Parrellel Cuda Implementation of 3D DWT

Graham Pellegrini 0352804L

Abstract—**This report details the porting of a serial Multi-Level 3D Discrete Wavelet Transform (DWT) to a parallel implementation using CUDA. Key decisions, comparative approaches, and performance results are discussed, highlighting speedup and efficiency gains. The report concludes with an evaluation of results and future work directions for further optimization.**

I. Feedback Considerations

The first assignment focused on the serial implementation of a Multi-Level 3D Discrete Wavelet Transform (DWT). For this assignment, the scope shifted to a single-level 3D DWT to emphasize parallelization using CUDA. Feedback from the previous assignment was incorporated to meet the requirements.

The report format, previously incorrect, has been updated to the IEEE Signal Processing template, with a double-column layout and minimum 11pt font size. Additionally, this report provides a more detailed analysis of testing and results, including the suggested inverse transform ('idwt.h') for comparison with the original image.

Earlier plans proposed parallelizing all three dimensions of the 3D DWT simultaneously, but feedback noted that dependencies between dimensions made this infeasible. Instead, dimensions are processed sequentially, with separate kernels for each, and synchronization ensures one dimension's transform completes before the next begins. Contrary to earlier assumptions, no synchronization is needed within kernels, as thread indexing determines the section of the volume to process. Proper block and grid calculations ensure efficient thread management within the defined scope.

Kernel calls alternate input and output pointers to avoid extra memory management or data copying. Parallelizing I/O, as previously suggested, was not prioritized because CUDA kernels focus on computation, not data handling. As such, the existing I/O headers ('savebin.h' and 'loadbin.h') remain unchanged, with memory transfers handled by other CU functions.

In the kernel, on-the-fly vectors for sum low and sum high were initially substituted with shared memory caches, along with the coefficients, to potentially improve performance. However, after further evaluation, these were reverted to the constant memory implementation, as it was found to be more efficient in terms of memory access latency and overall kernel execution time.

II. Project Structure

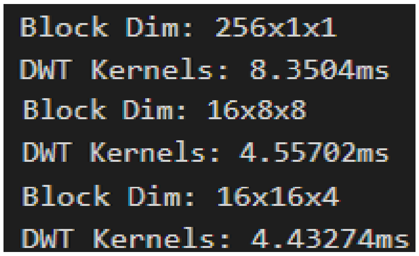
The project meets all specified requirements for build configurations and naming conventions. The implementation is logically organized, with the main functionality and DWT implementation consolidated in 'assignment-2.cu' within the src folder to streamline the Makefile and avoid separate compilation complexities. Key includes are 'idwt.h' for inverse transform functionality, 'kernels.cuh' for individual 1D DWT and volume mapping kernels, and 'loadbin.h' and 'savebin.h' for binary file handling. The Makefile, located at the root is responsible for the project compilation.

The Makefile has been built to ensure support for the CUDA by using 'nvcc' and respective cuda includes are located within the files. Separate build configurations are provided for debug and release modes, where the debug build disables optimizations, enables assertions (-DDEBUG), and includes debug symbols (-g). Conversely, the release build optimizes for performance (-O2) while excluding debug symbols and assertions. Additionally, profiling support is incorporated through dedicated targets for NVIDIA tools (ncu and nsys). Which will enable the system profile analysis to identitfy regions of the code that can be further optimized or improvements made.

III. Devision of Problem

The main challenge that defines the structure for performing parallel operations is the fact that the volume must be treated as a whole. When performing dimension-wise DWT operations, the entire volume; containing all the rows, columns, and depth—must be considered to ensure the operations are executed correctly. This directly impacts how data management is handled and the order in which kernel operations and threaded access are managed.

