**CS451**

**Assignment 4**

**Name : Sowmya Parameshwara**

**Email Id :** [**sparameshwara@hawk.iit.edu**](mailto:sparameshwara@hawk.iit.edu)

1. **Standard score using CUDA.**

* Input is stored by transposing the matrix, so that the attributes of a column are stored in a single row. This will optimise the algorithm since all threads in a block will access nearby elements, while normalising.
* Each row is normalised at a time for calculating standard score, the calculated values are stored in output matrix by transposing.
* Number of threads in a block is set as 16 (This value is determined based on the performance metrics shown below).
* The number of blocks is decided dynamically based on matrix size "N" and number of threads.
* The contents of a row are divided among the blocks. In each block,Each thread populates one elements of the block into shared data. We then calculate partial sum without divergence, on the data stored in shared memory.
* Once all blocks compute partial sum, we launch a kernel function using a single block by passing the calculated values from the previous step.
* This will calculate the final sum and final squared sum. To this final block we ensure the size of the partial sum array passed equals the next nearest power of 2 of "the number of blocks", as partial sum algorithm works only for powers of 2.
* The above data is used to calculate standard deviation for that row using the formula ((totalSquareSum + N\*powf(mean, 2.0) - 2 \* mean \* totalSum)/(float)N)
* The above value is used to calculate standard score for every element in that row.
* The above steps repeat for every row, calculating the standard score for all elements in the row.

**Key factors for improving performance:**

* Transposing the matrix and storing the column attributes to be normalised in a row, ensured the threads in a block are accessing nearby elements.
* In each block using shared memory area for calculating partial sum without divergence ensured less trips to global memory area.
* Allocating cuda memory once and reusing the same memory for normalising all the rows iteratively.

**Other approaches tried :**

* Before launching cuda kernel function for every row, a dimension object was created for storing the gridSize and blockSize. However this degraded performance.
* After each block calculated its partial sum, I created an array of length equal to nearest power of 2 of the number of blocks setting all elements of the array to zero. And than copying the results from the block to this array. However doing this step for each row degraded performance. Hence this memory area was created once, and reused for every row.
* Implemented the algorithm initially without transposing the matrix, this caused threads in a block to access non nearby elements degrading performance. Hence changed the implementation to do calculations on transposed matrix.

**Current algorithm performance result :**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | Sequentia | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
| 1000 | 17.085 | 339.918 | 338.918 | 369.911 | 349.909 | 374.903 | 886.77 | 348.914 | 334.918 |
| 2000 | 85.807 | 381.907 | 372.909 | 381.908 | 370.904 | 379.901 | 377.901 | 348.914 | 377.907 |
| 3000 | 180.302 | 347.916 | 441.892 | 432.895 | 429.888 | 424.89 | 432.887 | 584.857 | 440.892 |
| 4000 | 620.152 | 375.909 | 512.876 | 672.837 | 506.868 | 511.867 | 503.869 | 505.876 | 522.872 |
| 5000 | 607.91 | 421.897 | 420.898 | 606.853 | 585.848 | 596.845 | 785.796 | 586.856 | 598.853 |
| 6000 | 911.65 | 475.884 | 469.886 | 703.83 | 706.816 | 694.819 | 696.819 | 846.792 | 695.83 |
| 7000 | 1384.56 | 537.869 | 536.87 | 830.799 | 1045.73 | 808.789 | 810.79 | 807.802 | 817.799 |
| 8000 | 1835.84 | 608.852 | 603.853 | 958.769 | 952.752 | 933.757 | 934.757 | 948.767 | 1085.73 |
| 9000 | 1839.35 | 1197.71 | 686.834 | 687.834 | 1091.72 | 1081.72 | 1079.72 | 1084.72 | 1091.73 |
| 10000 | 1857.19 | 1385.66 | 778.811 | 778.811 | 1254.67 | 1242.68 | 1235.68 | 1253.67 | 1258.69 |
| 11000 | 1972.35 | 1593.61 | 1029.75 | 878.787 | 1415.63 | 1407.63 | 1407.63 | 1589.58 | 1426.65 |
| 12000 | 2530.38 | 1813.56 | 980.763 | 988.761 | 1624.58 | 1746.55 | 1742.55 | 1597.58 | 1620.6 |
| 13000 | 2966.55 | 2055.5 | 1112.73 | 1307.68 | 1831.52 | 1804.53 | 1795.53 | 2021.47 | 1793.56 |
| 14000 | 3410.31 | 2322.44 | 1247.7 | 1236.7 | 2259.41 | 2008.48 | 2008.48 | 2006.47 | 2141.47 |
| 15000 | 4344.17 | 2588.37 | 1381.67 | 1380.67 | 2272.41 | 2239.42 | 2233.42 | 2219.42 | 2238.45 |

**Analysis of the above result :**

* CUDA algorithm works better than sequential algorithm when matrix size equals 4000 and above.
* The percentage of performance improvement gained gets better as the size of the matrix increases.
* The performance is best when number of threads in a block equals 16.
* After 32 threads in a block, the performance degrades as the number of threads in a block increases for N>=9000.