CUDA IMPLEMENTATION ON FINITE IMPULSE RESPONSE

High-Performance Computing Project Report

Problem Statement: Parallel simulation of moving average finite impulse response filter

Faculty guide: Dr. Noor Mahammad

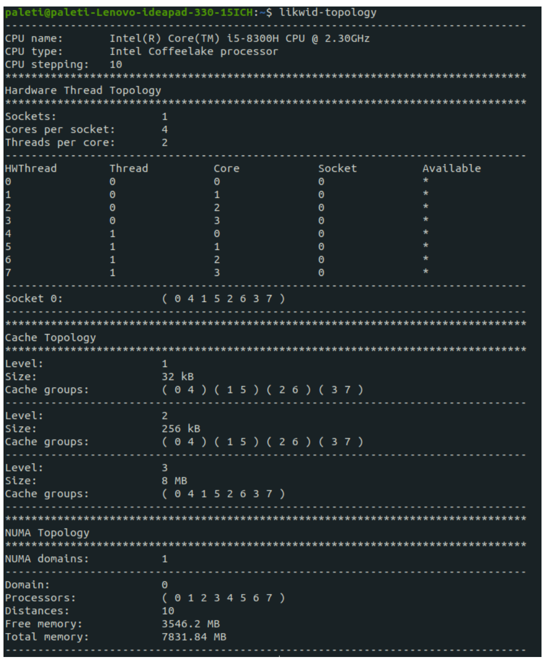
By,

PALETI KRISHNASAI

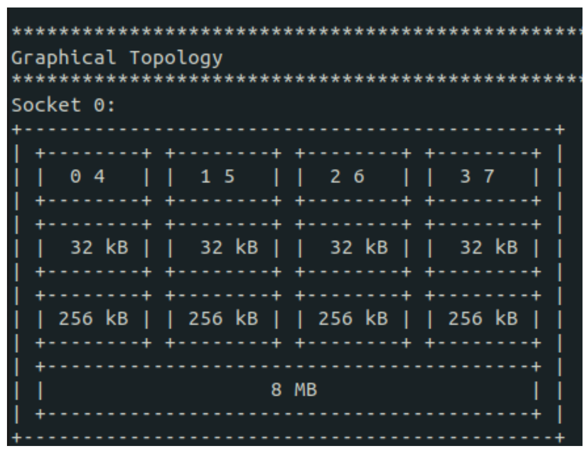
CED18I039

Hardware Configuration:

PU NAME: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
Number of Sockets: 1
Cores per Socket: 4
Threads per core: 2
L1d cache: 128 KiB
L1i cache: 128 KiB
L2 cache: 1 MiB
L3 cache: 8 MiB



```
paleti@paleti-Lenovo-ideapad-330-15ICH:~$ likwid-topology
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
CPU type:       Intel Coffeelake processor
CPU stepping:   10
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                1
Cores per socket:       4
Threads per core:       2
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1               0               1               0               *
2               0               2               0               *
3               0               3               0               *
4               1               0               0               *
5               1               1               0               *
6               1               2               0               *
7               1               3               0               *
--------------------------------------------------------------------------------
Socket 0:               ( 0 4 1 5 2 6 3 7 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   32 kB
Cache groups:           ( 0 4 ) ( 1 5 ) ( 2 6 ) ( 3 7 )
--------------------------------------------------------------------------------
Level:                  2
Size:                   256 kB
Cache groups:           ( 0 4 ) ( 1 5 ) ( 2 6 ) ( 3 7 )
--------------------------------------------------------------------------------
Level:                  3
Size:                   8 MB
Cache groups:           ( 0 4 1 5 2 6 3 7 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:           1
--------------------------------------------------------------------------------
Domain:                 0
Processors:             ( 0 1 2 3 4 5 6 7 )
Distances:              10
Free memory:            3546.2 MB
Total memory:           7831.84 MB
--------------------------------------------------------------------------------
```



```
*******************************************
Graphical Topology
*******************************************
Socket 0:
+-----------------------------------------+
| +-------+ +-------+ +-------+ +-------+ |
| |  0 4  | |  1 5  | |  2 6  | |  3 7  | |
| +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ |
| | 32 kB | | 32 kB | | 32 kB | | 32 kB | |
| +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ |
| |256 kB | |256 kB | |256 kB | |256 kB | |
| +-------+ +-------+ +-------+ +-------+ |
| +-------------------------------------+ |
| |                 8 MB                | |
| +-------------------------------------+ |
+-----------------------------------------+
```