Previously, the serial implementation consisted of a single 'dwt 1' function that handled the transformation along one dimension. However, in the parallel implementation, three separate kernels are defined, one for each dimension, to minimize excessive copying of temporary data. The 'dwt 3d' function defines a block with a dimension size of (16, 16, 4). Initially, other iterations and tests using sizes such as (256,1,1), (16,8,8) or (8, 8, 8).



```
Block Dim: 256x1x1
DWT Kernels: 8.3504ms
Block Dim: 16x8x8
DWT Kernels: 4.55702ms
Block Dim: 16x16x4
DWT Kernels: 4.43274ms
```

Fig. 1. Different Block Dimensions Tested

However, since our volume is more likely to have significantly larger dimensions in the rows and columns than in the depth, the depth (Z-dimension) was given a quarter of the block size. This ensures that the total threads in each block remains a factor 64 (16×16×4 = 1024). The choice of block size strikes a balance between generalization, by avoiding fine-tuning for one specific test volume case, and specification by addressing CT scans and 3D images whoes depth index will be at least four times smaller than the rows and columns constructing each slice.

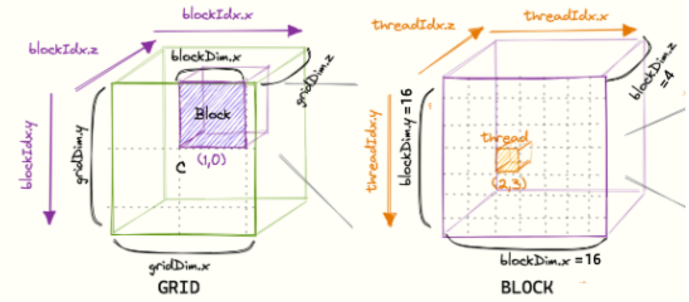


Fig. 2. Grid and Block Dimensions 3D division [1]

After defining the suitable block dimensions as (16, 16, 4), we move up and needed to compute the appropriate grid dimensions for the thread access to ensure that the entire 3D volume is accounted for. The grid dimensions are calculated by dividing the total number of rows/columns/depth by the corresponding block dimensions axis. However, since this division may not result in a whole number, the formula includes an adjustment to ensure full coverage of the data. Specifically, we add the block dimension minus one to the total size before performing integer division:

$$
\begin{aligned}
\text{Grid dimension} &= \left\lceil \frac{\text{Total size}}{\text{Block size}} \right\rceil \\
&= \frac{\text{Total size} + \text{Block size} - 1}{\text{Block size}}
\end{aligned} \tag{1}
$$

So for our 3D grids and blocks, the grid dimensions for example the rows kernel become the following:

$$
\begin{aligned}
\text{row\_grid} = \Bigg(&\left\lceil \frac{\text{rows} + \text{blockDim.x} - 1}{\text{blockDim.x}} \right\rceil, \\
&\left\lceil \frac{\text{cols} + \text{blockDim.y} - 1}{\text{blockDim.y}} \right\rceil, \\
&\left\lceil \frac{\text{depth} + \text{blockDim.z} - 1}{\text{blockDim.z}} \right\rceil \Bigg)
\end{aligned} \tag{2}
$$

With the thread access structure defined, we can examine the kernel implementation. Each kernel identifies the specific row, column, and depth indexes to process based on thread and block indexes. It validates that these indexes are within bounds, skipping transformations for out-of-bound indexes. Once validated, the kernel performs the 1D DWT operation along the respective dimension, calculating the sum of the low and high components and storing them in the flattened row-major order. Each kernel uses two float pointers in device memory: one for input data and one for temporary output data.

In the 'dwt 3d' function, these pointers are swapped after each kernel call to ensure data dependencies are correctly handled between dimensions. After the 1D DWT kernels, a mapping kernel reorders the transformed volume back to its original layout using thread indexes. This mapping function, initially designed for a multi-level implementation, was retained to ensure accurate reordering of transformed data. Once complete, the temporary volume is freed, and the final data is returned to the host.

IV. Effective Use Of Advanced Facilities

This section discusses the optimization methods implemented using CUDA facilities, with a focus on memory hierarchy. After loading the volume from the binary file, necessary data for kernel execution is ported to the device via the 'toGPU' function.

