Parallelizing Class Imbalance Problem Using SMOTE

E4750_2021Fall_CLIP_report

Angad Nandwani(an3077), Avik Dhupar(ad3910)
Columbia University

Abstract

In classification problems, a dataset may have multiple classes but their sizes could be in different proportions. A model trained on such an imbalanced dataset will only be able to learn the majority classes well, but will perform poor classification of the minority classes. To solve this problem, we use a pre-existing technique called Synthetic Minority Oversampling Technique(SMOTE). SMOTE uses k-nearest neighbors and vector distance calculation to generate synthetic data. In this project, we demonstrate a way to parallelize the SMOTE technique using GPUs in order to generate synthetic data to balance the dataset.

Key Words*: synthetic minority oversampling technique, CUDA, machine learning, parallel processing, GPU*

1. Overview

1.1 Problem in a Nutshell

The class imbalance problem in supervised machine learning leads to poor performance of the model, especially when deployed in the real world. For instance, consider the case of credit card transactions, wherein the motive of the model will be to predict fraudulent transactions accurately. However, in the real world, the major portion of the credit card transactions are not fraudulent, and hence any dataset won't have enough points for this class to get trained properly. Any model training algorithm won't have sufficient points to learn attributes of the minority class, which leads to wrong results. The model gets biased to the class whose points are in excess, thereby not classifying the fraudulent transactions properly. To resolve this problem, a common method used is Synthetic Minority Oversampling Technique (SMOTE). SMOTE operates on the minority class, and generates synthetic data which can be used to train unbiased machine learning models, thereby predicting classes in the dataset accurately.

Another instance where SMOTE is rapidly used is when the samples in one of the classes have many features with their values missing. In this scenario, the dataset might have the same number of samples, but since many of its important features had values missing, the training will not create a major impact and a biased model will be built.

SMOTE is highly used in the machine learning pipeline with datasets having imbalance classes, but it is often found to be slow to execute. The algorithm for SMOTE involves computation of distance between each point to all the other points in the dataset, which causes the code complexity of O(N^2). This step of forming an identity matrix as part of the SMOTE algorithm is the major bottleneck in the algorithm, which we have parallelized in the project using Graphics Processing Units (GPU).

SMOTE algorithm internally includes two functionality, vector distance, where it computes the euclidean distance of each point with all the other points in the dataset, and k-nearest neighbor, where it computes the nearest neighbor to any selected point to synthesize new points. In this project, we have parallelized both these functionality on GPU and observe major improvements in the elapsed time for the function.

1.2 Prior Work

Gutiérrez et Al [4] first demonstrate the use of commodity hardware for running SMOTE. They use various systems such as a server with NVIDIA Tesla k20 GPU with 5 GB of memory and 2496 CUDA cores, an Intel Xeon E5-2630 processor at 2.30 GHz and 128 GB of main memory. The next system they use is a desktop computer that uses an NVIDIA GeForce GTX 680 with 2 GB of memory and 1532 CUDA cores, an Intel core i7 3820 processor at 3.60 GHz and 24 GB of main memory. Lastly, they use a laptop computer that uses an NVIDIA GeForce GTX 740 m with 384 CUDA cores, an Intel Core i5 3337U processor at 1.8 GHz and 8 GB of main memory.

Chawla et al [5] demonstrate an improved SMOTE technique wherein, they use both oversampling of minority samples and undersampling of majority samples to produce the new dataset, which they claim provides better classifier performance than using oversampling only.

Ferńandez et al [6] reflect upon the implementation and applications of SMOTE, and go on to present the next set of challenges to extend SMOTE to Big Data problems, such as the curse of dimensionality, noise reduction, and real time processing.

2. Description

2.1. Goal and Objectives

The goal as stated is to generate new synthetic points belonging to minority class on both CPU and GPU using SMOTE technique, and later compare the difference in elapsed time for both parallel and serial version of the implementation. The three different kernels implemented on GPU and which were used for time comparison are

1) Naive GPU implementation of vector distance and k-NN
2) GPU implementation of vector distance and k-NN, with element sorting happening on CPU for k-NN.
3) GPU implementation of vector distance with shared Memory implementation of k-NN.

2.2. Problem Formulation and Design

Below is the step wise explanation for the SMOTE algorithm [6] which we have parallelized in this project.

➢ Assuming the minority class in the dataset A, for all x ∈ A, the k-nearest neighbors of x are obtained by calculating the euclidean distance between x and every other sample in set A.

➢ To obtain N new points, let's assume an A1 set having N (x_1,x_2…x_n) points which are selected through the nearest neighbor of x, where x ∈ A.

➢ Now, for all k=1,2,3..n, we will generate new points which will be given by, x` = x + rand(0,1) * |x-xk|, and x' ∈ minority class.

Above is the general technique used for the SMOTE algorithm, however to make sure that our results are comparable between CPU and GPU, we have tweaked the algorithm a little.

After selecting the k-nearest neighbors for x, we took the average of the features to generate points, rather than multiplying them with the random number. This definitely reduced the accuracy of the point belonging to minority class to some extent, however this was an important step to make sure that the elapsed time was comparable between CPU and GPU in our experiment. Our goal for this research was not to generate points accurately, which can be used while training in the machine learning algorithm, but was to compare time execution(elapsed time) for the SMOTE algorithm between GPU and CPU. It is important to note, that this is a very minute change, which once verifies the results, can be switched easily to multiple with any random point generated as part of the algorithm. However, then we will not be able to compare results generated by the parallel algorithm.

The core algorithms that power SMOTE are k-Nearest Neighbor & vector distance. Let's assume a minority class in the dataset as demonstrated in Fig. 1, with 5 rows and 5 features belonging to class "yes". The first step here is to impute the vector distance between each point, and for the same we will be invoking n*n threads on the GPU to parallely impute the vector distance D_{ij} where D is the distance between i & j points (Fig. 1, b: imputing vector distance).

