Assignment 2 Report

Julius Putra Tanu Setiaji (A0149787E), Chen Shaowei (A0110560Y)

19 November 2018

1 Program Design

The program is implemented in CUDA. The goal is to achieve the best possible speedup. Primary design considerations are:

- Ensure maximum utilisation of computation resources.
- Minimise overhead as much as possible (especially by reducing the number of branching to prevent idle threads in an SIMT execution model).
- Ensure correctness of output by ensuring that only one process writes to result buffer.

2 Program Overview

1. *In host:* Read input.
2. *In host:* Setup:
    - Copy timestamp used to compute hash from host to device.
    - Copy target value from host to device.
    - Copy previous hash from host to device.
3. *In host:* Execute the kernel with the specified blocks-per-grid and threads-per-block.
4. *In kernel:* Compute initial nonce by calculating $blockId \times threadsPerBlock + threadId$.
5. *In kernel:* Construct SHA256 input and compute hash.
6. *In kernel:* Compare first 64 bits of digest with target value.
    - If the computed hash is satisfactory, execute an `atomicExch((int*)&is_found, 1)` to determine if current thread should write to output buffer.
    - Otherwise, increment the nonce by the value of $blocksPerGrid \times threadsPerBlock$.
7. *In kernel:* Check if `is_found` is set.
    - If set, return.
    - If unset, go to 5.
8. *In host:* Wait for kernel to complete by calling `cudaDeviceSynchronize()`.
9. *In host:* Print output.

3 Points to Note / Implementation Details

- Memory transfer between host and device memory is expensive and should be minimised. In our program, the only memory transfer between host and device are:
  - Copying of input and initial parameters from host to device in the setup stage.
  - Retrieve output value from device to host.
    - **Note:** This is an implicit transfer. The buffer for storing output values is allocated as Unified Memory.
- Input values (timestamp, target, previous hash) are stored in constant memory since they are only read, not written to.
- A test and set operation `atomicExch` is used when a valid nonce is found to ensure that only one thread writes to output buffer.
- The completion flag must be marked as `volatile` to ensure that Unified Memory optimisations do not cause the variable to be migrated into register/shared/local memory and its value is read from the global memory each time.
- Due to the avalanche effect in SHA256, the program searches for valid nonces incrementally starting from 0 for simplicity.

4 Performance Analysis and Discussion

4.1 Platform Overview

Various tests were run on the SoC compute cluster(XGPC) and the Jetson TX2 module in the lab. Given below is an overview of the two platforms:

- **XGPC**
  - CPU: 2× Intel Xeon 4108 (Dual socket)
    - 8 Cores
    - 16 Threads
  - GPU: NVIDIA Tesla V100-PCIE
    - 80 Volta SM
    - 64 CUDA cores per SM
- **Jetson TX2**
  - CPU: NVIDIA Denver2 + ARM Cortex A-57
    - 2 + 4 Cores (Heterogeneous)
  - GPU: NVIDIA
    - 2 Pascal SM
    - 128 CUDA cores per SM
  - *Note:* System memory is shared between CPU and GPU

5 Kernel Performance

In this section, we shall discuss the performance of the CUDA kernel. All values reported are 10-run averages. Target exponent of 2^{48} is used so as to not overflow the hardware counters when running `nvprof`.

Note: The following analysis does not consider the fixed overhead of copying input parameters from host to device and subsequent retrieval of result value. This is discussed in the next section.

5.1 Kernel Performance on Jetson TX2

The program was run on the Jetson TX2 with blocks-per-grid values between 1 and 64 and threads-per-block values between 1 and 256.

