ECE 4822: Engineering Computation IV

Homework No. 7: Parallel DSP on a GPU

By Shahzad Khan

The goal of this assignment is to take our filter code from Hw_06 to a GPU and parallel it with that instead of OpenMP.

The process of porting my OpenMP code to CUDA was relatively simple, the reason is because I used similar techniques to split up the between 1 GPU. With the FIR filter because there isn't any feedback loop this allows for me to plug it basically straight into the GPU.

```
 98   // Launch enough threads
 99   int threadsPerBlock = 256;
100   int blocks = (numSamples + threadsPerBlock - 1) / threadsPerBlock;
101   fir_filter_cuda<<<blocks, threadsPerBlock>>>(d_samples, d_coef, d_output, numSamples,
numCoef);
102   // Stop timing
103   cudaEventRecord(stop);
104   cudaEventSynchronize(stop);
105   cudaDeviceSynchronize();
```

We just have to calculate the threads and blocks and change the function to a __global__

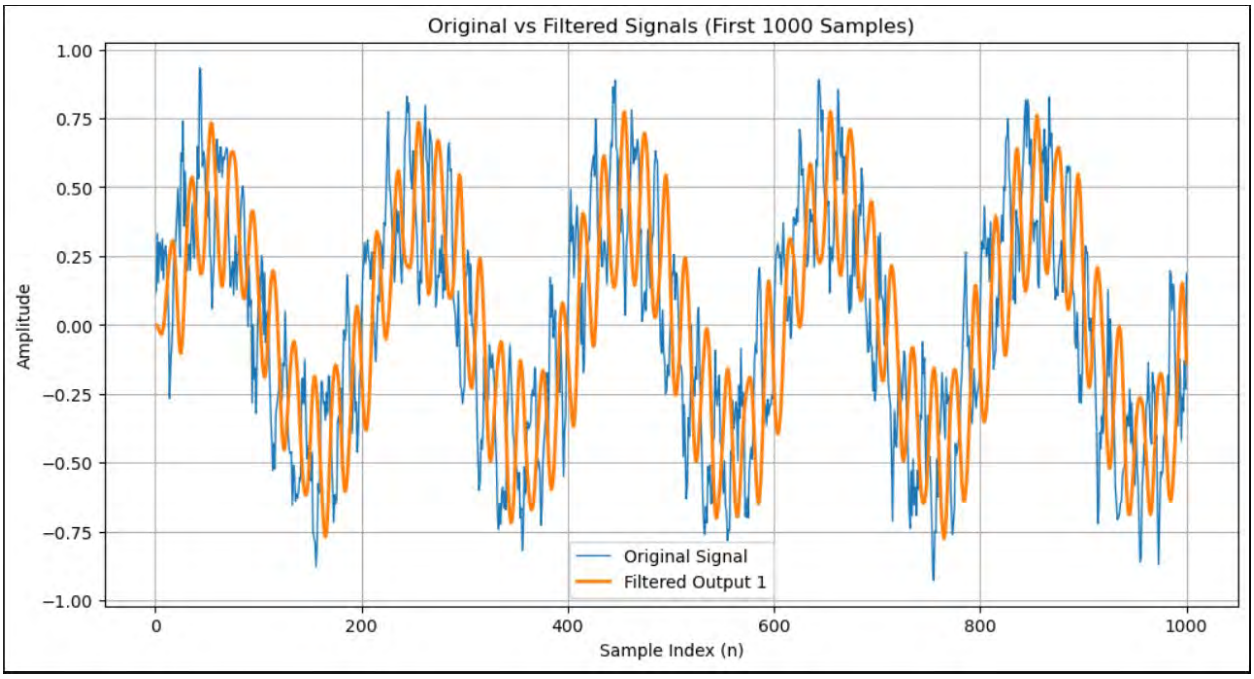
As for the timing of the program we see that

```
CUDA kernel execution time:
22.524481 ms
```

For my giant_text file which is about 1GB. Comparing this to my OpenMP FIR filter

```
nedc_130_[1]: time ./x.exe ../../hw_07/sine_signal.txt
coef.txt
real    0m35.343s
user    0m48.448s
sys     0m1.862s
```

Which is a significant increase, which is expected and we have the same filter output



Then looking at the IIR Filter

The CUDA kernel implements an IIR filter by computing the output signal y[n] from the input x[n] using feed-forward (b) and feedback (a) coefficients. Each block processes a contiguous segment of the input sequentially to preserve the recursive dependency of the filter, while threads within a block compute partial sums of the coefficients in parallel. Shared memory stores the input and output histories for fast access, and atomic operations combine partial sums safely. This approach balances the sequential nature of IIR filtering with intra-block parallelism, enabling efficient GPU execution while maintaining correct results.

