapic cr8_legacy
                            abm sse4a misalignsse 3dnowprefetch
                                osvw skinit wdt tce topoext
                                perfctr_core perfctr_nb bpext
                            perfctr_llc mwaitx cpb hw_pstate
                                ssbd ibpb vmmcall fsgsbase bmi1
                                avx2 smep bmi2 rdseed adx
                            smap clflushopt sha_ni xsaveopt
                                xsavec xgetbv1 clzero irperf
                                xsaveerptr arat npt lbrv svm_lock
                            nrip_save tsc_scale vmcb_clean
                                flushbyasid decodeassists
                                pausefilter pfthreshold avic
                                v_vmsave_vmload
                            vgif overflow_recov succor smca sev
                                sev_es
```

3.2 Memory Device

```
Array Handle: 0x0013
Error Information Handle: 0x001C
Total Width: 64 bits
Data Width: 64 bits
Size: 4 GB
Form Factor: SODIMM
Set: None
Locator: Bottom - Slot 2 (right)
Bank Locator: P0 CHANNEL B
Type: DDR4
Type Detail: Synchronous Unbuffered (Unregistered)
```

Speed: 2667 MT/s

Manufacturer: Kingston

Serial Number: E6629A6F

Asset Tag: Not Specified

Part Number: HP26D4S9S1ME-4

Rank: 1

Configured Memory Speed: 2400 MT/s