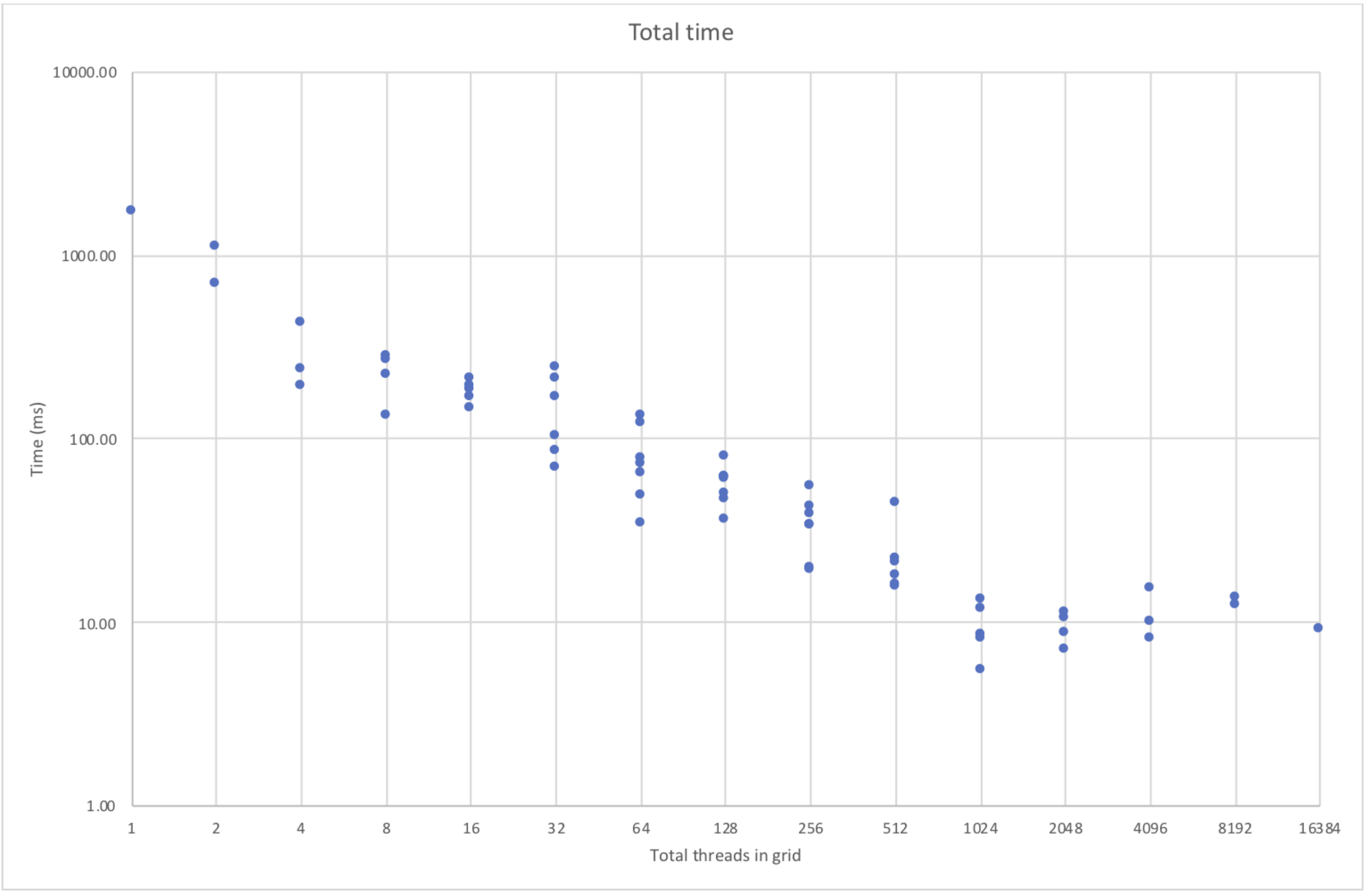

Figure 1: Scatter graph of time taken on Jetson against total threads in grid