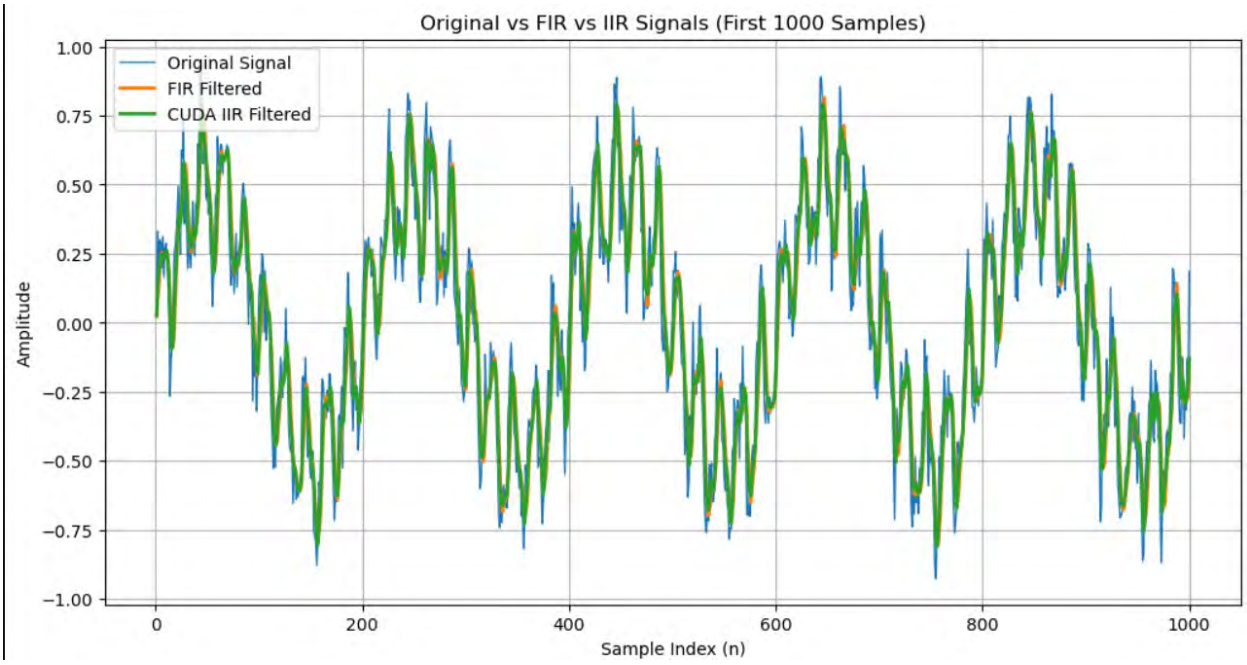
Now timing this code we see

Kernel execution time: 7425.291504 ms

Now our previous hw_06 code

```
nedc_130_[1]: time ./x.exe ../../hw_07/sine_signal.txt b.txt a.txt

real    0m35.284s
user    0m42.802s
sys     0m2.108s
```



For OpenMP on multiple CPUs compared to GPU acceleration, there are a few key trade-offs. CPUs have fewer, but more powerful, cores, often useful for tasks with heavy branching or sequential dependencies. OpenMP has simple parallelization over those cores, but scaling is necessarily limited by the number of available CPU threads and memory bandwidth. GPUs possess thousands of lightweight cores that are optimized for data-parallel operations; hence, they are well-suited to tasks such as FIR filtering where every output can be calculated independently. However, the transfer of memory and the configuration of threads/blocks on the GPU have to be carefully managed, and algorithms with sequential dependencies, such as IIR filters, cannot utilize such parallelism on a GPU. From a practical perspective, if the workload is large and highly parallel, then much higher throughput comes from the GPU, while CPUs may perform better for small workloads or algorithms where strong sequential components exist in the workload.

In this assignment, I successfully ported FIR and IIR filters to CUDA, demonstrating how GPU parallelism can accelerate DSP computations. The FIR filter benefited the most, as its feed-forward structure allows each output to be computed independently, resulting in substantial speedup over the CPU implementation. The IIR filter, while limited by its recursive dependency, still achieved meaningful performance gains through intra-block parallelism on filter coefficients and efficient use of shared memory. Overall, this project highlights the power of GPUs for large-scale signal processing tasks, while also emphasizing the importance of understanding algorithm dependencies when designing parallel implementations.