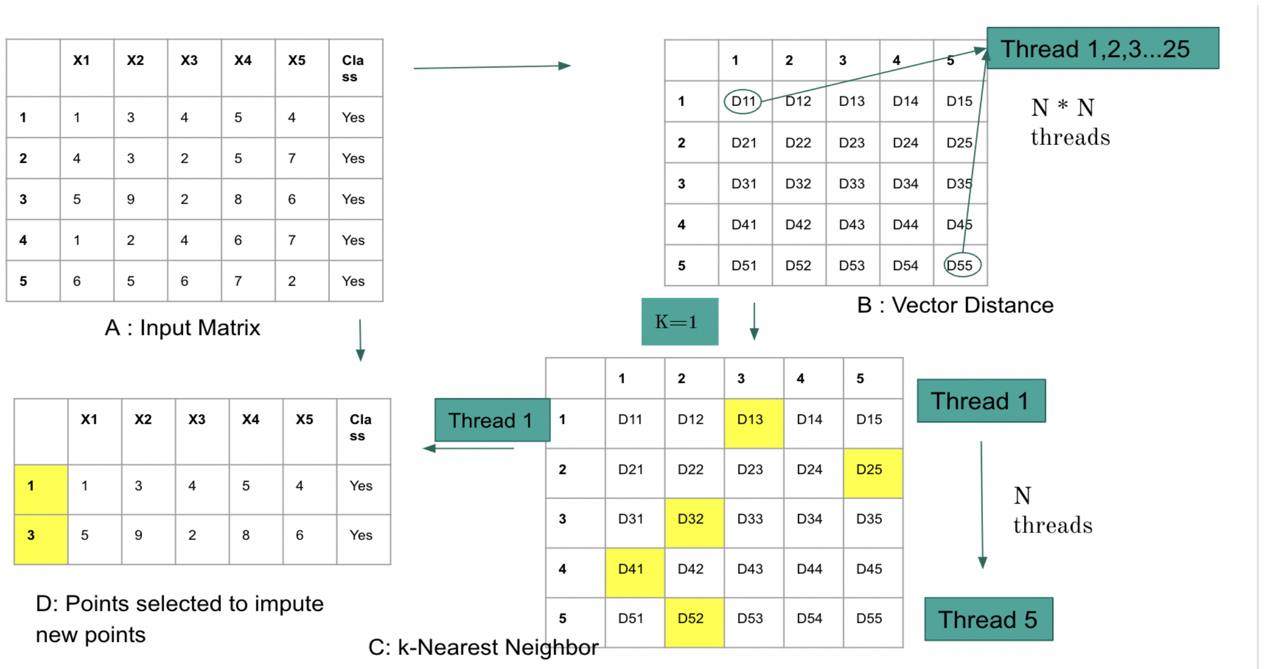


Fig 1: Invocation of threads for Vector Distance & K-Nearest Neighbors

Then, based on the value of k, we will be imputing k-nearest points. In the example, k value is 1, so for point 1, 3rd point (marked in yellow) is the nearest. Similarly for 2nd it is fifth, 3rd it is second, for 4th it is first and 5th it is second. Please note, these points were assumed to be nearest and not calculated (Fig 1, c: finding k-nearest points).. For this step, we will be invoking N threads for n rows in parallel, and each thread will be responsible to independently find k-nearest points for each sample in the vector distance output.

Once we have the k-nearest neighbors, we will simply impute the average of each feature to synthesize the new point for the minority class (Fig 1, d: synthesizing new points).

2.3 System and Software Design

Given below are the low level design representing the implementation of both vector distance and k-nearest neighbors.

2.3.1 Vector Distance : Invocation of Kernel

- ➢ If rowsCount * rowsCount <=1024 : then invoke threads in both X & Y with blockSize_X = blockSize_Y = rowsCount and GridSize_X = Grid_Size_Y = 1
- ➢ Else: Invoke threads in both X & Y with blockSize_X = blockSize_Y=32 and GridSize_X = Grid_Size_Y= math.ceil(rowsCount/32)

2.3.2 Vector Distance : Algorithm

For each thread with unique idx & idy:

- ➢ If(idx & idy <=rowsCount && idx!=idy): we need to impute the distance between row indices idx, idy i.e. D12 will represent distance between point 1 & 2, and then store it in result matrix at position "idx * rowsCount+idx"
- ➢ else if(idx==idy) : Store 0.0 to the result at the position - "idx * rowsCount+idx"

