Introduction to Parallel and Distributed Processing

Assignment 3

Author : Kavitha Elizebeth Varughese
UBID : 50560694

Overview

In this assignment, we design and implement an efficient CUDA program for the fast application of Gaussian Kernel Density Estimates. We will compare the runtimes of our solutions between our CUDA GPU model and our sequential model.

Problem Description

Given a vector $\boldsymbol{X} = [x_1, x_2, \dots, x_n]$, We estimate the following

$$\hat{}f_h(x) = \frac{1}{n \cdot h} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

where,

$$K(x) = \frac{1}{2\pi} e^{\frac{-x^2}{2}}$$

Thus, we need to compute $\boldsymbol{Y} = [y_1, y_2, \dots, y_n]$ where $y_i = \hat{}f_h(x_i)$ for some predefined bandwidth h.

Proposed Solution

The computation of every value of y_i is dependent on all values of $\boldsymbol{X}$. Pulling the values from the global memory every time will not give us the performance we want. So, we will try to leverage shared memory for performance.

We cannot store the entire vector $\boldsymbol{X}$ in our shared memory, as the size could be larger than our shared memory. So, we need to a efficient algorithm, which will allow us to save a part of $\boldsymbol{X}$ in the shared memory and keep overwriting the shared memory to complete the computation.

Streamlined Process for Parallelization

Instead of loading the entire vector $\boldsymbol{X}$ in the shared memory in one go, we will load a block size of data in the shared memory. Therefore, when calling the kernel, we will initialize the shared memory to a size of a block. *[APPENDIX 1.1]*

With the help of indexing mechanism provided by CUDA, we can assign a value of $\boldsymbol{X}$ to each thread of the GPU. *[APPENDIX 1.2]*

Now the fun begins.

In a loop, we load the shared memory with a block size data from $\boldsymbol{X}$. Every thread requires every value of X to compute their y_i. Using ***syncthreads***, we ensure that every thread can sees the same data in the shared memory and completes the partial computation in each cycle before overwriting the shared memory with the next block size data of $\boldsymbol{X}$. *[APPENDIX 1.3]*

In this way, we focus on utilizing shared memory and optimize the memory transfers between the CPU and the GPU.

Performance Analysis

System Specification

Hostname	cpn-v09-34.core.ccr.buffalo.edu
Number of Cores	56
CPU	Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz
Operating System	Ubuntu 24.04.1 LTS
Compiler	g++ (Gentoo 10.4.0 p5) 10.4.0
GPU Model	NVIDIA A100-PCIE-40GB
CUDA Version	11.8

Results

The following table accounts for the runtime in seconds between the CPU and GPU for various values of n and h.

I am running the same tests for 2 different memory allocations as well just for fun.

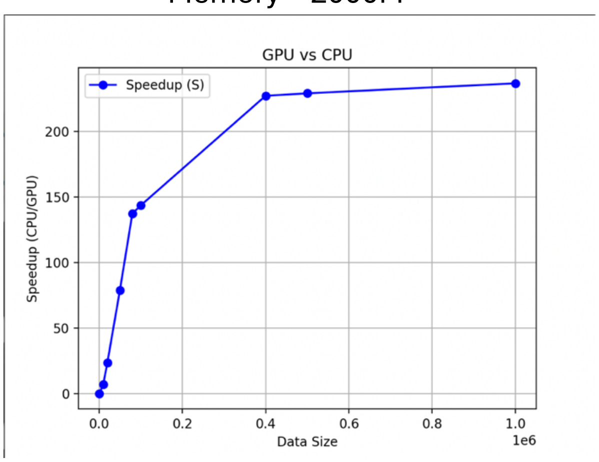
Memory = 2000M

n	GPU Runtime (s)	CPU Runtime (s)
500	0.463071	0.00305749
10000	0.154693	1.05875
20000	0.180783	4.26879
50000	0.335267	26.4008
80000	0.495377	68.0385
100000	0.740261	106.297
400000	7.48119	1700.22
500000	11.5838	2654.66
1000000	44.8964	10630.1

Memory = 32G

n	GPU Runtime (s)	CPU Runtime (s)
500	0.605956	0.00297401
10000	0.3151	0.934438
20000	0.312447	3.74471
50000	0.419532	23.3988
80000	0.459478	60.2911
100000	0.483655	93.8725
400000	1.33194	1505.35
500000	1.79408	2352.77
1000000	5.43868	9404.65

Memory = 2000M



Memory = 32G

Observation

As expected, we see diminished performance in a GPU as well when we have limited memory. Otherwise, the speedup is phenomenal !

References

Appendix

1.1. Calling the kernel and initializing the shared memory

```
/*
    Shared memory space = blockSize * sizeof(float)
    Copying the entire array of size n to shared memory is insane
*/
gaussian_kde_kernel<<<numBlocks, blockSize, blockSize * sizeof(float)>>>(n, h, d_x, d_y);
```

1.2. Determining which value of x, a thread should handle.

```
int gidx = blockIdx.x * blockDim.x + threadIdx.x;
int lidx = threadIdx.x;
```

1.3. GPU magic

```
float sum = 0.0;

for (int i = 0; i < gridDim.x; i++) {
    // save blockwise data into the shared memory in each iteration
    if ((lidx + i * blockDim.x) < n )
        sdata[lidx] = x[(lidx + i * blockDim.x)];

    // sync threads before computation
    __syncthreads();

    /*
        To compute y[k], take x[k] from global memory.
        Calculate the cummulative sum of elements till the block reached in this iteration
    */
    if (gidx < n) {
        int j = 0;
        while ((j < blockDim.x) && ((j + i * blockDim.x) < n)) {
            float diff = (x[gidx] - sdata[j]) / h;
            sum += (expf(-0.5 * diff * diff) / (sqrtf(2.0 * M_PI)));
            j++;
        }

    }

    // Sync threads before overriding values of next block into shared memory.
    __syncthreads();
}

y[gidx] = sum / (n * h);
```