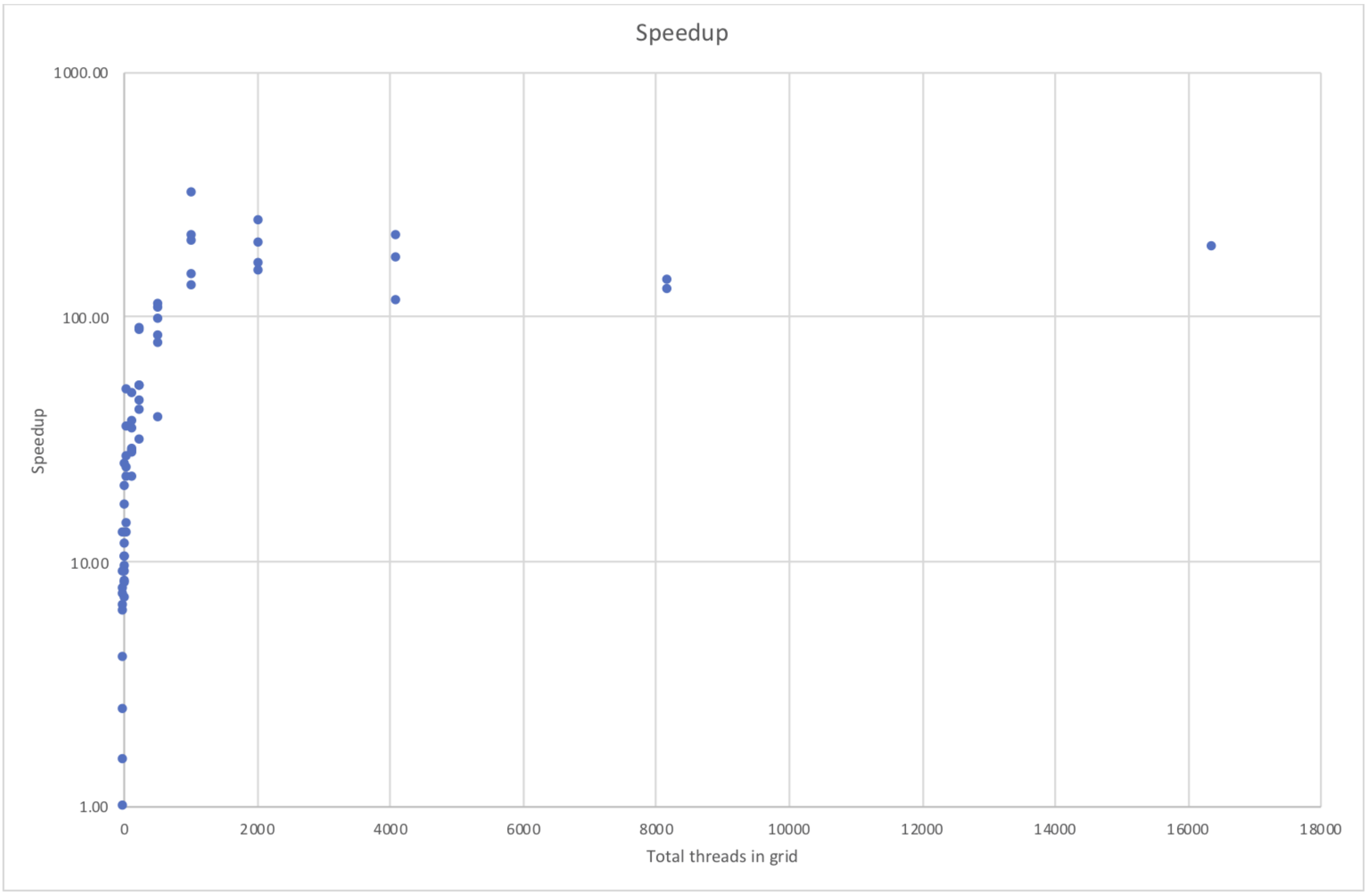

Figure 2: Scatter graph of speedup achieved on Jetson as number of threads increases