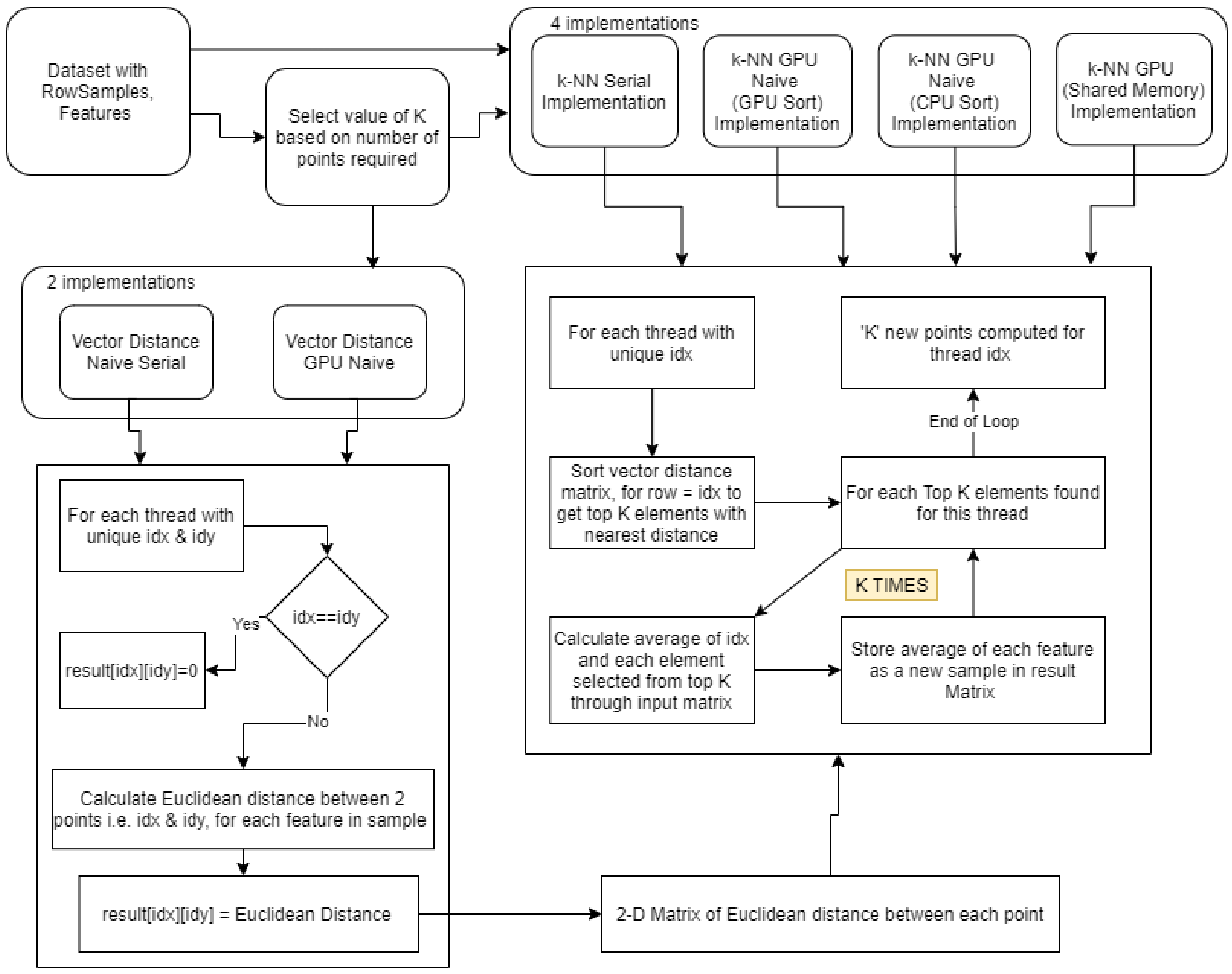

Dataset with RowSamples, Features
Select value of K based on number of points required
4 implementations
k-NN Serial Implementation
k-NN GPU Naive (GPU Sort) Implementation
k-NN GPU Naive (CPU Sort) Implementation
k-NN GPU (Shared Memory) Implementation
2 implementations
Vector Distance Naive Serial
Vector Distance GPU Naive
For each thread with unique idx & idy
idx==idy
Yes
No
result[idx][idy]=0
Calculate Euclidean distance between 2 points i.e. idx & idy, for each feature in sample
result[idx][idy] = Euclidean Distance
2-D Matrix of Euclidean distance between each point
For each thread with unique idx
Sort vector distance matrix, for row = idx to get top K elements with nearest distance
For each Top K elements found for this thread
End of Loop
'K' new points computed for thread idx
K TIMES
Calculate average of idx and each element selected from top K through input matrix
Store average of each feature as a new sample in result Matrix

Fig 2: Flow Diagram for algorithms: Vector Distance & K-Nearest Neighbor

2.3.3 K-Nearest Neighbor: Invocation of Kernel

- ➢ If rowsCount <=1024 : invoke threads in X with blockSize_X =rowsCount and GridSize_X = 1
- ➢ Else: invoke threads in X with blockSize_X = 1024 and GridSize_X= math.ceil(rowsCount/1024)

2.3.4 K-Nearest Neighbor: Algorithm

For each thread idx, if it is less than rowsCount:

- ➢ First step will be to impute k-nearest neighbor for this sample
- ➢ For this, we have first sorted the entire row of features for this sample, and picked "first K" from the result. We have tried sorting in both CPU and GPU and hence we have 2 variants for the Naive Version of Knn
- ➢ For each point in the top K selected Points
  - ○ Calculate the average of each feature between the samples selected i.e. idx and Kth Point.
  - ○ Store the average obtained for each feature as the new Sample in the result matrix.
- ➢ On Completion, K new points will be generated for a given idx and stored in the result matrix.

3. Results

As observed through Fig 3, the GPU variant for both Vector Distance and K-Nearest Neighbor algorithm have shown better elapsed time when compared to CPU i.e. serial version of it. Fig 3, demonstrates experiment on the randomly generated dataset with the rows increasing from 64 to 8192, and having a fixed number of features and k value (8,3 respectively). The major improvements in elapsed time was observed in vector distance where the parallel algorithm on GPU resulted in approximately **300X** improvement for the 8192 rows in the dataset. The K-nearest neighbors algorithm on the GPU resulted in **2X** faster elapsed time. Below are some of the interesting observations from the three different kernels of the k-nearest neighbor algorithm, which had less benefit in elapsed time when compared to VectorDistance.

3.1 K-Nearest Neighbor: CPU Sort Vs GPU Sort

For the k-nearest neighbor implementation, we have 3 kernels in the GPU. One, where the logic of sorting happens on the CPU, and GPU threads just select the top k for each row to impute new points. Another, where the logic of sorting and selection of k nearest points, both happen on the GPU itself. The idea behind implementing these 2 separately was the thought that the sorting on CPU for the elements should happen fast. However, after implementation, we realized that the CPU is responsible for sorting the entire matrix of vector distance which is then divided as per threads invoked, however in GPU sorting, each thread is sorting

Number of Rows	Vector Distance CPU Naive (ms)	Vector Distance GPU Naive (ms)	k-NN CPU Naive (ms)	k-NN GPU Naive (ms)	k-NN GPU (CPU Sort) (ms)	Knn Gpu Shared (ms)
64	28.07	2.086	8.192	3.922	3.567	1.025
256	441.469	2.915	68.781	24.115	48.985	24.047
1024	7306.5	31.242	964.638	465.625	859.034	371.749
2048	28304.937	124.387	3850.55	1926.440	3532.538	1911.049
4096	113108.70	461.8450	15462.044	8273.1103	14616.88	8273.375
8192	463131.68	1413.88	63622.26	39773.44	61138.83	39557.10

Fig 3: Elapsed Time in milliseconds for varying row sizes in input matrix with 8 features, and K=3

each row in the vector distance matrix, which is resulting in substantial parallelization. Since this is an interesting observation which we realized as part of the comparison and implementation, we have included it in the project report.

