CSE 570 Introduction to Parallel and Distributed Processing

Name : Ashwin Panditrao Jadhav

UB Name : ajadhav5

UB Number : 50405435

A3 – Gaussian Kernel in Nvidia Cuda

Algorithm

1. The algorithm first creates new input and output variables d_x and d_y which will be used in device kernel function. The variable d_x will store the random floating-point numbers and d_y will store the Gaussian Kernel calculated values. These variables are allocated in device memory using cudaMalloc() function which have size in bytes.
2. The algorithm initializes 1024 threads and then calculates the number of blocks depending upon the input size n and the number of threads, numBlocks = (n + numThreads - 1) / numThreads.
3. Now, the algorithm allocates memory for the input/output variables and copies the input data which is received from host to device using cudaMemcpy with parameter as cudaMemcpyHostToDevice.
4. Once the copy is completed, the algorithm calls the Gaussian kernel function in parallel depending on the number of threads and blocks passed between "<<<…….. >>>".
5. The kernel first initializes a shared variable using __shared__ on the device which will be used for caching the data in kernel, so that it can be accessed quickly while performing gaussian calculations. But before the calculations can start the algorithm first calculates the index of input data using "threadIdx, blockIdx, and blockDim" which are thread id, block id and block dimension respectively.
6. Now, the algorithm performs Gaussian calculations using iterations for each index in device kernel function using cached values and the iterations will run up to n over each input value of d_x using the index values and then sum them up with all the calculated gaussian kernel values to the output variable d_y.
7. Once all the iterations are over and the control leaves from the kernel the output then is copied back to host in y array. The last step done by the algorithm is to deallocate/free the cuda allocated memory on device.

Asymptotic Time Complexity

1. As the calculations are done in parallel over large number of threads and cuda cores, the communication time for copying of data from host to device and vice versa is much higher than the actual processing time.
2. The time taken for processing is O(n) as the algorithm iterated over the input for n times but this processing is done in parallel for (n + numThreads- 1)/numThreads.
3. So, the asymptotic time complexity is O(n / numThreads*numBlock) + μ, where μ is the communication/copying overhead.

Speedup and Efficiency

Average Speedup :

The speed up is calculated by keeping the cuda thread count as 1 and block count as 1 and then by increasing the thread and block count as well as the size of data n from 100 Million, 1 Billion, 10 Billion to 100 Billion.

For each configuration, three tests are performed and then the average has been taken and then which can be seen from below table :

Nodes/Threads and GPU	100 Million	1 Billion	10 Billion	100 Billion
1 Node / 64 Threads	88%	86%	87%	88%
1 Node / 128 Threads	87%	91%	89%	87%
1 Node / 256 Threads	89%	95%	88%	89%
1 Node / 512 Threads	88%	97%	88%	88%
1 Node / 1024 Threads / 1 V100 GPU	87%	92%	89%	87%
2 Node / 1024 Threads / 2 V100 GPU	92%	107%	100%	92%

Average Efficiency:

The efficiency is again calculated by keeping the cuda thread count as 1 and block count as 1 and then by increasing the thread and block count as well as the size of data n from 100 Million, 1 Billion, 10 Billion to 100 Billion, below is the average efficiency of the algorithm.

