Lab 3: Breadth-First Search

ECSE 420 Parallel Computing Fall 2023

5 November 2023

Group 14: Samantha Handal 260983914 & Rebecca Mizrahi 260975001

Architecture

The experiments were run on a Google Colab T4 GPU, which is based on the Turing architecture. This GPU provides 320 Turing Tensor Cores and 2,560 CUDA cores, with a total of 16 GB GDDR6 memory, making it well-suited for parallel computations like BFS.

Sequential

The sequential implementation of BFS is important in setting a baseline to compare the parallelized schemes. The execution of one iteration took 2.250000 ms.

Parallel Implementation 1: Global Queue

Scheme and Implementation

The code for BFS operation with a parallel global queuing algorithm using CUDA has the following parallel scheme. The code explicitly allocates memory on the GPU for various arrays and variables using `cudaMalloc` and copies data from the host to the device using `cudaMemcpy`. The core computation is done in the `globalQueuingKernel` kernel function, which performs the BFS operation on the graph. This kernel processes nodes in parallel, handling a portion of work each, with the same operations as in the sequential code. The main difference is that the gate solver is defined on the device, as is the variable holding the number of next-level nodes, which is updated atomically by the kernel. The portion handled per thread is determined with the given parameters for block number and block size (thread number), and the elements per thread is calculated accordingly to evenly distribute work among threads. After each kernel execution, the code copies back the results from the device to the host.

Experimental Results

The execution time for the kernel to execute varied for the given block and thread configurations. It demonstrates how different combinations of thread and block configurations impact the overall performance of the BFS algorithm. Each combination resulted in a significant decrease in execution time when compared to the sequential algorithm, as they all had some degree of parallelization. From the table below, it is evident that as the load is more spread out among threads and blocks, the execution time decreases. The minimum time achieved is with 35 blocks and 128 threads, where each thread has a smaller portion of nodes to handle. On the other hand, the maximum time achieved is with 32 threads and 10 blocks. This combination achieves a much lower execution time than sequential, but it is not the most efficient way to balance the load, as each thread handles a large portion of nodes.

Table 1: Kernel execution time for different thread, block combinations

	32 threads	**64 threads**	**128 threads**
10 blocks	0.952608 ms	0.08192 ms	0.050944 ms
25 blocks	0.07552 ms	0.045024 ms	0.02896 ms

35 blocks	0.055296 ms	0.033248 ms	0.0248 ms

Parallel Implementation 2: Blocking Queue

Scheme and Implementation

The parallelization scheme of the Block Queue for the Breadth-First Search was designed to leverage the shared memory capabilities of CUDA-enabled GPUs. The key idea behind this approach is to create a local queue for each block, which allows threads within the same block to quickly share and access data without resorting to global memory access, which is slower.

In the blockQueuingKernel, each CUDA thread processes a subset of the nodes at the current level of the BFS. When a thread identifies a node that needs to be visited at the next level, it attempts to add this node to the block's local queue in shared memory. If the local queue is full, the thread adds the node directly to the global queue, ensuring no nodes are lost. The use of atomic operations prevents race conditions, ensuring that two threads do not write to the same location in memory simultaneously.

This design was aimed to minimize global memory transactions, which are a common bottleneck in GPU computing due to the higher latency compared to shared memory operations. By storing intermediate nodes in the local queues, we reduce the frequency and volume of atomic operations on global memory, which can significantly improve performance.

Experimental Results

The runtime results have been recorded in Table 2 and suggest that increasing the number of blocks (numBlock) and the size of each block (blockSize), while keeping the blockQueueCapacity constant, generally leads to better performance. The results also show that increasing the blockSize from 32 to 64 will improve performance.

These results show the effectiveness of the block queuing scheme when leveraging larger block sizes. It indicates that a higher degree of parallelism (with more threads working concurrently) can alleviate the workload per thread, leading to faster execution times.

Table 2: Kernel execution time for different thread, block, and block queue capacity combinations

		blockSize	
numBlock	**blockQueueCapacity**	**32**	**64**
25	**32**	0.27024 ms	0.167616 ms
	64	0.263392 ms	0.15584 ms
35	**32**	0.204032 ms	0.118112 ms

	64	0.195296 ms	0.116768 ms

Discussion

Upon analyzing the experimental results of both the Global Queue and Block Queue implementations, it is evident that the Global Queue approach outperforms the Block Queue strategy in the context of the experiments conducted on a Google Colab T4 GPU.

The Global Queue, with its simplicity and direct approach, benefits from the high-throughput capabilities of the T4 GPU's CUDA cores, with the fastest execution time observed at 0.0248 ms with 35 blocks and 128 threads. This configuration allows for a high degree of parallelism while maintaining a manageable workload for each thread, leading to efficient global memory usage and minimized overhead.

In contrast, the Block Queue method, which aimed to leverage shared memory for reducing global memory transactions, did not achieve the anticipated performance gains. The execution times ranged from 0.116768 ms to 0.27024 ms, which are consistently higher than those of the Global Queue. This could be attributed to several factors, such as the overhead of managing the block-local queues, the synchronization requirements between threads within a block, and the potential underutilization of shared memory when the block queue capacity is not fully utilized.

Furthermore, the choice of blockSize and blockQueueCapacity did not show a straightforward correlation with performance improvement in the Block Queue approach, indicating that the balance between these parameters is complex and possibly constrained by the specific characteristics of the BFS graph being processed.

These observations suggest that while the Block Queue approach has theoretical benefits, especially for scenarios where global memory access is a bottleneck, its practical application may be limited by the overheads introduced by shared memory management and the specific access patterns of the BFS algorithm.

In conclusion, the Global Queue method, with its lower execution times, is more efficient for parallelizing BFS on the T4 GPU architecture in this set of experiments. It highlights the importance of aligning parallelization strategies with both the algorithm characteristics and the architectural features of the execution platform.