Assignment 3 - Gaussian Kernel in Nvidia CUDA
Sanyam Gupta (sanyamgu)
16/12/2021

Overview
Implementation of an NVIDIA CUDA program for fast computing of Gaussian kernel density estimate. Program has been created in c++ by leveraging the cuda library. The performance of the program has been analyzed on CCR's general-compute partition. Results have been aggregated for 5 runs, and stdout file for 1 run has been reported.

Hardware capabilities.

	V100 PCIe	V100 SXM2	V100S PCIe
GPU Architecture	NVIDIA Volta		
NVIDIA Tensor Cores	640		
NVIDIA CUDA® Cores	5,120		
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS	8.2 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS	16.4 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS	130 TFLOPS
GPU Memory	32 GB /16 GB HBM2		32 GB HBM2
Memory Bandwidth	900 GB/sec		1134 GB/sec
ECC	Yes		
Interconnect Bandwidth	32 GB/sec	300 GB/sec	32 GB/sec
System Interface	PCIe Gen3	NVIDIA NVLink™	PCIe Gen3
Form Factor	PCIe Full Height/Length	SXM2	PCIe Full Height/Length
Max Power Comsumption	250 W	300 W	250 W
Thermal Solution	Passive		
Compute APIs	CUDA, DirectCompute, OpenCL™, OpenACC®		

FIG 1 - GPU Hardware Specs (Source - NVIDIA V100S Datasheet)

The code has been deployed on a GPU capable machine with GRES gpu:tesla_v100-pcie-16gb:2. Due to lack of support of <helper_cuda.h> library, exact cuda capabilities could not be ascertained.

Algorithm

Since data volume is assumed to be large enough to not file in a single blocks memory. Each block is repeatedly assigned a cunk of data from device memory. After the necessary calculations have been completed to this chunk, the next chunk is loaded. The data volume is in multiple of block size, so all blocks have equal amount of data, even the last block.

```
__global__
void
gaussianKernel(int n, float h, float* d_x, float* d_y){

    // index to be operated upon
    int gidx = threadIdx.x + blockIdx.x * blockDim.x;

    if (gidx < n){
        // buffer to load data
        extern __shared__ float buffer[];

        // temporarily maintain calcuate result
        float resultIndex = 0.0;

        //load data
        float val = d_x[gidx];
        __syncthreads();

        //since data volume is larger than block memory so load data in chunks
        for (int i = 0; i < gridDim.x; i++){
            buffer[threadIdx.x] = d_x[threadIdx.x + i * blockDim.x];
            __syncthreads();

            //calucuate khat with loaded data
            for (int j = 0; j < blockDim.x; j++)
                resultIndex += exp(-1 * pow(((val - buffer[j]) / h), 2) / 2);
        }

        //function value
        resultIndex /= (pow((2*3.14), 0.5) * n * h);

        //store back result
        d_y[gidx] = resultIndex;

    }

}
```

Fig 2 - Code Sample

Ascertaining Performance

Performance stats have been gathered from the following Job IDs - 7276180, 7276202, 7276203, 7276204 and 7276205. Aggregated results are reported below.

Observed Runtimes

		Size (x = 1024000)		
		x	2x	3x
	7276180	37.3175	138.207	302.941
	7276202	37.3668	138.302	303
Job Id	7276203	37.3435	138.068	302.896
	7276204	37.3798	138.29	302.962
	7276205	37.3144	138.026	302.901

Fig 4 - Runtime by job IDs

Average running times

The depicted times have been averaged over 3 runs. Maximum and minimum times have been discarded.

Size (x = 1024000)	x	2x	3x
time (s)	37.343	138.188	302.935

Fig 4 - Average Runtimes

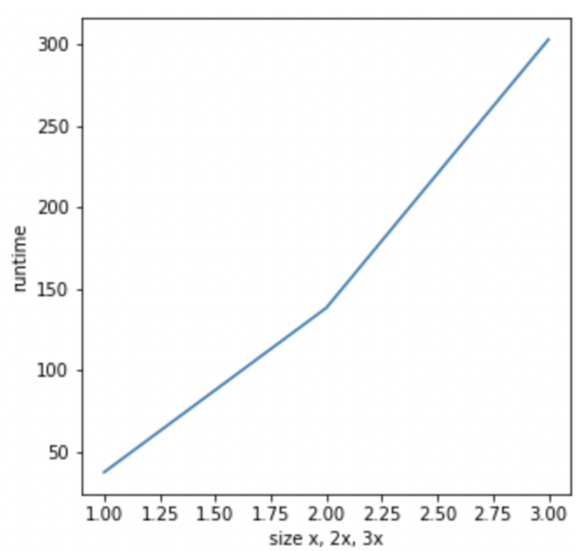

Fig 5 - Runtime plot

Performance Comment

The Linear (almost) increase in runtime is due to repeated loading of data from device memory to block memory. The algorithm implemented is not the most efficient. Runtime can be further decreased by opting for a more optimized data loading strategy.