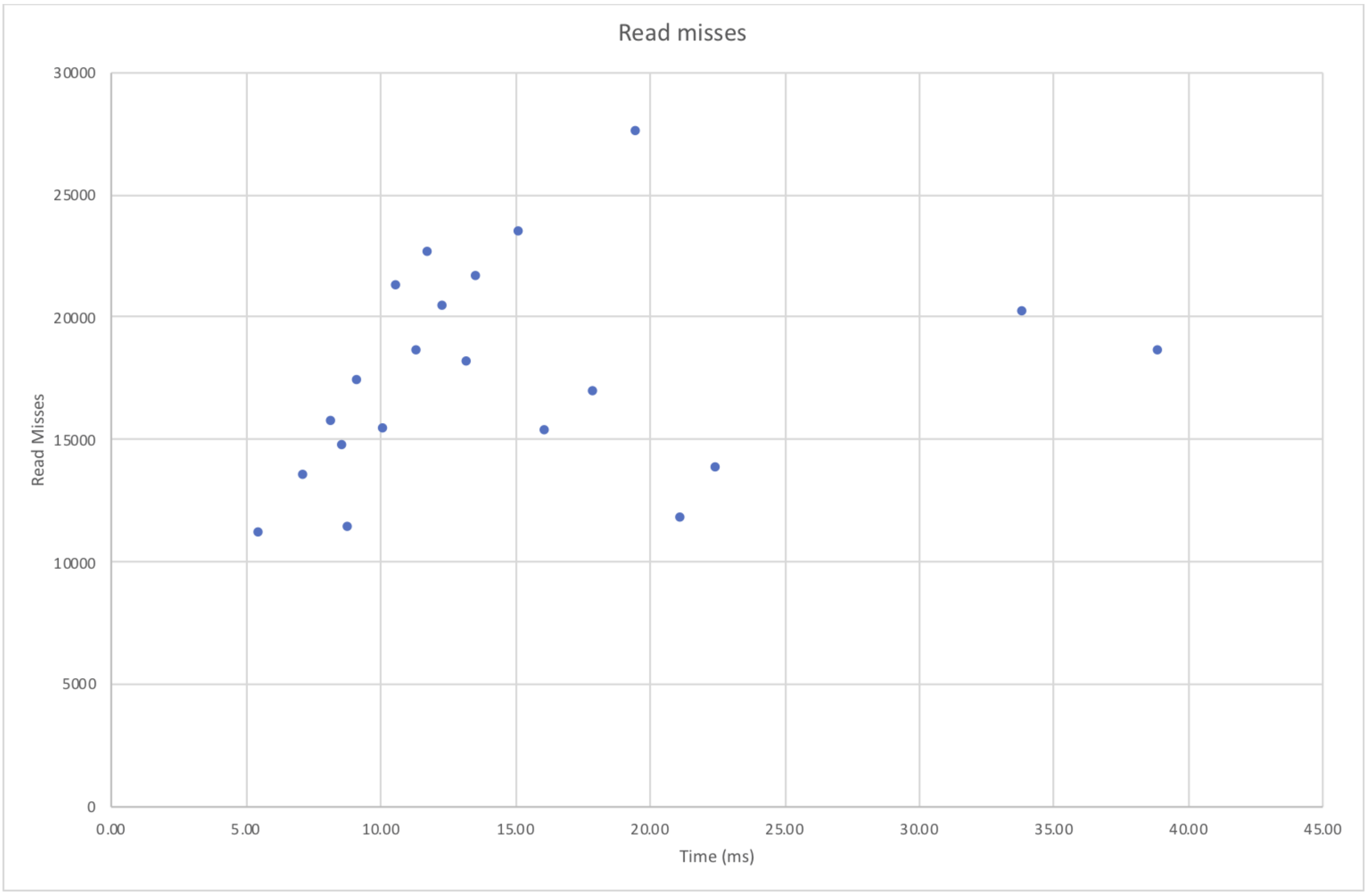

Figure 3: Scatter graph of number of read misses against time taken

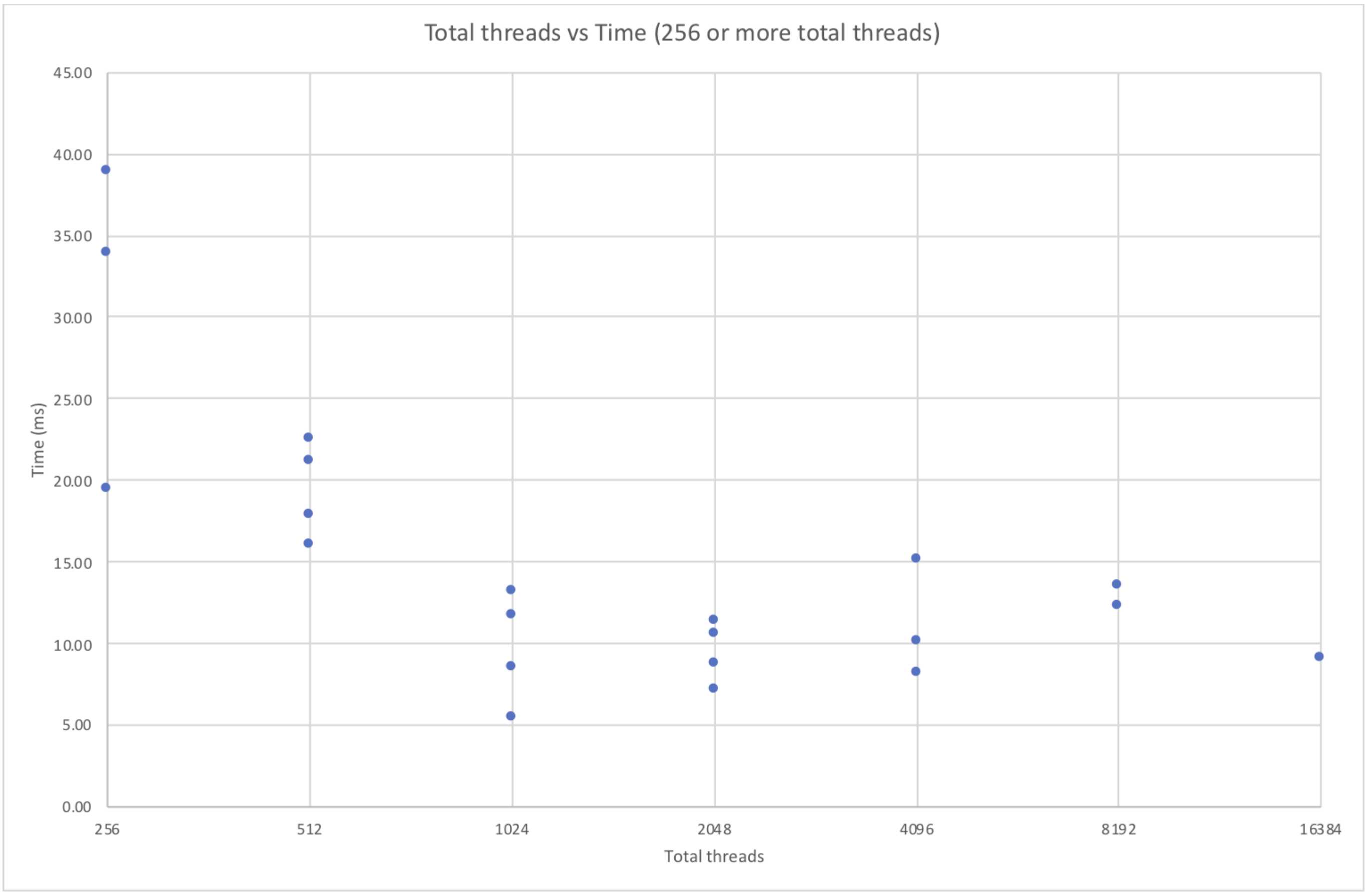

Figure 4: Scatter graph of time taken against total threads when >= 256 total threads are scheduled

5.2 Kernel Performance on XGPC

The program was run on the Jetson TX2 with blocks-per-grid values between 1 and 80 and threads-per-block values between 1 and 256.