Minimum Voltage: 1.2 V

Maximum Voltage: 1.2 V

Configured Voltage: 1.2 V

3.3 Software

1

OS: Linux Ubuntu 6.5.0-21-generic.x86_64

GCC: gcc (Ubuntu 13.2.0-4ubuntu3) 13.2.0

Swap: 4GiB

Performance and SpeedUp

4.1 Performance

Regarding performance, testing was carried out for three dimensions of the graph: 10000 nodes, 20000 nodes, 30000 nodes. N.B. These numbers refer to the number of nodes in each of the two partitions. It was chosen to have the same number of nodes in each partition as a matter of simplicity. To ensure a fair veracity of the results, each test was run 10 times. Before showing the test results, it is important to specify that regarding MPI, the -O3 optimization of gcc was used and 4 MPI processes were used.

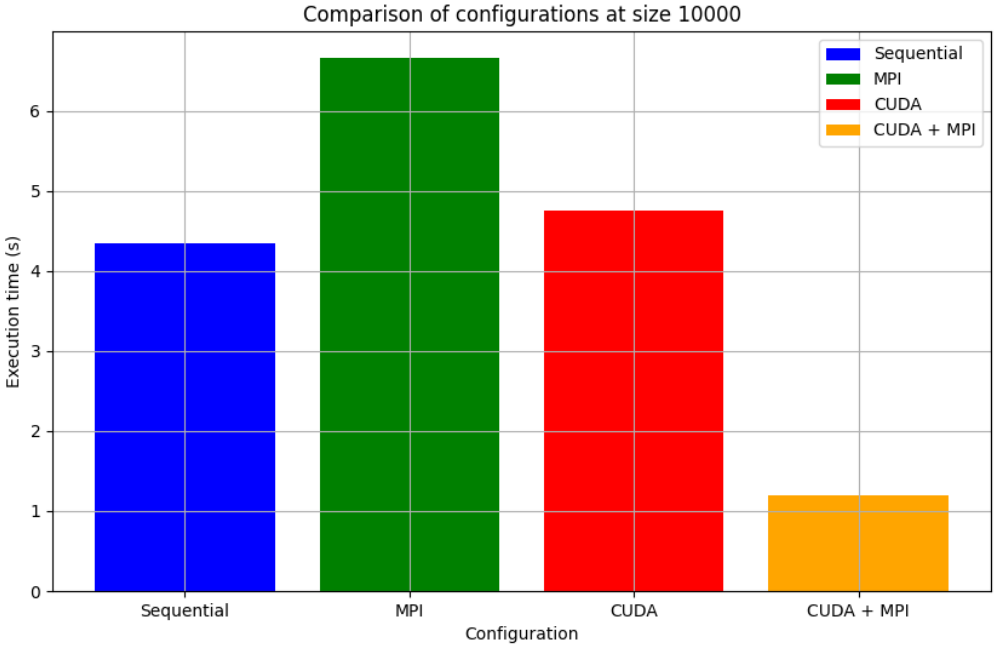

Figure 4.1: Comparison of configurations at size 10000

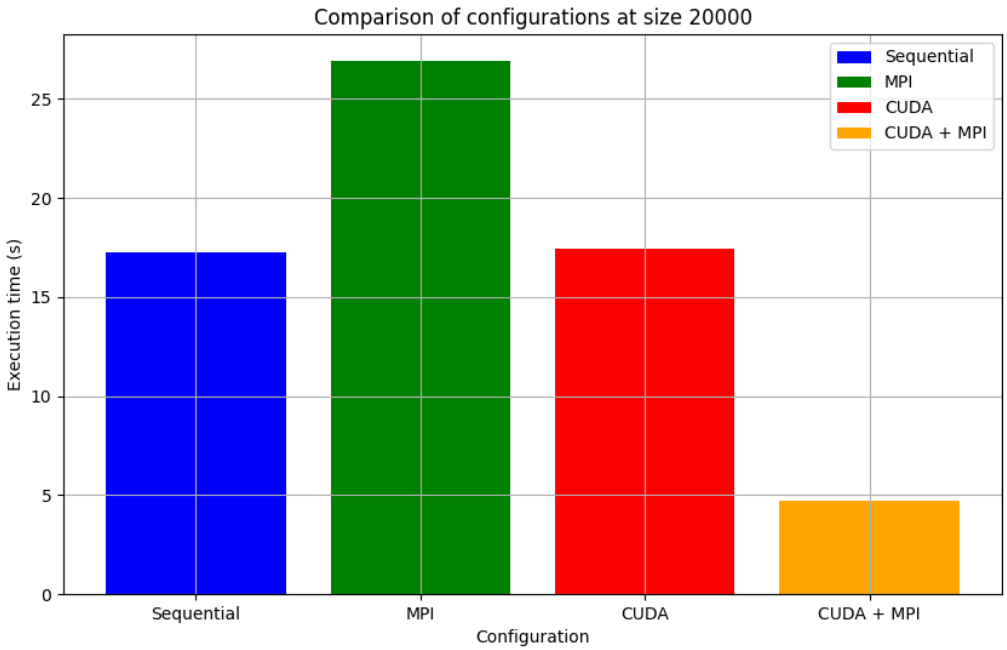

Figure 4.2: Comparison of configurations at size 20000

Regarding the 10000 and 20000 sizes, we can see that the performance of CUDA is similar to the serial version. The same is true for MPI, which is even worse than the serial version, this may be due to the overhead caused by synchronization and communication between processes.

However, we can see that the hybrid CUDA+MPI version increases performance dramatically and thus is the optimal choice.