3.2 K-Nearest Neighbor: Shared Memory

For the shared memory implementation of the K-NN, we had to explicitly add a limitation of supporting only 128 rows upto 8 max features. This limitation occurred to restrict the shared memory size upto 1024 floating points array. The idea is that for every block i.e. 1024 elements, we will be storing the maximum used 128 points from the k-nearest neighbors matrix to the shared memory, which will be frequently accessed to impute the new points using the features for these points. However, the current implementation only includes the part where the first 128 points from each block were stored in the shared memory. These points were later accessed from the shared memory to impute features of the new point. Due to this current limitation of only being able to store 128 points from the block into shared memory, lead to very little improvement in the performance of k-NN gpu shared method. The logic for finding the top 128 points which had the maximum hits from the k-nearest neighbor required more time, and thus was way beyond the scope of this project.

It is important to note that the results comparison between the serial implementation and parallel implementation for both vector distance and knn have matched. Below is the graph (Fig 4) on log scale for the better representation and time comparison between the serial and parallel versions of the implementation. As highlighted in the matrix above, vector distance GPU implementation had the highest benefit in elapsed time, which was the initial bottleneck for the SMOTE implementation

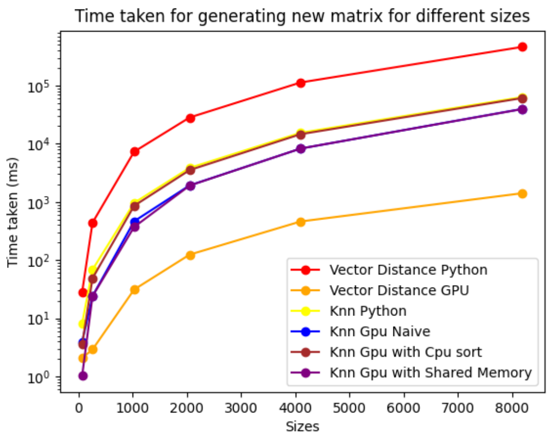
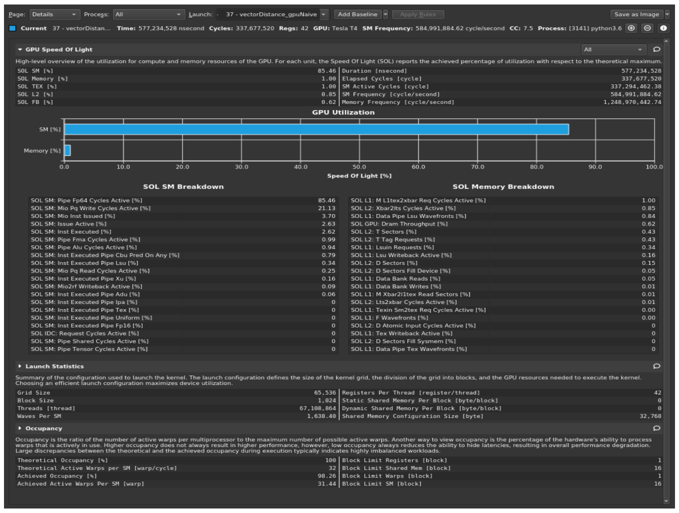


Fig 4: Time Comparison (Log Scale)

We also were able to perform the profiling for both k-nn and vector distance function. Fig 5, demonstrates the high usage of GPU (approx 85%) for the vector distance algorithm, which explains the efficiency of results. Since, we didn't see major benefit in elapsed time for knn, the gpu utilization was less, and hence we have included those diagrams in the appendices section of the report.

