Lab 1 Report – Logic Gates Simulation

ECSE 420 – Parallel Computing – Fall 2023

October 18, 2023

Group 14

Samantha Handal - 260983914

Rebecca Mizrahi - 260975001

Architecture

Google Colab offers a free T4 GPU for computational tasks, which is part of NVIDIA's Turing family. T4 has 16 Streaming Multiprocessors (SMs) and a total of 2560 CUDA cores. This means it's well-suited to handle thousands of threads in parallel, displaying benefits from increasing the number of threads in grid configuration.

With a memory bandwidth of up to 320 GB/s, data transfer between the global memory and the CUDA cores is rapid, minimizing potential bottlenecks that might arise when fetching pixel data.

Google Colab uses a unified memory model for GPU memory allocation; memory space is shared between the CPU (host) and GPU (device). This simplifies allocation and migration, as data migration between the CPU and GPU is managed automatically. In code that uses explicit memory allocation, for when one wants a more fine-grained control over memory usage, the code bypasses the automatic memory migration provided by the unified memory model. Colab allows you to explicitly allocate memory on the GPU if you choose to do so.

Sequential Logic Gates

The code for sequential logic gates does not leverage CUDA for parallel execution, and its main purpose is for sequential logic gate simulation to serve as a baseline for the parallelized code.

Parallelization with explicit memory allocation

In the explicit version, separate memory is allocated for the host/CPU and device/GPU, and data transfers between the two are explicitly managed, using cudaMemcpy. The data migration time is the time it takes to copy data from the CPU's memory to the GPU's memory. This operation can be relatively slow due to the need to copy data between separate memory spaces. There is also additional overhead involved, as you need to manually allocate and copy data between host and device memory. The findings of timing the data migration time reflect this information. For 10,000 inputs, the time taken is over 3 times higher than the time for the unified memory allocation version. As the inputs increase, more memory must be allocated and more transfers occur for these inputs. The time increases somewhat proportionally, reflecting the nature of the data transfers with this method.

After the data migration, the kernel execution time measures how long it takes to perform the actual logic gate computations on the GPU. This time mainly depends on the GPU's processing capabilities and the complexity of the operations. It is substantially lower than unified memory, not because of overhead associated with explicit memory allocation, but with latency with unified memory, as is discussed later. Logically, the time for kernel execution increases as more inputs are added, as there are more computations to complete. In our

code's case, this means more blocks of 256 threads are created and more GPU power is required. Thus, more time elapses before synchronization.

Parallelization with unified memory allocation

Unified memory simplifies data migration by allowing a single memory space accessible by both the CPU and GPU. It is designed to automatically manage data migration between the CPU and GPU. Only the data that is needed by the program is migrated, so unnecessary transfers are avoided. The actual data migration from host to device (or vice versa) can be deferred until the moment the data is actually accessed by the GPU or CPU. This means that when your program runs, data remains in its current location (either in CPU or GPU memory) until it is required for computation. Because of this, measuring data transfer time for unified memory is not straightforward. In this lab, we attempted a 'hacky' way of measuring data transfer time, by attempting to force a memory migration with a trivial kernel (parallel_logic_gate<<<1, 1>>>(A, B, gate, output, 0)), accessing the data on the device after setting it on the host, and then synchronizing. This resulted in the times: 0.026240 ms, 0.023872 ms, and 0.031200 ms. This approximation does indeed show the speedup of data transfer for unified memory over explicitly allocated memory. However, it is the only section that does not experience proportional increase in time with higher input levels; time remains relatively stable and low regardless of number of inputs. The deferred migration of memory approach minimizes unnecessary data transfers, as data that is not used by the program at a given moment remains in its current location (CPU or GPU). As such, the time that would be associated with data transfers with explicit memory allocation is generally postponed to the kernel execution, when data will be retrieved from the GPU, a miss will occur, and it will be automatically fetched from the CPU. This is why our approximated time for data transfer is not really relevant- no transfers should occur until the kernel is called. This is why the time is so low. Though, in general, the approach of moving data only when necessary may reduce the time spent on data migration and improve the overall performance of GPU-accelerated applications.

However, largely because of the migrations only occurring when needed by the program, unified memory may introduce some overhead due to its automatic migration. This overhead can impact the time taken for kernel execution, especially if the data being used by the kernel hasn't been migrated to the GPU yet. Due to its dependency on data locality and access patterns, unified memory can be less efficient, as shown in this case in which the time taken for the kernel to execute is substantially higher than explicit memory allocation. The code complexity and data transfer time are reduced but data accesses during kernel execution with unified memory result in more misses and consume more time. As seen with the other methods, higher inputs cause a substantial and somewhat proportional increase in time. This is because more data must be accessed per kernel execution.

Conclusion

In summary, unified memory and explicit memory allocation both have factors affecting latency positively and negatively, and each method can be beneficial depending on the purpose of a program. Working with unified memory simplifies memory management but introduces some overhead during kernel execution due to automatic data migration and transfers occurring at the last-minute among other factors. Working with explicit memory allocation introduces overhead during the manual data transfer. This lab highlights the importance of considering data access patterns, resource sharing, and the specific requirements of an application when choosing between unified and explicit memory allocation. Each approach has its advantages and trade-offs, and the choice should be based on the needs of the project and the efficiency of data management in the given context.

Appendix

Figure 1: Table of time taken for kernel execution and data migration with different memory approaches and at different input levels

Inputs	Data migration (host to device) time		Kernel execution time	
	Explicit	Unified	Explicit	Unified
10,000	0.090112 ms	/	0.023968 ms	0.532480 ms
100,000	0.380224 ms	/	0.041984 ms	1.024480 ms
1,000,000	2.878688 ms	/	0.173856 ms	4.833280 ms