Two main types of memory are transferred: the 3D volume and the filter low and high coefficients.

The coefficients are hard-coded as 2D vectors in ‘assignment.cu‘, with the db number determining the appropriate coefficient vector. These coefficients, essential for convolution, are transferred to device memory using either shared memory or constant memory.

For shared memory, additional float pointers are created for the coefficients alongside the volume pointer. In ‘toGPU‘, the coefficients are fetched based on the db number, allocated using ‘cudaMalloc‘, and copied to global memory with ‘cudaMemcpy‘. Within the kernels, shared memory is defined locally, enabling threads within the same block to access it more quickly than global memory.

For constant memory, the coefficients are declared as constant memory in the kernel file. In ‘toGPU‘, no memory allocation is required; instead, ‘cudaMemcpyToSymbol‘ copies the coefficients to constant memory. This memory is read-only, cached on the device, and allows kernels to access it faster than global memory.

Both implementations were analyzed using the NVIDIA profiler, focusing on data transfer times and kernel performance. It was noted that during profiling, CUDA initialization timings obscured the initial ‘cudaMalloc‘ and ‘cudaMemcpyToSymbol‘ timings.

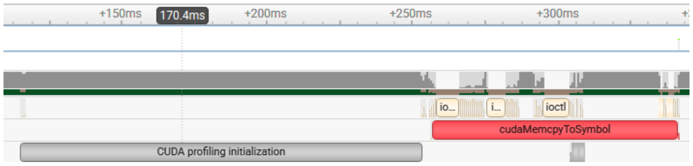
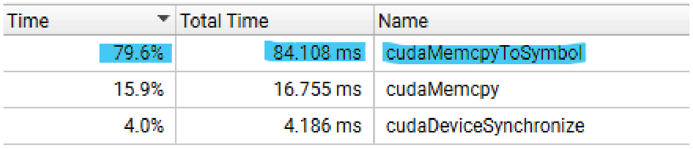


Fig. 3. cudaMemcpyToSymbol blamed for initalization time



Time	Total Time	Name
79.6%	84.108 ms	cudaMemcpyToSymbol
15.9%	16.755 ms	cudaMemcpy
4.0%	4.186 ms	cudaDeviceSynchronize

Fig. 4. Summary showing abnormally high cudaMemcpyToSymbol time

Since the initial CUDA calls were responsible for initialization, the profiler attributed their timings to initialization overhead. To address this, additional ‘cudaEvent‘ markers were added to measure the memory transfer time. These events also reflected initialization timings, but they allowed for isolating the actual time taken for memory transfers after initialization.

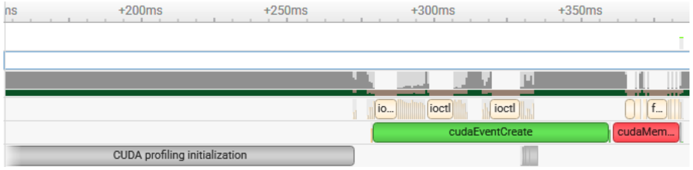
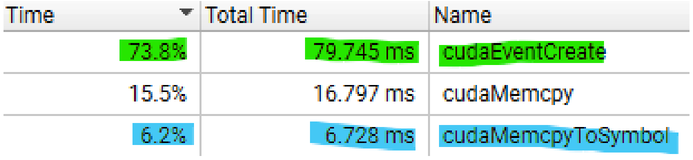