4. Discussion and Further Work

There are various things which we can pursue further in this project. One, is the shared memory implementation of vector distance algorithm. Although the improvement noticed was 300X, we can still improve this time by storing the initial matrix in shared memory. There will be few constraints which will have to be overcome while storing the matrix, however, given enough time and thought, we are confident that the results of shared memory implementation will be interesting to note. Also, as stated in the k-nearest neighbor shared memory implementation, the logic to find the maximum hit per block and then store those elements in the shared memory, can be also pursued as further optimization in the project. Once we achieve the best implementation for both k-NN and vector distance, which are also comparable to the serial implementation, then we can put the combination of these two codes in the machine learning pipeline to observe better accuracies on the test dataset, as the model will now have enough points to learn and will not be biased.



```
Current  37 - vectorDistan...  Time: 577,234,528 nsecond  Cycles: 337,677,520  Regs: 42  GPU: Tesla T4  SM Frequency: 584,991,884.62 cycle/second  CC: 7.5  Process: [3141] python3.6

GPU Speed Of Light
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.
SOL SM [%]          85.46 | Duration [nsecond]                  577,234,528
SOL Memory [%]       1.00 | Elapsed Cycles [cycle]              337,677,520
SOL TEX [%]          1.00 | SM Active Cycles [cycle]         337,294,462.38
SOL L2 [%]           0.85 | SM Frequency [cycle/second]      584,991,884.62
SOL FB [%]           0.62 | Memory Frequency [cycle/second] 1,248,970,442.74

SOL SM Breakdown                                      SOL Memory Breakdown
SOL SM: Pipe Fp64 Cycles Active [%]         85.46 | SOL L1: M L1tex2xbar Req Cycles Active [%]  1.00
SOL SM: Mio Pq Write Cycles Active [%]      21.13 | SOL L2: Xbar2lts Cycles Active [%]          0.85
SOL SM: Mio Inst Issued [%]                  3.70 | SOL L1: Data Pipe Lsu Wavefronts [%]        0.84
SOL SM: Issue Active [%]                     2.63 | SOL GPU: Dram Throughput [%]                0.62
SOL SM: Inst Executed [%]                    2.62 | SOL L2: T Sectors [%]                       0.43
SOL SM: Pipe Fma Cycles Active [%]           0.99 | SOL L2: T Tag Requests [%]                  0.43
SOL SM: Pipe Alu Cycles Active [%]           0.94 | SOL L1: Lsuin Requests [%]                  0.34
SOL SM: Inst Executed Pipe Cbu Pred On Any [%] 0.79 | SOL L1: Lsu Writeback Active [%]          0.16
SOL SM: Inst Executed Pipe Lsu [%]           0.34 | SOL L2: D Sectors [%]                       0.15
SOL SM: Mio Pq Read Cycles Active [%]        0.25 | SOL L2: D Sectors Fill Device [%]           0.05
SOL SM: Inst Executed Pipe Xu [%]            0.16 | SOL L1: Data Bank Reads [%]                 0.05
SOL SM: Mio2rf Writeback Active [%]          0.09 | SOL L1: Data Bank Writes [%]                0.01
SOL SM: Inst Executed Pipe Adu [%]           0.06 | SOL L1: M Xbar2l1tex Read Sectors [%]       0.01
SOL SM: Inst Executed Pipe Ipa [%]              0 | SOL L2: Lts2xbar Cycles Active [%]          0.01
SOL SM: Inst Executed Pipe Tex [%]              0 | SOL L1: Texin Sm2tex Req Cycles Active [%]  0.00
SOL SM: Inst Executed Pipe Uniform [%]          0 | SOL L1: F Wavefronts [%]                    0.00
SOL SM: Inst Executed Pipe Fp16 [%]             0 | SOL L2: D Atomic Input Cycles Active [%]       0
SOL IDC: Request Cycles Active [%]              0 | SOL L1: Tex Writeback Active [%]               0
SOL SM: Pipe Shared Cycles Active [%]           0 | SOL L2: D Sectors Fill Sysmem [%]              0
SOL SM: Pipe Tensor Cycles Active [%]           0 | SOL L1: Data Pipe Tex Wavefronts [%]           0

Launch Statistics
Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel.
Choosing an efficient launch configuration maximizes device utilization.
Grid Size                 65,536 | Registers Per Thread [register/thread]            42
Block Size                 1,024 | Static Shared Memory Per Block [byte/block]        0
Threads [thread]      67,108,864 | Dynamic Shared Memory Per Block [byte/block]       0
Waves Per SM            1,638.40 | Shared Memory Configuration Size [byte]       32,768

Occupancy
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process
warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation.
Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.
Theoretical Occupancy [%]                       100 | Block Limit Registers [block]     1
Theoretical Active Warps per SM [warp/cycle]     32 | Block Limit Shared Mem [block]   16
Achieved Occupancy [%]                        98.26 | Block Limit Warps [block]         1
Achieved Active Warps Per SM [warp]           31.44 | Block Limit SM [block]           16
```

Fig 5: 85% Utilization of GPU with Naive Vector Distance

5. Conclusion

We set out this project to implement the SMOTE technique and achieve better results in elapsed time as compared to serial implementation. We were able to figure out the bottleneck in the algorithm i.e. vector distance algorithm and achieve significant improvement of 300X on the algorithm. In addition, k-NN was also implemented and observed 2X benefit on the elapsed time. In total, the synthetic points when generated on CPU are taking approximately **nine minutes** for the 8192 rows, however, our current best implementation of the algorithm took **less than a minute**. This led to 90% reduction in elapsed time when both these algorithms were parallelized.

Please note, these experiments were performed on Google Cloud Platform, using a machine with the following configuration:

- ➢ Machine type: n1-standard-4 (4 vCPUs, 15 GB memory)
- ➢ CPU platform: Intel Haswell
- ➢ GPUs: 1 x NVIDIA Tesla T4
- ➢ OS: ubuntu-1804-bionic-v20210825

6. Acknowledgements

We would like to express our thanks and gratitude to Professor Zoran Kostić, who inspired us and taught us the concepts of parallel computing on Nvidia's CUDA platform, and provided us with Google Cloud Platform credits to test our code with. We would also like to thank Arjun Ahuja for promptly addressing our technical problems, and the immense support that he provided as part of his role as the teaching assistant. Further, we would like to thank the Data Science Institute and the Department of Electrical Engineering at Columbia University in the City of New York for putting together this phenomenal course on heterogeneous computing. Last but certainly not the least, we would like to thank our families and friends who supported us through the rigorous semester, and kept our spirits high.

7. References

[1] CLIP Project Github Repository https://github.com/eecse4750/e4750-2021Fall-Project-CLIP-an3077-ad3910

[2] AI, C.X., 2005. GPU Gems 3

[3] Kirk, D. and Wen-Mei, W.H., 2016. Programming massively parallel processors: a hands-on approach. Morgan kaufmann.

[4] Gutiérrez, P.D., Lastra, M., Benítez, J.M. and Herrera, F., 2017. Smote-gpu: Big data preprocessing on commodity hardware for imbalanced classification. Progress in Artificial Intelligence, 6(4), pp.347-354.

[5] Chawla, N.V., Bowyer, K.W., Hall, L.O. and Kegelmeyer, W.P., 2002. SMOTE: synthetic minority over-sampling technique. Journal of artificial intelligence research, 16, pp.321-357.