INTRODUCTION

In signal processing, a finite impulse response (FIR) filter is a filter whose impulse response (or response to any finite length input) is of *finite* duration, because it settles to zero in finite time. This is in contrast to infinite impulse response (IIR) filters, which may have internal feedback and may continue to respond indefinitely (usually decaying).

The impulse response (that is, the output in response to a Kronecker delta input) of an N^{th}-order discrete-time FIR filter lasts exactly *N + 1* samples (from first nonzero element through last nonzero element) before it then settles to zero.

For a causal discrete-time FIR filter of order *N*, each value of the output sequence is a weighted sum of the most recent input values:

$$\begin{aligned} y[n] &= b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N] \\ &= \sum_{i=0}^{N} b_i \cdot x[n-i], \end{aligned}$$

- $x[n]$ is the input signal,
- $y[n]$ is the output signal,
- N is the filter order; an N^{th}-order filter has $N+1$ terms on the right-hand side
- b_i is the value of the impulse response at the *i*'th instant for $0 \le i \le N$ of an N^{th}-order FIR filter. If the filter is a direct form FIR filter then b_i is also a coefficient of the filter.

MOVING AVERAGE FIR FILTER ANALYSIS

A moving average filter is a very simple FIR filter. It is sometimes called a boxcar filter, especially when followed by decimation. The filter coefficients, $b_0, \ldots, b_N$, are found via the following equation:

$$b_i = \frac{1}{N+1}$$

To provide a more specific example, we select the filter order:

$$N = 2$$

The impulse response of the resulting filter is:

$$h[n] = \frac{1}{3}\delta[n] + \frac{1}{3}\delta[n-1] + \frac{1}{3}\delta[n-2]$$

Parallel Code [CUDA] [comments included to explain parallization]

```
/*
Author : Paleti Krishnasai CED18I039
Simulation of N-order moving average FIR filter
    N : order of the filter
    n : instance
    filter equation : output_signal[n] = (input_signal[n-1] +
input_signal[n] + input_signal[n1+1]) / (N+1)

    The FIR function under consideration is
        y[n] = (x[n+1] + x[n] + x[n-1]) / 6
*/
```

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include<stdlib.h>
#include<time.h>
#include<math.h>
#define n_size 1000
// kernel has the loop  called the most number of times in the serial
// code(based on profiling)
_global__ void generate(float *inputsignal, float *outputsignal)
{
int index=threadIdx.x+blockIdx.x*blockDim.x;
if(index>=1 && index < 100000)
{
 output_signal[index] = (inputsignal[index-1] + inputsignal[index] +
input_signal[index+1]) * 0.142857143;
}
}

int main() {
 float inputsignal[n_size], outputsignal[n_size]={0};
cudaEvent_t start, end;
// host copies of variables a, b & c
float *d_inputsignal, *d_outputsignal;
// device copies of variables a, b & c
```

```
int size = n_size*sizeof(float);
// Allocate space for device copies of a, b, c
cudaMalloc((void **)&d_inputsignal, size);
cudaMalloc((void **)&d_outputsignal, size);
// Create Event for time
cudaEventCreate(&start);
cudaEventCreate(&end);
// Setup input values
for (int i = 0; i < size; i++)
 {
   float random_input =  rand()%1000;
   input_signal[i] =i*random_input;