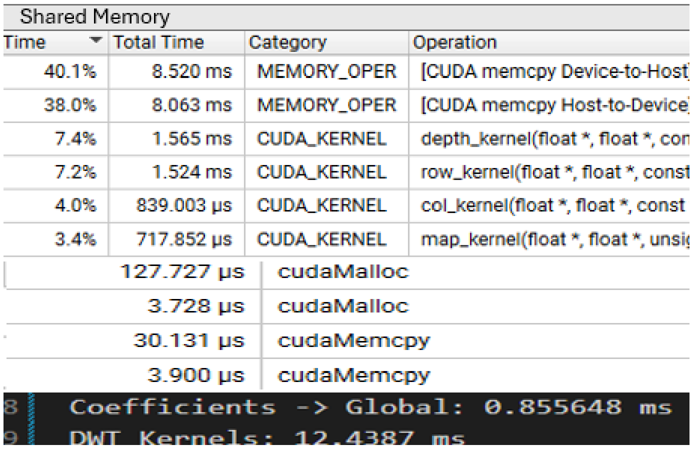
Fig. 5. cudaEvent blamed for initalization time



Time	Total Time	Name
73.8%	79.745 ms	cudaEventCreate
15.5%	16.797 ms	cudaMemcpy
6.2%	6.728 ms	cudaMemcpyToSymbol

Fig. 6. Summary showing blamed time moved to cudaEvent

Since the blaiming issue is resolved and we know why our results initially seemed to be obscured by so much we can foucs back on the comparision between shared memory and constant memory. Each case will be analysed with the NVIDIA profiler and confiremd with a cudaEvent record of the time taken for the both the memory transfer and the kernel execution.



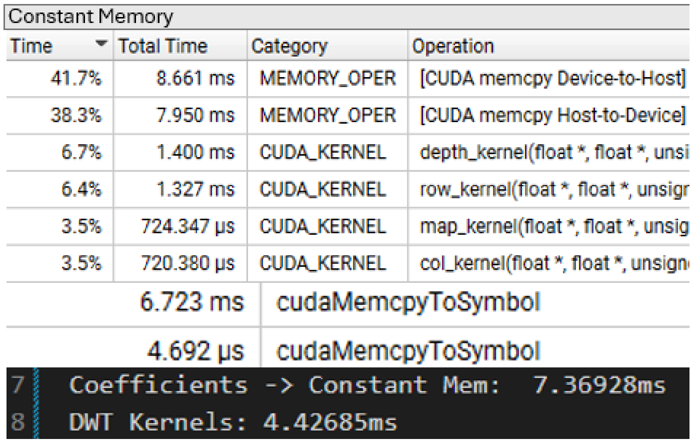
Shared Memory

Time	Total Time	Category	Operation
40.1%	8.520 ms	MEMORY_OPER	[CUDA memcpy Device-to-Host
38.0%	8.063 ms	MEMORY_OPER	[CUDA memcpy Host-to-Device
7.4%	1.565 ms	CUDA_KERNEL	depth_kernel(float *, float *, con
7.2%	1.524 ms	CUDA_KERNEL	row_kernel(float *, float *, const
4.0%	839.003 µs	CUDA_KERNEL	col_kernel(float *, float *, const
3.4%	717.852 µs	CUDA_KERNEL	map_kernel(float *, float *, unsi

Total Time	Name
127.727 µs	cudaMalloc
3.728 µs	cudaMalloc
30.131 µs	cudaMemcpy
3.900 µs	cudaMemcpy

```
Coefficients -> Global: 0.855648 ms
DWT Kernels: 12.4387 ms
```

Fig. 7. Shared Memory Profiling Results



Constant Memory

Time	Total Time	Category	Operation
41.7%	8.661 ms	MEMORY_OPER	[CUDA memcpy Device-to-Host]
38.3%	7.950 ms	MEMORY_OPER	[CUDA memcpy Host-to-Device]
6.7%	1.400 ms	CUDA_KERNEL	depth_kernel(float *, float *, unsi
6.4%	1.327 ms	CUDA_KERNEL	row_kernel(float *, float *, unsigr
3.5%	724.347 µs	CUDA_KERNEL	map_kernel(float *, float *, unsig
3.5%	720.380 µs	CUDA_KERNEL	col_kernel(float *, float *, unsign

Total Time	Name
6.723 ms	cudaMemcpyToSymbol
4.692 µs	cudaMemcpyToSymbol

```
Coefficients -> Constant Mem:  7.36928ms
DWT Kernels: 4.42685ms
```