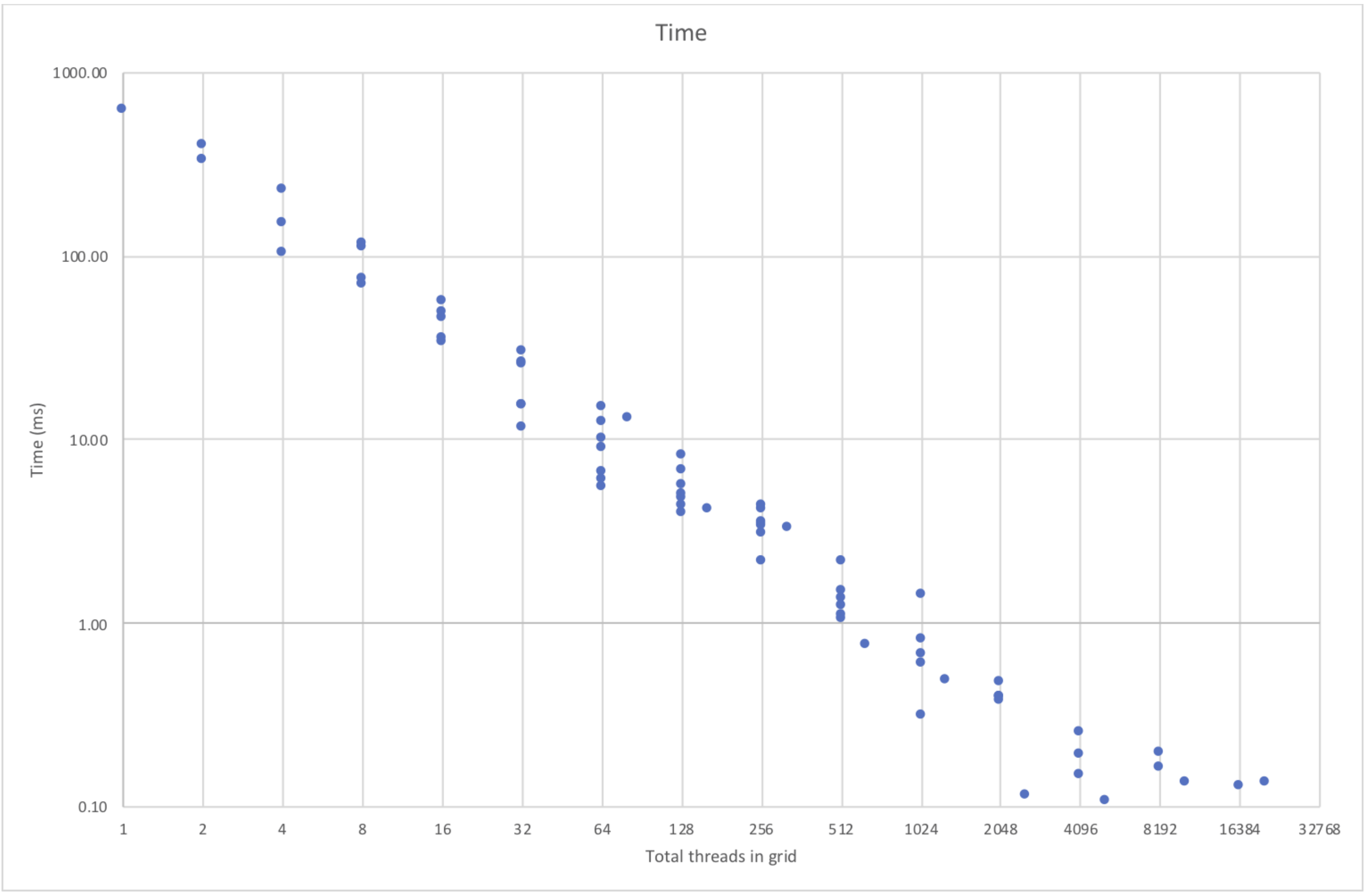

Figure 5: Scatter graph of time taken on XGPC against total threads in grid

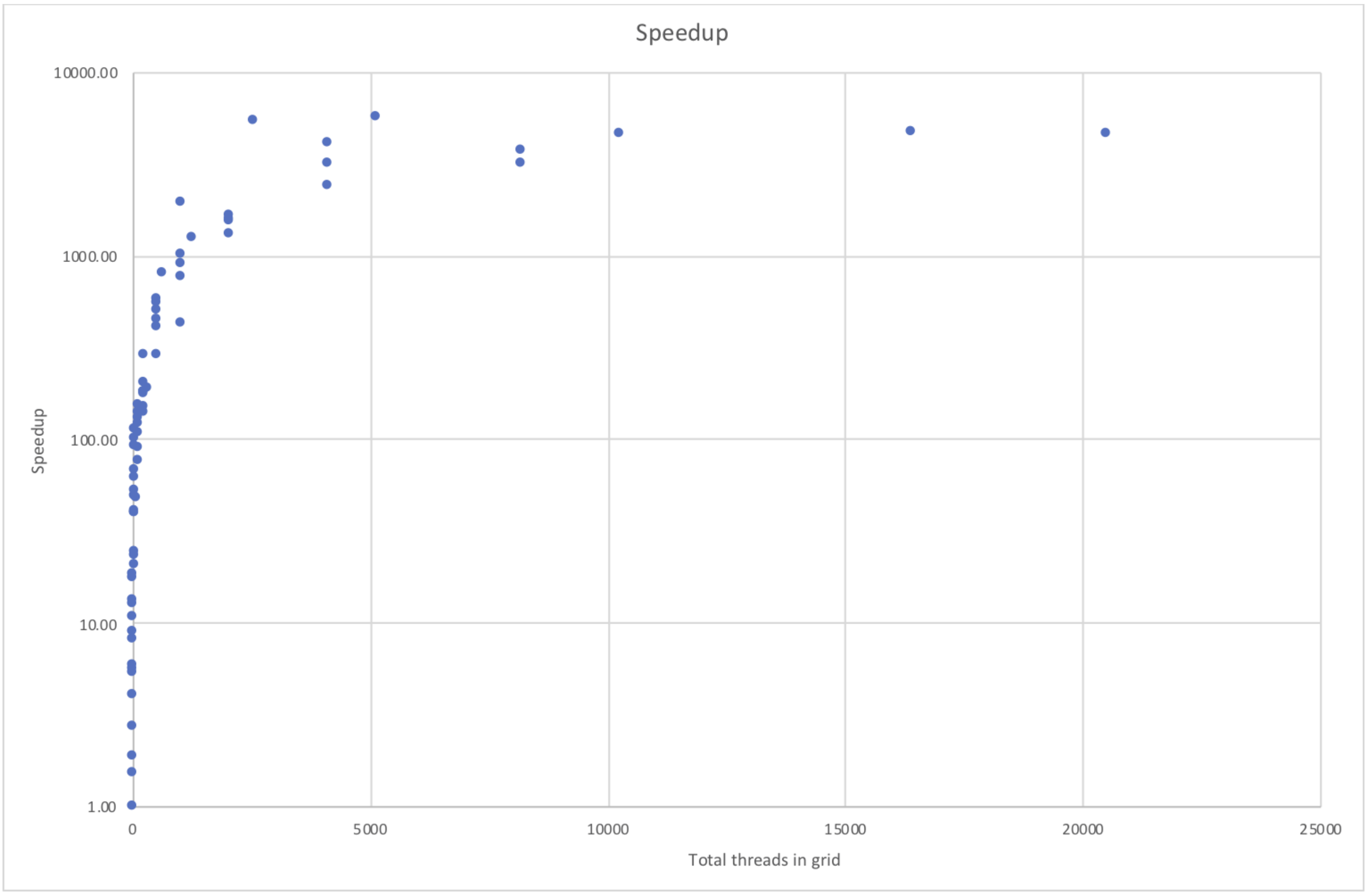

Figure 6: Scatter graph of speedup achieved on XGPC as number of threads increases

No of blocks	No of threads/block	Total threads	time/s
80	64	5120	0.11
80	128	10240	0.13
80	256	20480	0.14

Table 1: Increasing the total number of threads/block has negligible impact on performance

5.3 Kernel Stall Dependency

Node	stall_inst_fetch/%	stall_exec_dependency/%	stall_memory_dependency/%
Jetson	10.83	26.82	49.98
XGPC	15.09	55.15	17.56

Table 2: Comparison of kernel stall dependency categories according to `nvprof`

5.4 Discussion

We observe from the results above that in general, the program becomes faster when the number of total threads increases on both Jetson and XGPC. This trend can be attributed to the following reasons:

- Each block can only be executed by a single SM. Therefore, the program performs better when the number of blocks is at least equal to the number of available SMs.
- The number of threads per block should be at least equal to the number of CUDA cores on each SM to ensure that all CUDA cores are utilised.

- Performance continues to improve even as the number of threads per block exceeds the number of CUDA cores available on each SM. This is because the GPU can hide memory latency when there are extra schedulable warps by executing an alterative warp when the currently active warp is waiting for a memory operation. The large number of registers on the GPU allows it to maintain the execution state of multiple threads at once and thus perform very low cost "context switches". This behaviour is most noticeable on the Jetson TX2 because 49.98% of stalls are due to memory dependency (Table 2).

The experiments on the Jetson TX2 demonstrate the powerful ability of the GPU to hide memory latency. When considering the test cases where there are at least 2 blocks and 256 total threads, we can see that there is little correlation between read misses and execution time (Figure 3). It is more important to ensure that there is a large number of schedulable threads so that the GPU can hide memory latencies effectively (Figure 4).