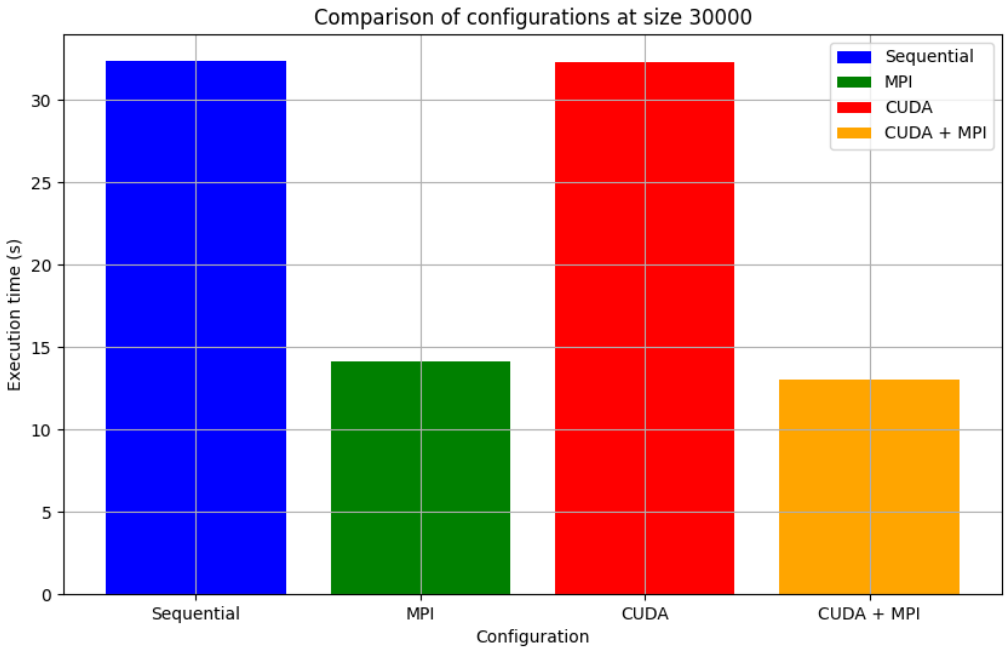

Figure 4.3: Comparison of configurations at size 30000

Finally, we can see that as the size increases, the version with MPI greatly increases performance. However, this argument does not apply to the version with CUDA. However, the hybrid version still turns out to be the best.

4.2 SpeedUp

MPI:

Size	Speedup
10000	0.6615
20000	0.6863
30000	4.1096

CUDA:

Size	Speedup
10000	0.9778
20000	0.9864
30000	0.9925

CUDA+MPI:

Size	Speedup
10000	3.905
20000	4.7514
30000	3.8701

Looking at the speedup results for MPI, CUDA, and CUDA+MPI, we can observe some interesting findings:

MPI: The speedup obtained with MPI seems to be below 1 for all graph sizes, suggesting that parallel execution with MPI is actually slower than serial execution. This could be due to significant overhead caused by communication and synchronization between processes.

CUDA: Similarly, the speedup obtained with CUDA is below 1 for all graph sizes, although it is slightly higher than that of MPI. This might indicate that the parallel implementation with CUDA fails to fully exploit the capabilities of GPUs for relatively small data sizes.

CUDA+MPI: On the other hand, the combined use of CUDA and MPI appears to provide higher speedup compared to individual implementations. For all graph sizes, the speedup

is above 1, indicating an improvement in performance over serial execution. However, the speedup seems to vary significantly across graph sizes, with better performance for larger sizes.

In general, these results suggest that the combined implementation of CUDA and MPI might be the best strategy for achieving optimal performance across a wide range of graph sizes, although scalability and optimization are still important for achieving better results.

Conclusions

Both MPI and CUDA implementations individually failed to achieve significant speedup over serial execution for the given graph sizes. This suggests that neither MPI nor CUDA alone could efficiently harness the parallel processing power for relatively small datasets. The combined use of CUDA and MPI demonstrated improved performance compared to individual implementations. CUDA leverages the parallel computing capabilities of GPUs, while MPI facilitates communication between processes. This combination allows for better utilization of resources and improved performance, especially for larger datasets where CUDA's GPU parallelism can be effectively exploited.

While CUDA+MPI showed promising results, the scalability of this approach should be further investigated. It's essential to determine whether the performance gains observed for larger datasets hold true as the dataset size continues to increase. Scalability is crucial for ensuring that the solution remains effective as the problem scales.

Optimization plays a crucial role in maximizing performance. This includes optimizing algorithms for parallel execution, minimizing communication overhead, and leveraging hardware-specific features for improved efficiency. Further optimization efforts could potentially enhance the performance of both individual and combined implementations. In conclusion, while individual implementations may struggle to achieve significant speedup for smaller datasets, a combined approach leveraging CUDA and MPI shows promise for improving performance, especially for larger datasets. However, optimization and scalability remain critical considerations for achieving optimal performance across different hardware architectures and dataset sizes.

Note: A version of MPI was also added in which each process works on a subgraph of the original graph, but performance is found to be worse.