Fig. 8. Constant Memory Profiling Results

Figures 6 and 7 confirm results consistent with theoretical expectations. The shared memory implementation shows minimal data transfer time to device memory (0.856 ms) due to the small size of the coefficients. However, kernel execution is slower because shared memory is only accessible to threads within the same block, requiring each block to replicate the shared memory. Conversely, the constant memory implementation incurs a higher transfer time for copying coefficients (7.369 ms) but benefits from device-wide access and caching, adding no overhead to kernel execution. As shown by profiler results, the constant memory implementation is faster and more efficient for this case due to reduced kernel overheads.

- **Shared Memory Time:** Data Transfer Time + Kernel Execution Time = 0.856 ms + 12.439 ms = 13.295 ms
- **Constant Memory Time:** Data Transfer Time + Kernel Execution Time = 7.369 ms + 4.427 ms = 11.796 ms
- **Speed Up:** $\frac{\text{Total Shared Memory Time}}{\text{Total Constant Memory Time}} = \frac{13.295}{11.796} = 1.127$

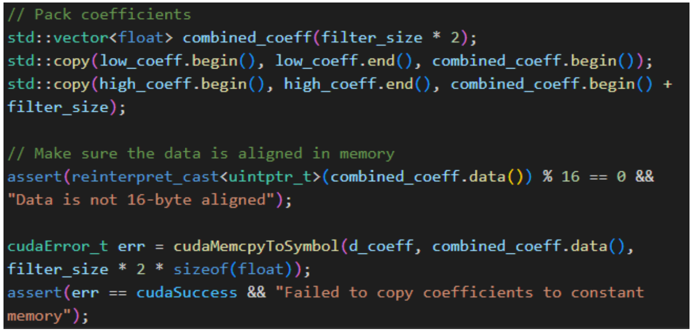
To transfer the 3D volume to device memory, it must first be flattened into a 1D vector using row-major order. This is handled within the ‘toGPU‘ function. The row-major indexing follows the formula:

$$\text{index} = \text{depth} \times \text{rows} \times \text{cols} + \text{row} \times \text{cols} + \text{col} \quad (3)$$

This indexing ensures correct mapping by iterating through columns within rows at each depth, then progressing through rows at a depth, and finally across the depths of the volume. Once flattened, the volume is allocated and copied to device memory using ‘cudaMalloc‘ and ‘cudaMemcpy‘, ensuring seamless access by the threads in the kernel. It is to be noted that the volume allocation could not be done in a more efficient manner such as using streams and concurrently executed memory transfers due to the nature of the data dependencies and the need for the entire volume to be available for the kernel execution.

After all kernel calls and the transformed volume has been mapped back to its original pointer, the data must be retrieved from device memory to the host. This is handled by the ‘toCPU‘ function, which allocates memory on the host and copies the flattened volume back using ‘cudaMemcpy‘. The volume is then reshaped into its original 3D layout using the inverse of the row-major indexing formula, ensuring the data is correctly ordered for subsequent processing or visualization.

Further analysis of the constant memory implementation revealed an important optimization opportunity. By merging the two sets of constant coefficients for the low-pass and high-pass filters into a single constant memory array, the number of cudaMemcpyToSymbol calls was reduced to one. This change decreased memory transfer time and slightly improved performance. The kernel access logic was updated accordingly, using the coefficient length to determine the starting index of the high-pass coefficients in the array.