Due to the extremely fast memory on the Tesla GPU, tests on the XGPC node were mostly bottlenecked by execution dependencies (Table 2). Memory access was the cause for only 17.56% of stalls. As a result, the best speedup was achieved when the program spawned exactly one thread for each CUDA core. Scheduling additional threads actually has a negligible impact on performance (Table 2 and Figure 5).

Node	Serial/ms	Parallel(best)/ms	Speedup	num_blocks	num_threads/block
Jetson (256 cores)	1747.5	5.51	317.19	32	32
XGPC (5120 cores)	618.12	0.11	5699.89	80	64

Table 3: Comparison of serial and best parallel execution time

When considering the overall performance on the Jetson and XGPC, we can see that the Tesla GPU in the XGPC is significantly more powerful than the Jetson in both serial and parallel workloads (Table 3). Do take note that the speedup is actually more than the number of CUDA cores available. Our hypothesis is that this is because the GPU can hide memory latency by scheduling other warps during execution, which is not possible for serial execution.

6 Performance Comparison with OpenMP implementation

In this section, we shall compare the performance of our CUDA application with an alternative multi-threaded CPU OpenMP implementation.

The OpenMP implementation closely follows the design of the CUDA application. The primary difference is that the kernel code is wrapped in an *`#pragma omp parallel`* block. The storing of the result is protected by a *`#pragma omp critical`* block to prevent race condition.

The following tests were executed on a compute cluster node (XGPC6). The OpenMP implementation was executed with 32 threads, matching the number of CPU threads on the machine. The CUDA implementation was executed with 80 blocks per grid and 64 threads per block, following the result of the previous section. Both implementations were run for target values between 2^{30} and 2^{48}.