 }
// Copy inputs to device
cudaMemcpy(d_inputsignal, &inputsignal, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_outputsignal, &outputsignal, size, cudaMemcpyHostToDevice);
int Thread[]={1,2,4,6,8,10,12,16,20,32,64,128,150};
int thread_arr_size=13;

// edge cases ( first case and last case )
outputsignal[0] = ( (inputsignal[0] + inputsignal[1]) ) * 0.142857143;
outputsignal[size - 1] = ( (inputsignal[size - 2] + inputsignal[size - 1])
)* 0.142857143;

for(int i=0;i<thread_arr_size;i++)
{
      int Threads=Thread[i];
      cudaEventRecord(start);
      // Launch add() kernel on GPU
      generate<<<n_size/Threads,Threads>>>(d_inputsignal, d_outputsignal);
      cudaEventRecord(end);
      cudaEventSynchronize(end);
      float time = 0;
      cudaEventElapsedTime(&time, start, end);
      // Copy result back to host
      cudaError err = cudaMemcpy(&outputsignal, d_outputsignal, size,
cudaMemcpyDeviceToHost);
      if(err!=cudaSuccess) {
```

```
        printf("CUDA error copying to Host: %s\n",
cudaGetErrorString(err));
        }


        printf("Time Taken by the program for %d
Threads=%f\n",Threads,time);