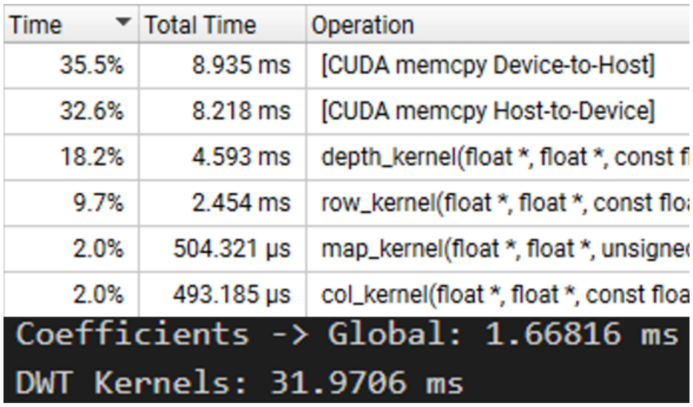
```
// Pack coefficients
std::vector<float> combined_coeff(filter_size * 2);
std::copy(low_coeff.begin(), low_coeff.end(), combined_coeff.begin());
std::copy(high_coeff.begin(), high_coeff.end(), combined_coeff.begin() +
filter_size);

// Make sure the data is aligned in memory
assert(reinterpret_cast<uintptr_t>(combined_coeff.data()) % 16 == 0 &&
"Data is not 16-byte aligned");

cudaError_t err = cudaMemcpyToSymbol(d_coeff, combined_coeff.data(),
filter_size * 2 * sizeof(float));
assert(err == cudaSuccess && "Failed to copy coefficients to constant
memory");
```

Fig. 9. Implementation of packed coefficients in a single array

Finally, the use of size t variables instead of int is recommended for CUDA development. As a 64-bit unsigned integer type, size t aligns better with device memory and improves efficiency when indexing memory locations.

V. Profile-guided optimization:

The NVIDIA profiler was previously utilized to analyze memory transfer and kernel execution times. To ensure accurate profiler results and reflect true implementation performance, all print statements and timing operations, such as cudaEvent markers, were wrapped in debug configurations. This eliminated interference from non-essential operations, resulting in more reliable profiler data.

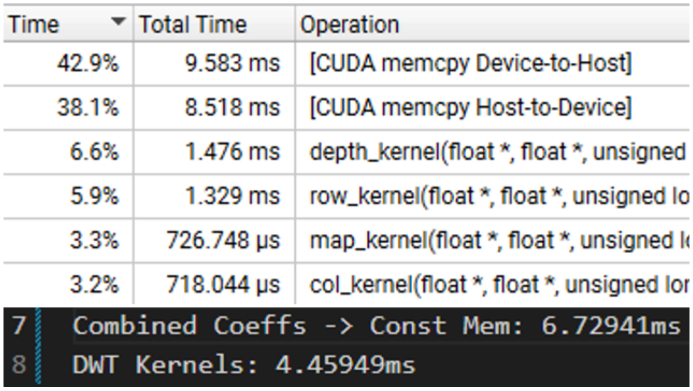


Time	Total Time	Operation
35.5%	8.935 ms	[CUDA memcpy Device-to-Host]
32.6%	8.218 ms	[CUDA memcpy Host-to-Device]
18.2%	4.593 ms	depth_kernel(float *, float *, const fl
9.7%	2.454 ms	row_kernel(float *, float *, const flo
2.0%	504.321 µs	map_kernel(float *, float *, unsigned
2.0%	493.185 µs	col_kernel(float *, float *, const floa

```
Coefficients -> Global: 1.66816 ms
DWT Kernels: 31.9706 ms
```

Fig. 10. Profiler Summary of Unoptimized Code

One major shortcoming of the unoptimized code was the high memory access latency, as threads individually accessed global memory for the coefficients. This inefficiency significantly contributed to the high kernel execution times. The profiler summary in Figure 9 highlights this issue, showing that total kernel execution times dominated the performance profile.