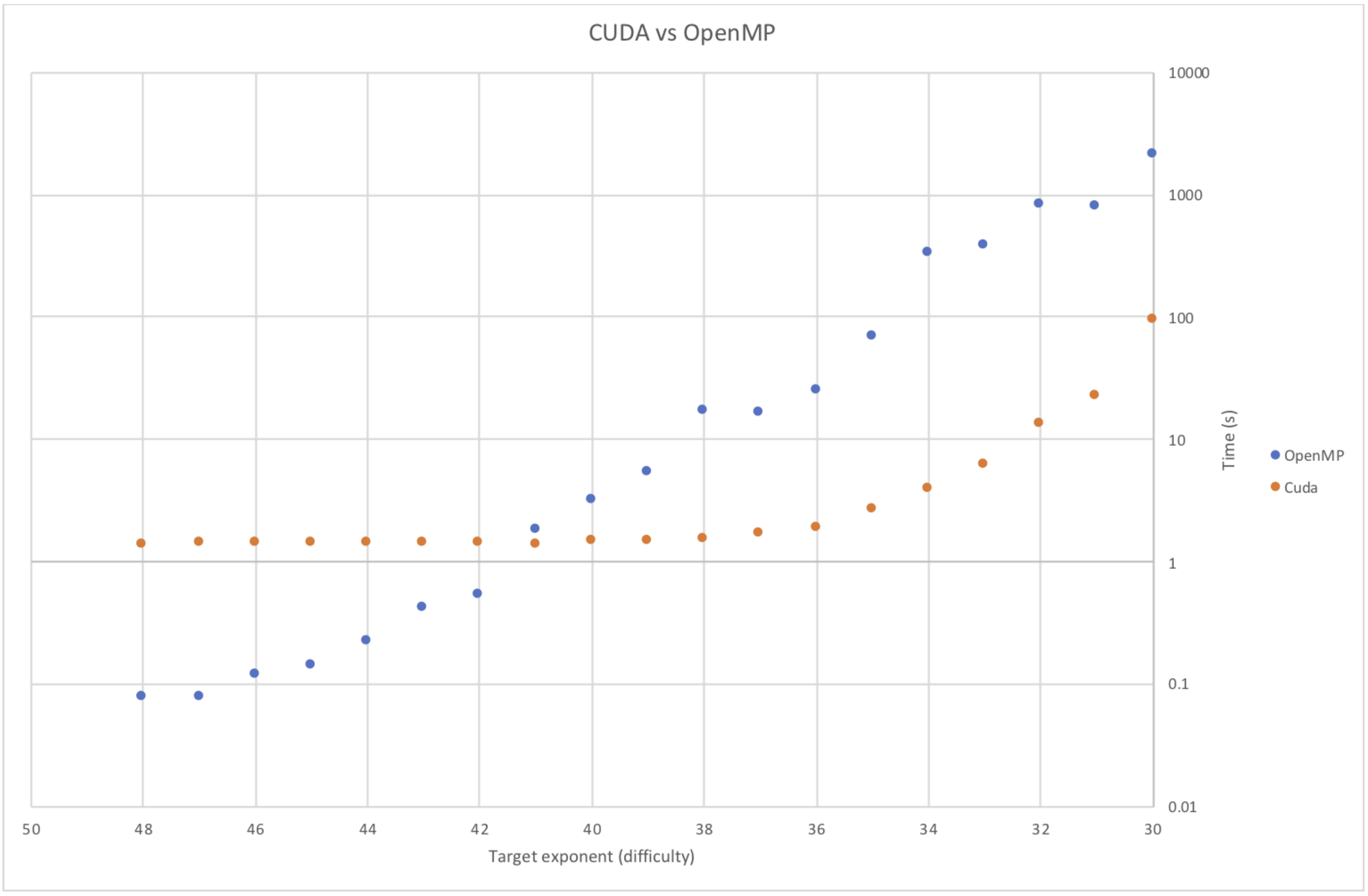

Figure 7: Scatter plot of time taken against target exponent of CUDA vs OpenMP implementations

6.1 Discussion

Figure 7 illustrates how data transfer between the host and device can be costly. Under the CUDA implementation, execution time is almost constant for targets between 2^{48} and 2^{37}. This is because the data transfer cost incurred is orders of magnitude greater than the computation cost. We know from the previous section that the kernel itself takes approximately $0.11ms$, while the total program required $1.41s$ to complete.

Since the OpenMP implementation did not have to incur this data transfer cost, it was significantly faster when the problem size was small (target was large). However, as the problem size becomes larger, the CUDA implementation took the lead. The data transfer cost was only incurred once regardless of program size, and no data transfers are necessary during the computation itself.

Thus, for larger (easier) target values whose computation time on CPU is shorter than the time taken to copy data to GPU, it is better to use the OpenMP implementation and vice versa.