}
// Cleanup
cudaFree(d_inputsignal);
cudaFree(d_outputsignal);
return 0;
}
```

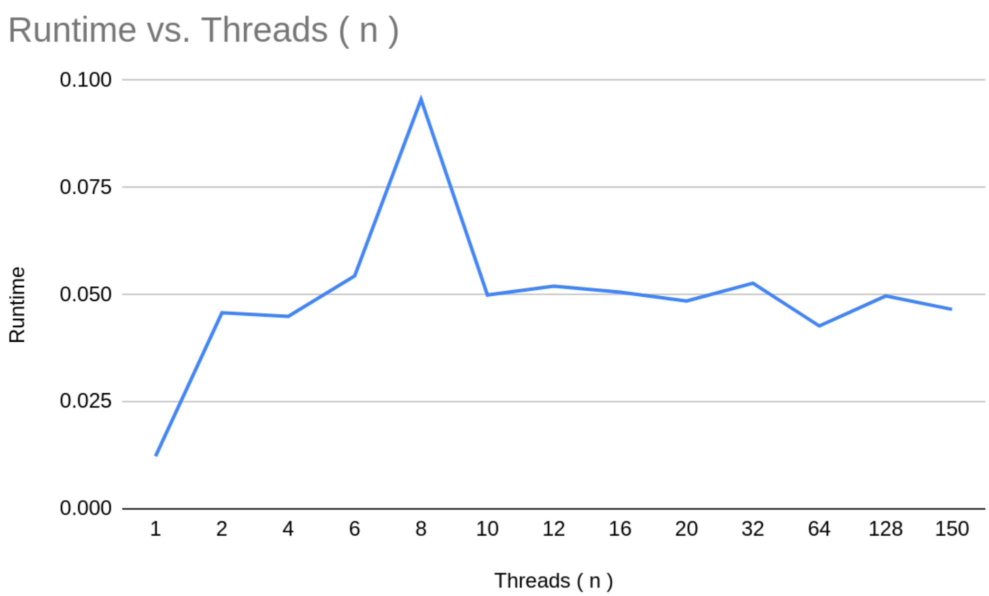
Observations:

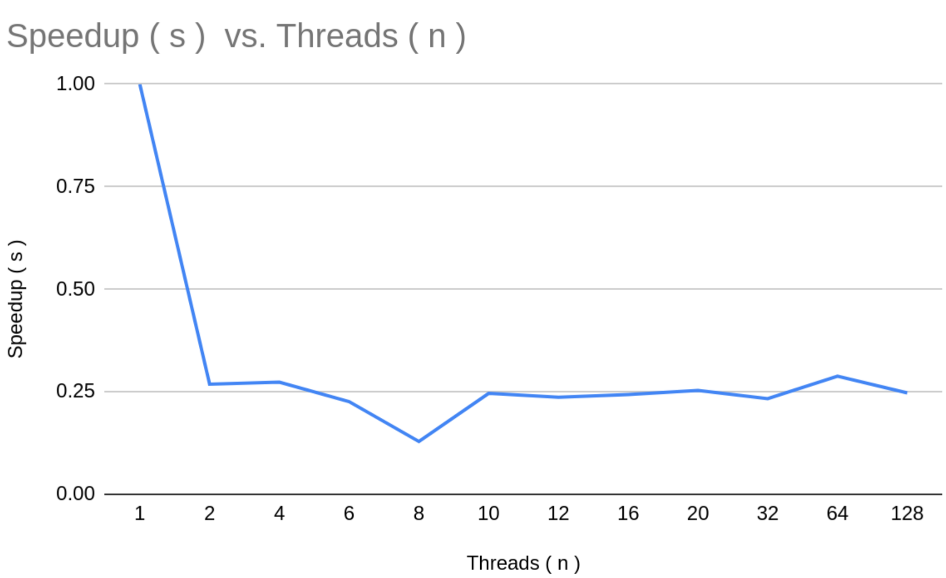
DA	Threads (n)	Runtime	Speedup (s)	Parallelization Fraction	1 - 1/s	1 - 1/n
N	1	0.012288	1			
N/2	2	0.04576	0.2685314685	-5.447916667	-2.723958333	0.5
N/4	4	0.044928	0.2735042735	-3.541666667	-2.65625	0.75
N/6	6	0.054368	0.2260153031	-4.109375	-3.424479167	0.8333333333
N/8	8	0.095584	0.1285570807	-7.74702381	-6.778645833	0.875
N/10	10	0.04992	0.2461538462	-3.402777778	-3.0625	0.9
N/12	12	0.051968	0.236453202	-3.522727273	-3.229166667	0.9166666667
N/16	16	0.05056	0.2430379747	-3.322222222	-3.114583333	0.9375
N/20	20	0.048512	0.253298153	-3.103070175	-2.947916667	0.95
N/32	32	0.052672	0.2332928311	-3.392473118	-3.286458333	0.96875
N/64	64	0.042688	0.287856072	-2.513227513	-2.473958333	0.984375
N/128	128	0.049696	0.2472633612	-3.06824147	-3.044270833	0.9921875
N/150	150	0.04656	0.2639175258	-2.80778104	-2.7890625	0.9933333333

Speed up can be found using the following formula,
S(n)=T(1)/T(n)
where, S(n) = Speedup for thread count ‘n’
T(1) = Execution Time for Thread count ‘1’ (serial code)
T(n) = Execution Time for Thread count ‘n’ (serial code)

Parallelization Fraction can be found using the following formula,
S(n)=1/((1 - p) + p/n)
where, S(n) = Speedup for thread count ‘n’
n = Number of threads
p = Parallelization fraction

```
Time Taken by the program for 1 Threads=0.012288
Time Taken by the program for 2 Threads=0.045760
Time Taken by the program for 4 Threads=0.044928
Time Taken by the program for 6 Threads=0.054368
Time Taken by the program for 8 Threads=0.050112
Time Taken by the program for 10 Threads=0.049920
Time Taken by the program for 12 Threads=0.051968
Time Taken by the program for 16 Threads=0.050560
Time Taken by the program for 20 Threads=0.048512
Time Taken by the program for 32 Threads=0.052672
Time Taken by the program for 64 Threads=0.042688
Time Taken by the program for 128 Threads=0.049696
Time Taken by the program for 150 Threads=0.046560
```





Inference:

(Note: Execution time, graph, and inference will be based on hardware configuration)

- The code throws a segfault when the size exceeds 1000.
- The runtime and speedup trends are very erratic and up and down.
- There is no speedup observed.

[OpenGL plotting is not implemented in MPI and CUDA projects as OpenGL installation proved to be difficult in virtual machines and colab]