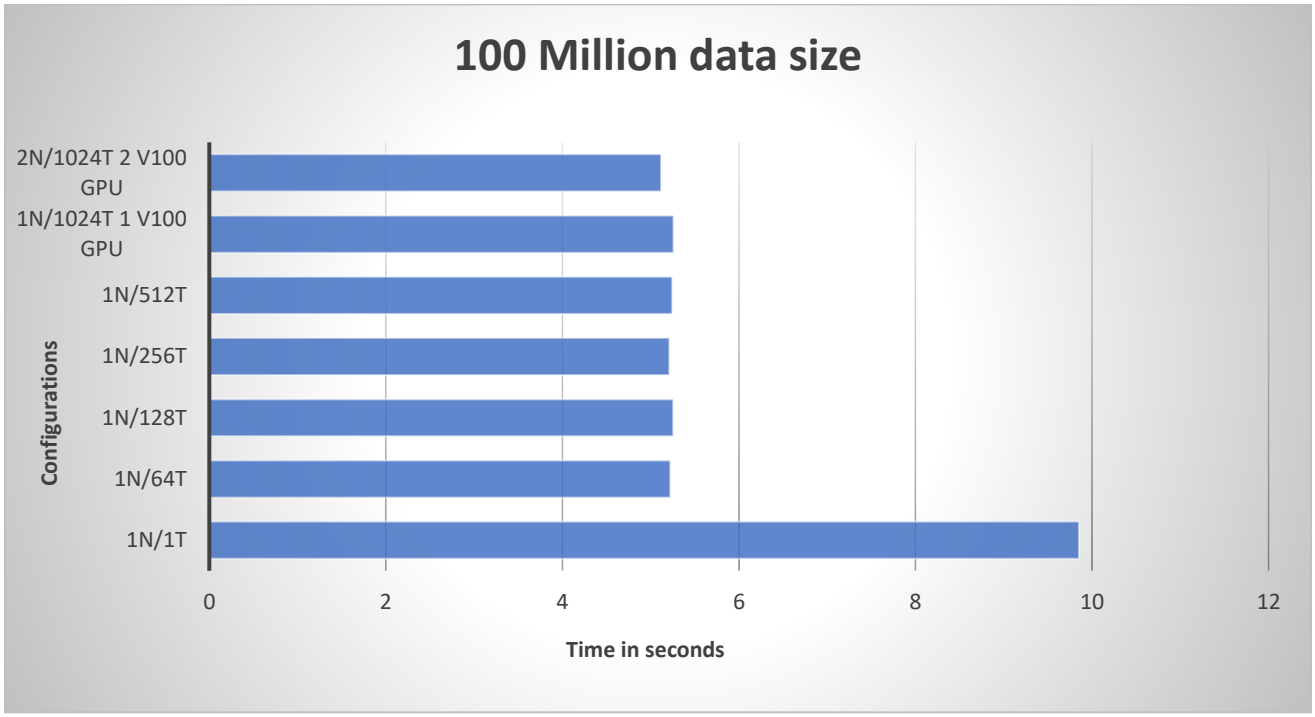
The average of all the experiments has been taken according to the data size and the average efficiency of the algorithm can be seen from the below table:

Nodes/Threads	100 Million	1 Billion	10 Billion	100 Billion
1 Node / 64 Threads	27.79%	29.09%	2.31%	2.48%
1 Node / 128 Threads	14.16%	14.88%	14.69%	14.65%
1 Node / 256 Threads	6.95%	7.61%	7.33%	7.38%
1 Node / 512 Threads	3.75%	3.84%	3.67%	3.67%
2 Node / 1024 Threads / 1 V100 GPU	1.76%	1.89%	1.84%	1.82%
2 Node / 1024 Threads / 2 V100 GPU	2%	2.03%	1.82%	1.89%
Average :	9.07%	9.89%	5.27%	5.31%

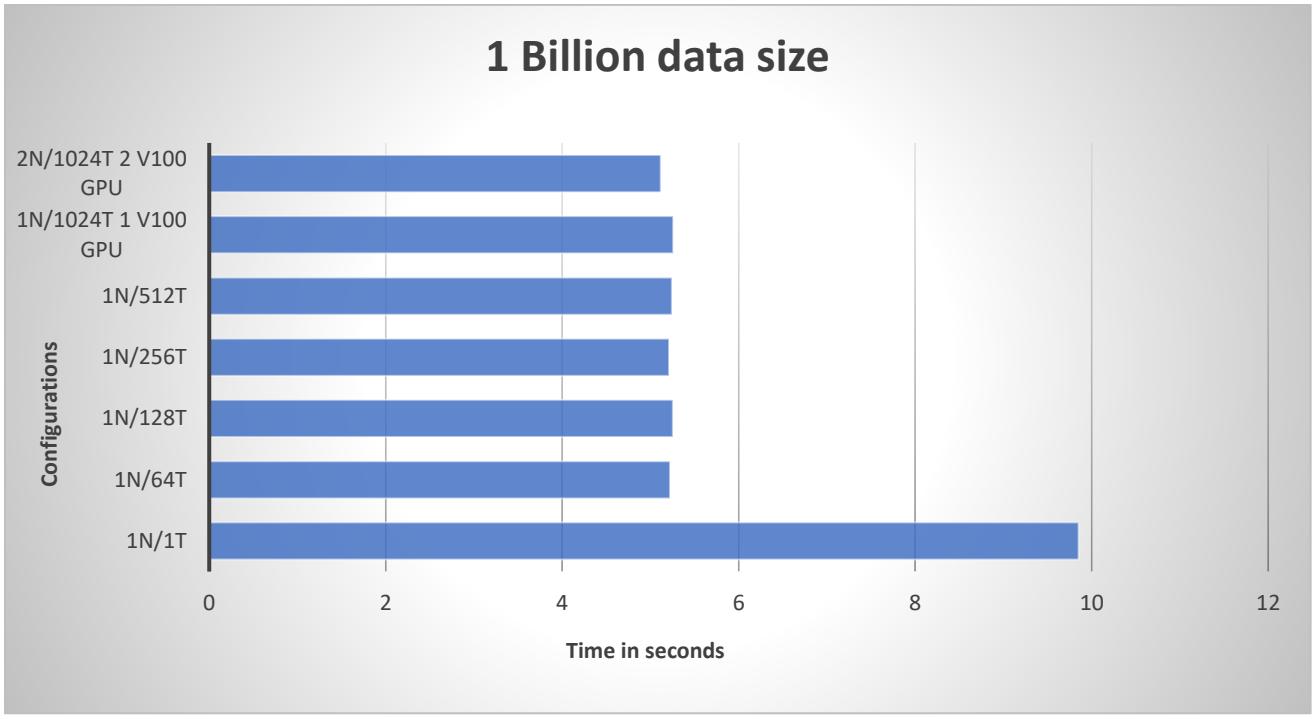
Speedup Graphs:

In graphs, N represents the number of Nodes and T represents the number of Threads.

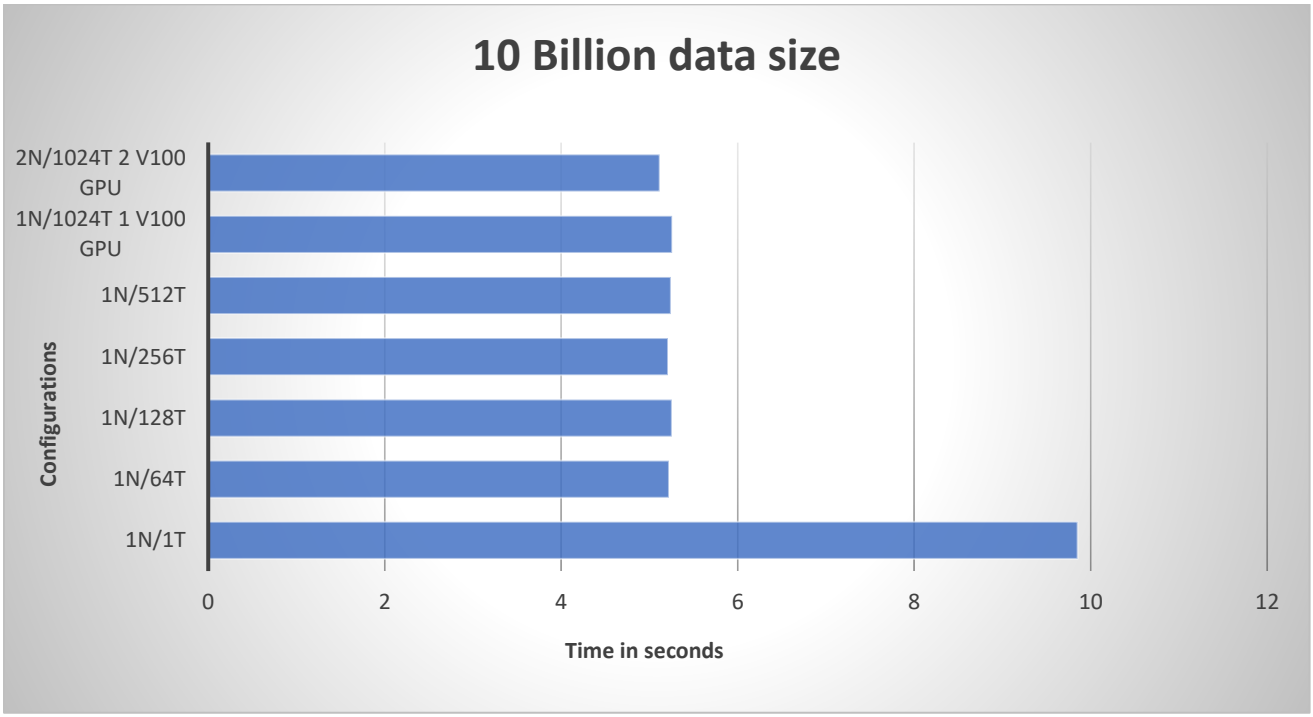
For 100 Million data size:



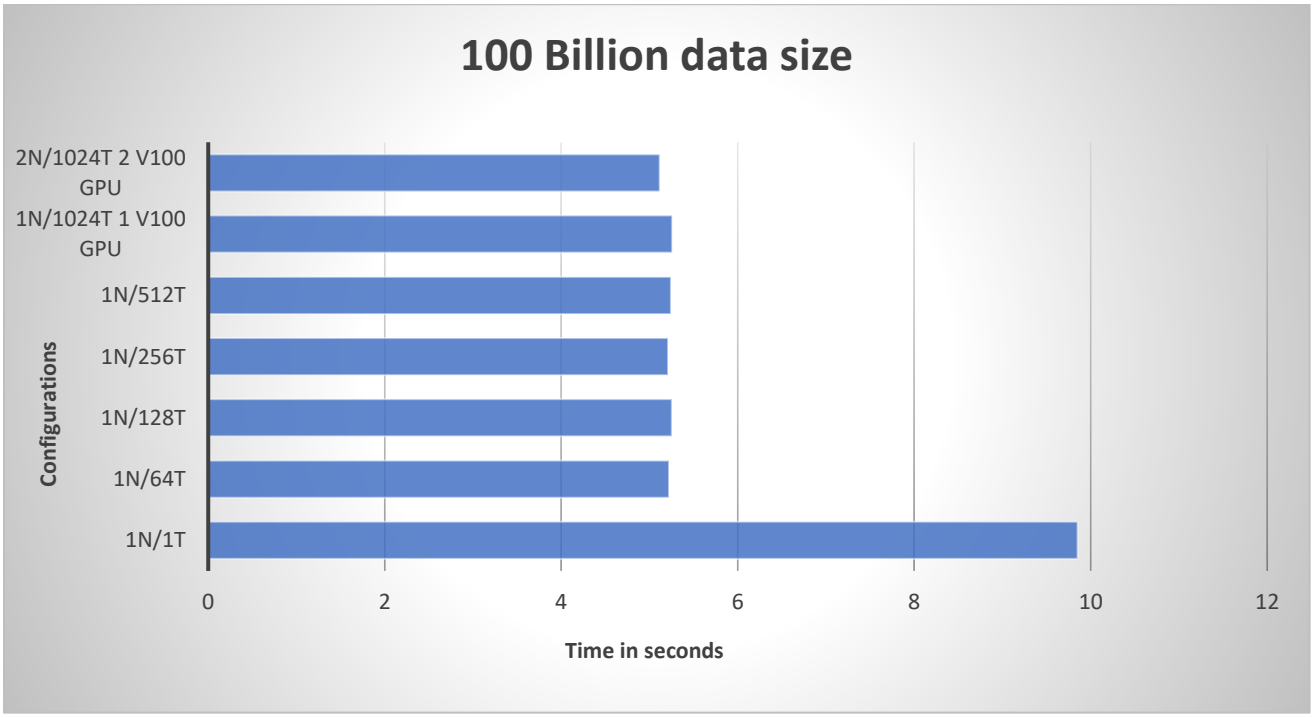
For 1 Billion data size:



For 10 Billion data size:



For 100 Billion data size:



Profiling for 100 Million data size using “ nvprof ” :

```
!nvprof ./a3 100000000 0.0001

==17490== NVPROF is profiling process 17490, command: ./a3 100000000 0.0001
==17490== Warning: Profiling results might be incorrect with current version of nvcc compiler used to compile cuda app. Compile with nvcc compiler 9.0 or later v
Tp: 0.983541s
==17490== Profiling application: ./a3 100000000 0.0001
==17490== Warning: 3 records have invalid timestamps due to insufficient device buffer space. You can configure the buffer space using the option --device-buffer
==17490== Warning: 2 records have invalid timestamps due to insufficient semaphore pool size. You can configure the pool size using the option --profiling-semaph
==17490== Profiling result:
No kernels were profiled.
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
     API calls:   72.88%  526.38ms         2  263.19ms  5.6690ms  520.71ms  cudaMemcpy
                  26.98%  194.87ms         2  97.433ms  181.45us  194.68ms  cudaMalloc
                   0.07%  483.72us         1  483.72us  483.72us  483.72us  cuDeviceTotalMem
                   0.04%  284.73us         1  284.73us  284.73us  284.73us  cudaLaunchKernel
                   0.03%  216.17us        96  2.2510us     128ns  109.01us  cuDeviceGetAttribute
                   0.00%  25.821us         1  25.821us  25.821us  25.821us  cuDeviceGetName
                   0.00%  8.2260us         2  4.1130us     785ns  7.4410us  cudaFree
                   0.00%  5.9160us         1  5.9160us  5.9160us  5.9160us  cuDeviceGetPCIBusId
                   0.00%  1.8460us         2     923ns     412ns  1.4340us  cuDeviceGet
                   0.00%  1.7380us         3     579ns     133ns     851ns  cuDeviceGetCount
```

Is the code always scalable?

No, the code is not always scalable. The scalability of the code is highly dependent on the size of data “n” and the number of threads being used.

Why not Scalable?

The biggest issue is the size of data. As the size of data increases the time taken for copying the data from host to device and vice versa also increases drastically (same can be seen in the profiling results for large data – 1 Billion). A better method of copying data can literally solve the scalability issue.

The code also suffers due to shared memory, a better implementation of shared memory may result in better scalability.

Also, an approach wherein the data can be split into some parts, maybe using multiple kernels and streams may prove to be a more scalable code.