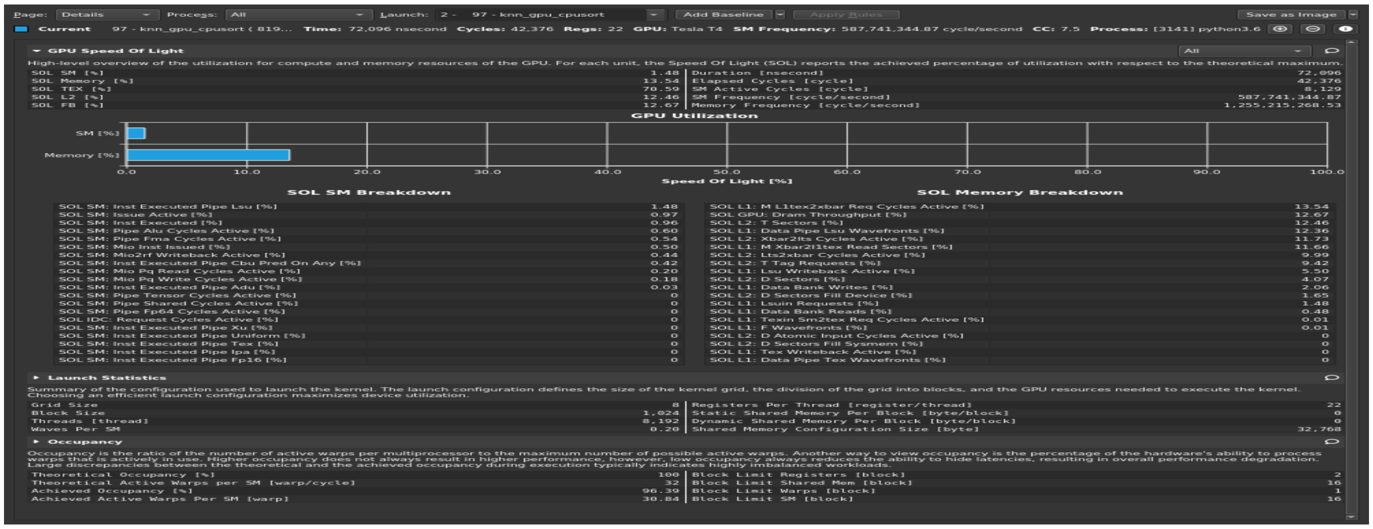
[6] Fernández, A., Garcia, S., Herrera, F. and Chawla, N.V., 2018. SMOTE for learning from imbalanced data: progress and challenges, marking the 15-year anniversary. Journal of artificial intelligence research, 61, pp.863-905.

[7] Maciejewski, T. and Stefanowski, J., 2011, April. Local neighbourhood extension of SMOTE for mining imbalanced data. In 2011 IEEE symposium on computational intelligence and data mining (CIDM) (pp. 104-111). IEEE.

8. Appendices

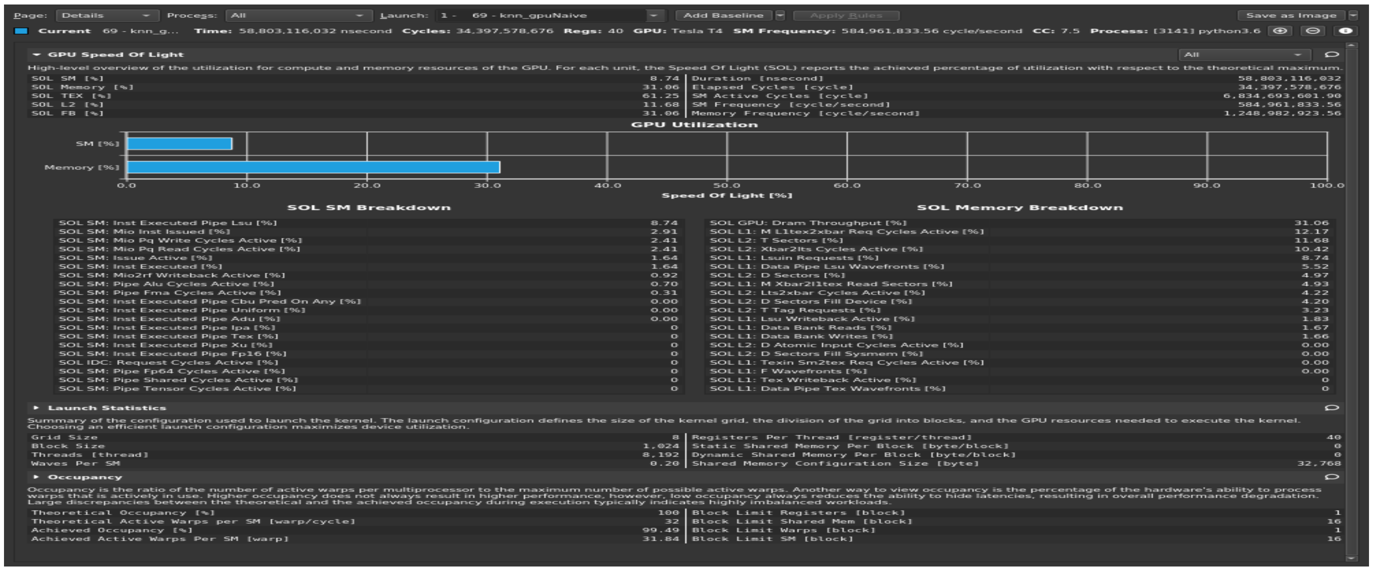
Individual Student Contributions (in %)

Task	% an3077	% ad3910
Overall	50%	50%
Vector Addition Serial	Pair Programming	Pair Programming
Vector Addition Parallel	Pair Programming	Pair Programming
k-NN Serial	Pair Programming	Pair Programming
K-NN Parallel	Pair Programming	Pair Programming
Report + PPT	50%	50%



