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.

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 fixed block size of (256) or equal dimensions of (8, 8, 8). 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.

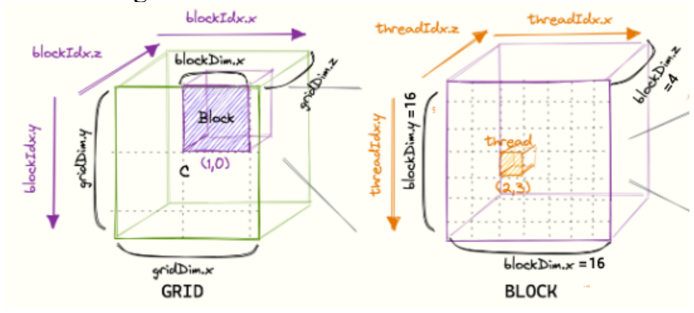


Fig. 1. Grid and Block Dimensions 3D division [1]

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.

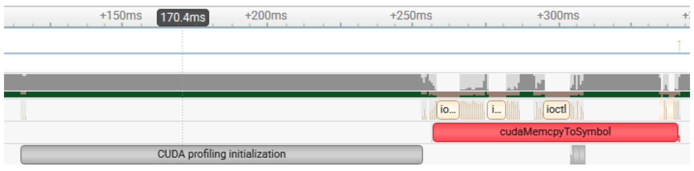
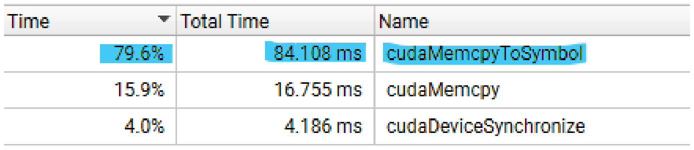


Fig. 2. cudaMemcpyToSymbol blamed for initalization time



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

Fig. 3. 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.

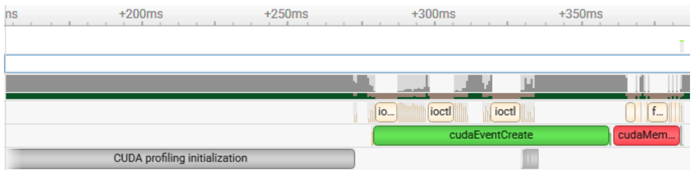
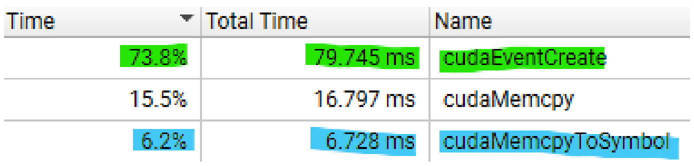


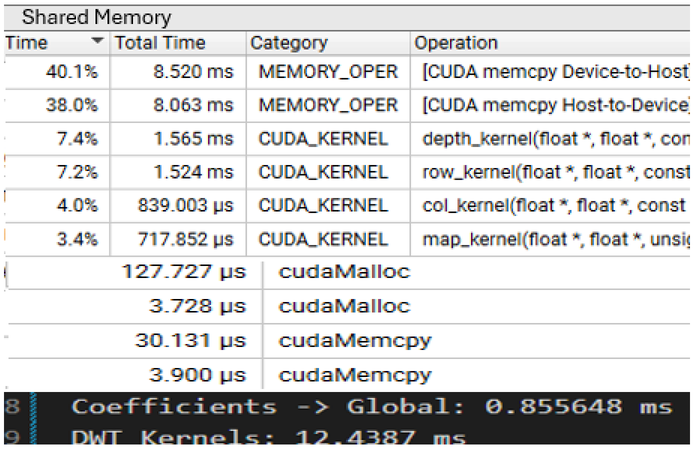
Fig. 4. cudaEvent blamed for initalization time



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

Fig. 5. 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.



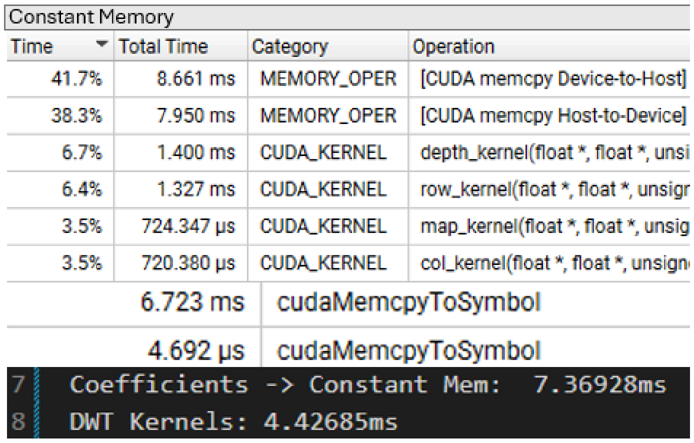
Shared Memory

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

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. 6. 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

6.723 ms	cudaMemcpyToSymbol
4.692 μs	cudaMemcpyToSymbol

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

Fig. 7. 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:

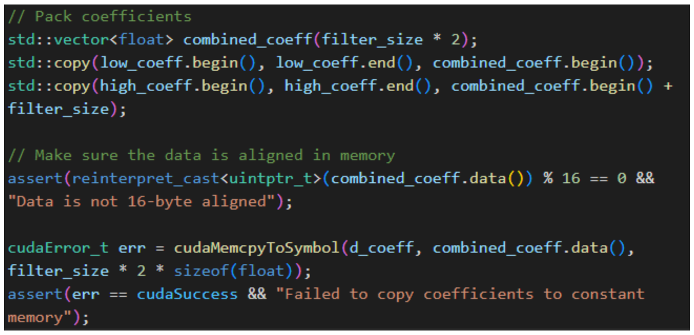
$$\text{index} = \text{depth} \times \text{rows} \times \text{cols} + \text{row} \times \text{cols} + \text{col} \quad (3)$$

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.

Lastly, 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.

V. Profile-guided optimization:

The NVIDIA profiler was previously utilized to analyze memory transfer and kernel execution times. However, further analysis of the constant memory implementation uncovered a minor but significant 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 is reduced to one. This consolidation decreases memory transfer time and yields a slight performance improvement. The updated single constant memory array is also reflected in how the kernel accesses the coefficients, with the coefficient length used to determine the index where the high-pass coefficients begin 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. 8. Code implemented to pack the coefficients into a single array

Furthermore, to saturate the profiler results and show the true performance of the implementation asserts and debug symbols encased any type of prints and timing operations like the ’cudaEvent’ markers. This ensures that the results are not skewed by any other operations other than those necessary for output. The profiler results are now more accurate and show the true performance of the implementation.

Finally the noted use of size t variables can be considered as a better practice than int variables when using the CUDA architecture. This is due to size t being a 64-bit unsigned integer type, which is easily aligned in the device memory and can be used to index the memory locations more efficiently.

Appendix A
Appendix Title

Appendix text goes here.

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.