Time	Total Time	Operation
42.9%	9.583 ms	[CUDA memcpy Device-to-Host]
38.1%	8.518 ms	[CUDA memcpy Host-to-Device]
6.6%	1.476 ms	depth_kernel(float *, float *, unsigned
5.9%	1.329 ms	row_kernel(float *, float *, unsigned lo
3.3%	726.748 µs	map_kernel(float *, float *, unsigned l
3.2%	718.044 µs	col_kernel(float *, float *, unsigned lor

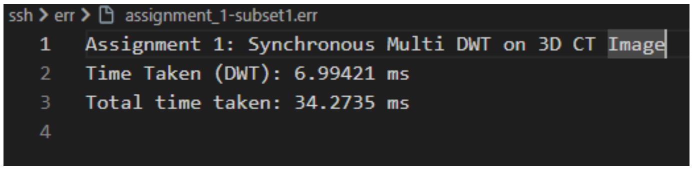
```
Combined Coeffs -> Const Mem: 6.72941ms
DWT Kernels: 4.45949ms
```

Fig. 11. Profiler Summary of Optimized Code

The final profiler results demonstrate substantial improvements over the unoptimized version. By leveraging constant memory, we introduced a minor overhead during the memory copy to symbols. However, this was outweighed by a significant reduction in kernel execution times, resulting in a marked decrease in the total execution time.

VI. Computation of Speedup

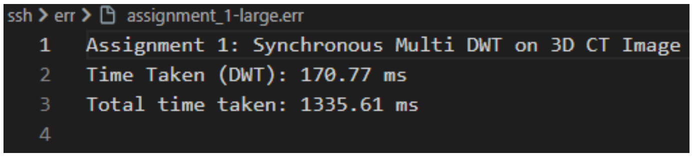
This section evaluates the speedup achieved through parallelization by comparing the timing results of the reference (serial) and parallel implementations. A subset of the CHAOS dataset was preprocessed in Python to create a smaller version with one-fourth the dimensions of the large dataset. Specifically, the large dataset has dimensions of $512 \times 512 \times 78$ (columns, rows, depth), while the smaller dataset has dimensions of $128 \times 128 \times 20$.



```
Assignment 1: Synchronous Multi DWT on 3D CT Image
Time Taken (DWT): 6.99421 ms
Total time taken: 34.2735 ms

```

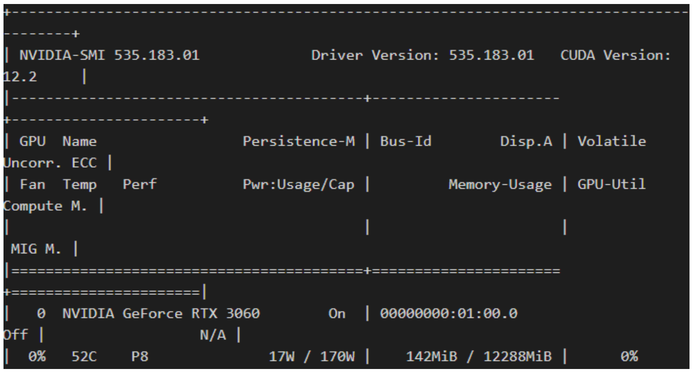
Fig. 12. Timing results for reference implementation (small set)

The small dataset executes faster than the large one due to fewer iterations in the serial implementation. While the parallel implementation follows a similar trend, the difference might be less pronounced, as it benefits more from optimized block execution than reduced iterations.



```
Assignment 1: Synchronous Multi DWT on 3D CT Image
Time Taken (DWT): 170.77 ms
Total time taken: 1335.61 ms

```

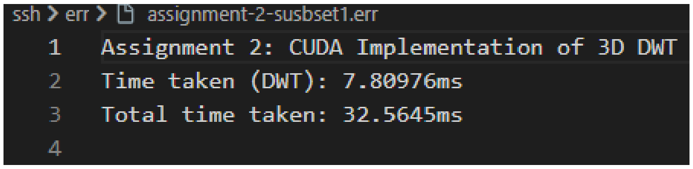
Fig. 13. Timing results for reference implementation (large set)



```
+---------------------------------------------------------------------------
--------+
| NVIDIA-SMI 535.183.01             Driver Version: 535.183.01   CUDA Version:
12.2     |
|-----------------------------------------+----------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile
Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util
Compute M. |
|                                         |                      |
 MIG M. |
|=========================================+======================
+======================|
|   0  NVIDIA GeForce RTX 3060        On  | 00000000:01:00.0
Off |                   N/A |
|  0%   52C    P8              17W / 170W |    142MiB / 12288MiB |      0%
```