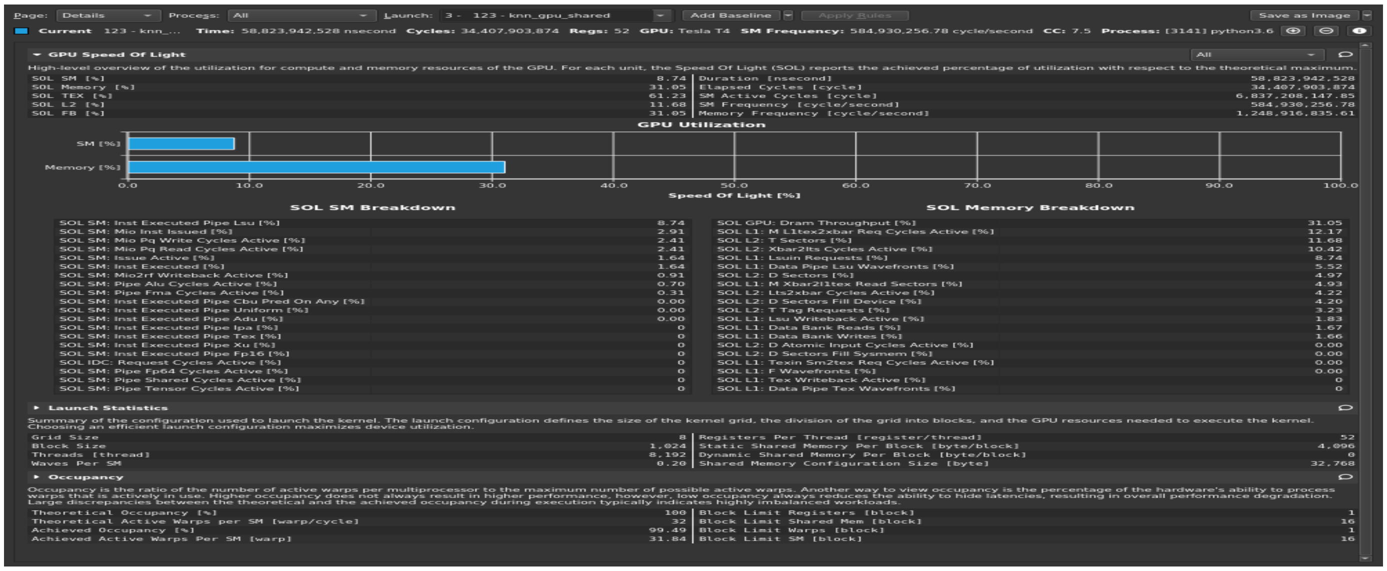
```
Page: Details   Process: All   Launch: 2 - 97 - knn_gpu_cpusort   Add Baseline   Apply Rules   Save as Image
Current  97 - knn_gpu_cpusort ( 819...  Time: 72,096 nsecond  Cycles: 42,376  Regs: 22  GPU: Tesla T4  SM Frequency: 587,741,344.87 cycle/second  CC: 7.5  Process: [3141] python3.6

▾ GPU Speed Of Light                                                                                     All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.
SOL SM [%]              1.48 | Duration [nsecond]                         72,096
SOL Memory [%]         13.54 | Elapsed Cycles [cycle]                     42,376
SOL TEX [%]            70.59 | SM Active Cycles [cycle]                    8,129
SOL L2 [%]             12.46 | SM Frequency [cycle/second]        587,741,344.87
SOL FB [%]             12.67 | Memory Frequency [cycle/second]  1,255,215,268.53

GPU Utilization
SM [%]
Memory [%]
0.0  10.0  20.0  30.0  40.0  50.0  60.0  70.0  80.0  90.0  100.0
Speed Of Light [%]

SOL SM Breakdown                                                  SOL Memory Breakdown
SOL SM: Inst Executed Pipe Lsu [%]                  1.48    SOL L1: M L1tex2xbar Req Cycles Active [%]     13.54
SOL SM: Issue Active [%]                            0.97    SOL GPU: Dram Throughput [%]                   12.67
SOL SM: Inst Executed [%]                           0.96    SOL L2: T Sectors [%]                          12.46
SOL SM: Pipe Alu Cycles Active [%]                  0.60    SOL L1: Data Pipe Lsu Wavefronts [%]           12.36
SOL SM: Pipe Fma Cycles Active [%]                  0.54    SOL L2: Xbar2lts Cycles Active [%]             11.73
SOL SM: Mio Inst Issued [%]                         0.50    SOL L1: M Xbar2l1tex Read Sectors [%]          11.66
SOL SM: Mio2rf Writeback Active [%]                 0.44    SOL L2: Lts2xbar Cycles Active [%]              9.99
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]      0.42    SOL L2: T Tag Requests [%]                      9.42
SOL SM: Mio Pq Read Cycles Active [%]               0.20    SOL L1: Lsu Writeback Active [%]                5.50
SOL SM: Mio Pq Write Cycles Active [%]              0.18    SOL L2: D Sectors [%]                           4.07
SOL SM: Inst Executed Pipe Adu [%]                  0.03    SOL L1: Data Bank Writes [%]                    2.06
SOL SM: Pipe Tensor Cycles Active [%]                  0    SOL L2: D Sectors Fill Device [%]               1.65
SOL SM: Pipe Shared Cycles Active [%]                  0    SOL L1: Lsuin Requests [%]                      1.48
SOL SM: Pipe Fp64 Cycles Active [%]                    0    SOL L1: Data Bank Reads [%]                     0.48
SOL IDC: Request Cycles Active [%]                     0    SOL L1: Texin Sm2tex Req Cycles Active [%]      0.01
SOL SM: Inst Executed Pipe Xu [%]                      0    SOL L1: F Wavefronts [%]                        0.01
SOL SM: Inst Executed Pipe Uniform [%]                 0    SOL L2: D Atomic Input Cycles Active [%]           0
SOL SM: Inst Executed Pipe Tex [%]                     0    SOL L2: D Sectors Fill Sysmem [%]                  0
SOL SM: Inst Executed Pipe Ipa [%]                     0    SOL L1: Tex Writeback Active [%]                   0
SOL SM: Inst Executed Pipe Fp16 [%]                    0    SOL L1: Data Pipe Tex Wavefronts [%]               0

▸ Launch Statistics
Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel.
Choosing an efficient launch configuration maximizes device utilization.
Grid Size                  8 | Registers Per Thread [register/thread]              22
Block Size             1,024 | Static Shared Memory Per Block [byte/block]          0
Threads [thread]       8,192 | Dynamic Shared Memory Per Block [byte/block]         0
Waves Per SM            0.20 | Shared Memory Configuration Size [byte]         32,768
▸ Occupancy
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process
warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation.
Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.
Theoretical Occupancy [%]                      100 | Block Limit Registers [block]      2
Theoretical Active Warps per SM [warp/cycle]    32 | Block Limit Shared Mem [block]    16
Achieved Occupancy [%]                       96.39 | Block Limit Warps [block]          1
Achieved Active Warps Per SM [warp]          30.84 | Block Limit SM [block]            16
```