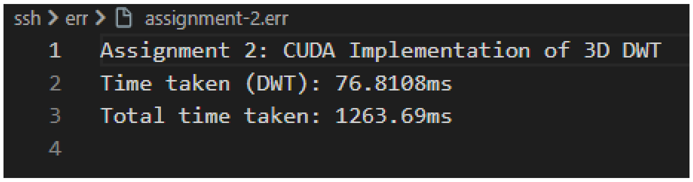
Fig. 14. Appropriate selection of device for parallel results

The snippet in Figure 13 shows that the appropriate device is being selected as '.sh' file includes 'nvidia-smi' to output the device information. This ensures that the parallel results are being executed on the correct device as stated in the requirements.



```
Assignment 2: CUDA Implementation of 3D DWT
Time taken (DWT): 7.80976ms
Total time taken: 32.5645ms

```

Fig. 15. Timing results for parallel implementation (small set)



```
Assignment 2: CUDA Implementation of 3D DWT
Time taken (DWT): 76.8108ms
Total time taken: 1263.69ms

```

Fig. 16. Timing results for parallel implementation (large set)

The parallel implementation shows a significant speedup over the serial implementation. The actual comparision will be done in the following section where it will be compared with the theoretical speedup acievable.

VII. Comparison Of Actual and Potential Speedup

To estimate the theoretically available speedup, we use Amdahl's Law:

$$S = \frac{1}{(1-P)+\frac{P}{N}} \tag{4}$$

Where:

- S is the theoretical speedup.
- P is the proportion of the code that is parallelizable.
- N is the number of cores available.

Using the following hardware analysis:

```
NVIDIA-SMI 535.183.01
Driver Version: 535.183.01
CUDA Version: 12.2

GPU: NVIDIA GeForce RTX 3060
CUDA Cores: 3584 (approx)
Memory: 12GB
```

The RTX 3060 has approximately $N = 3584$ CUDA cores. Let us assume that the parallelizable portion of the code which is mainly composed of the 1D DWT operations is $P = 0.8$ (80%). Using Amdahl's Law:

$$S = \frac{1}{(1-0.8)+\frac{0.8}{3584}} = 4.994 \tag{5}$$

Thus, the theoretical speedup with full parallelization and utilization of all available CUDA cores is approximately $S \approx 5.0$. Assuming that 80% of the kernel is parallelizable and the overhead for memory transfers is negligible, the parallel implementation can achieve a 5x improvement over the serial implementation.

However, this theoretical speedup represents an idealized scenario and does not account for the task division and other practical factors. Kernel execution times depend heavily on data volume, thread count, block size, and data access patterns. Additionally, the data dependencies precent pose a limitation to full parallelization and speedup.

Furthermore, the for parallelization using CUDA, the memory bandwidth is a critical factor that can limit the speedup. The memory bandwidth must be contantly fed with data to keep the CUDA cores busy. But can be a bottleneck if the data transfer rate is slower than the processing rate of the CUDA cores.

The actual speedup achieved in the parallel implementation is calculated as:

$$\text{Speedup (large)} = \frac{\text{Serial Time}}{\text{Parallel Time}} = \frac{170.77}{76.81} = 2.223 \tag{6}$$

$$\text{Speedup (small)} = \frac{\text{Serial Time}}{\text{Parallel Time}} = \frac{6.99}{7.81} = 0.895 \tag{7}$$

These results show that the theoretical speedup calculated using Amdahl's Law was significantly overestimated. Additionally, the actual speedup is lower, and for the small dataset, there is a slowdown due to memory transfer overhead. This indicates that the parallel implementation is only effective for large datasets, where the overhead is amortized over a greater number of iterations.