Fig 6: Utilization of GPU with K-nn Gpu Cpu Sort



```
Page: Details   Process: All   Launch: 1 - 69 - knn_gpuNaive   Add Baseline   Apply Rules   Save as Image
Current  69 - knn_g...  Time: 58,803,116,032 nsecond  Cycles: 34,397,578,676  Regs: 40  GPU: Tesla T4  SM Frequency: 584,961,833.56 cycle/second  CC: 7.5  Process: [3141] python3.6

▾ GPU Speed Of Light                                                                                     All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.
SOL SM [%]              8.74 | Duration [nsecond]                 58,803,116,032
SOL Memory [%]         31.06 | Elapsed Cycles [cycle]             34,397,578,676
SOL TEX [%]            61.25 | SM Active Cycles [cycle]         6,834,693,601.90
SOL L2 [%]             11.68 | SM Frequency [cycle/second]        584,961,833.56
SOL FB [%]             31.06 | Memory Frequency [cycle/second]  1,248,982,923.56

GPU Utilization
SM [%]
Memory [%]
0.0  10.0  20.0  30.0  40.0  50.0  60.0  70.0  80.0  90.0  100.0
Speed Of Light [%]

SOL SM Breakdown                                                  SOL Memory Breakdown
SOL SM: Inst Executed Pipe Lsu [%]                  8.74    SOL GPU: Dram Throughput [%]                   31.06
SOL SM: Mio Inst Issued [%]                         2.91    SOL L1: M L1tex2xbar Req Cycles Active [%]     12.17
SOL SM: Mio Pq Write Cycles Active [%]              2.41    SOL L2: T Sectors [%]                          11.68
SOL SM: Mio Pq Read Cycles Active [%]               2.41    SOL L2: Xbar2lts Cycles Active [%]             10.42
SOL SM: Issue Active [%]                            1.64    SOL L1: Lsuin Requests [%]                      8.74
SOL SM: Inst Executed [%]                           1.64    SOL L1: Data Pipe Lsu Wavefronts [%]            5.52
SOL SM: Mio2rf Writeback Active [%]                 0.92    SOL L2: D Sectors [%]                           4.97
SOL SM: Pipe Alu Cycles Active [%]                  0.70    SOL L1: M Xbar2l1tex Read Sectors [%]           4.93
SOL SM: Pipe Fma Cycles Active [%]                  0.31    SOL L2: Lts2xbar Cycles Active [%]              4.22
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]      0.00    SOL L2: D Sectors Fill Device [%]               4.20
SOL SM: Inst Executed Pipe Uniform [%]              0.00    SOL L2: T Tag Requests [%]                      3.23
SOL SM: Inst Executed Pipe Adu [%]                  0.00    SOL L1: Lsu Writeback Active [%]                1.83
SOL SM: Inst Executed Pipe Ipa [%]                     0    SOL L1: Data Bank Reads [%]                     1.67
SOL SM: Inst Executed Pipe Tex [%]                     0    SOL L1: Data Bank Writes [%]                    1.66
SOL SM: Inst Executed Pipe Xu [%]                      0    SOL L2: D Atomic Input Cycles Active [%]        0.00
SOL SM: Inst Executed Pipe Fp16 [%]                    0    SOL L2: D Sectors Fill Sysmem [%]               0.00
SOL IDC: Request Cycles Active [%]                     0    SOL L1: Texin Sm2tex Req Cycles Active [%]      0.00
SOL SM: Pipe Fp64 Cycles Active [%]                    0    SOL L1: F Wavefronts [%]                        0.00
SOL SM: Pipe Shared Cycles Active [%]                  0    SOL L1: Tex Writeback Active [%]                   0
SOL SM: Pipe Tensor Cycles Active [%]                  0    SOL L1: Data Pipe Tex Wavefronts [%]               0

▸ Launch Statistics
Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel.
Choosing an efficient launch configuration maximizes device utilization.
Grid Size                  8 | Registers Per Thread [register/thread]              40
Block Size             1,024 | Static Shared Memory Per Block [byte/block]          0
Threads [thread]       8,192 | Dynamic Shared Memory Per Block [byte/block]         0
Waves Per SM            0.20 | Shared Memory Configuration Size [byte]         32,768
▸ Occupancy
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process
warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation.
Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.
Theoretical Occupancy [%]                      100 | Block Limit Registers [block]      1
Theoretical Active Warps per SM [warp/cycle]    32 | Block Limit Shared Mem [block]    16
Achieved Occupancy [%]                       99.49 | Block Limit Warps [block]          1
Achieved Active Warps Per SM [warp]          31.84 | Block Limit SM [block]            16
```

Fig 7: Utilization of GPU with K-nn Gpu Naive



```
Page: Details   Process: All   Launch: 3 - 123 - knn_gpu_shared   Add Baseline   Apply Rules   Save as Image
Current  123 - knn_...  Time: 58,823,942,528 nsecond  Cycles: 34,407,903,874  Regs: 52  GPU: Tesla T4  SM Frequency: 584,930,256.78 cycle/second  CC: 7.5  Process: [3141] python3.6

▾ GPU Speed Of Light                                                                                     All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.
SOL SM [%]              8.74 | Duration [nsecond]                 58,823,942,528
SOL Memory [%]         31.05 | Elapsed Cycles [cycle]             34,407,903,874
SOL TEX [%]            61.23 | SM Active Cycles [cycle]         6,837,208,147.85
SOL L2 [%]             11.68 | SM Frequency [cycle/second]        584,930,256.78
SOL FB [%]             31.05 | Memory Frequency [cycle/second]  1,248,916,835.61

GPU Utilization
SM [%]
Memory [%]
0.0  10.0  20.0  30.0  40.0  50.0  60.0  70.0  80.0  90.0  100.0
Speed Of Light [%]

SOL SM Breakdown                                                  SOL Memory Breakdown
SOL SM: Inst Executed Pipe Lsu [%]                  8.74    SOL GPU: Dram Throughput [%]                   31.05
SOL SM: Mio Inst Issued [%]                         2.91    SOL L1: M L1tex2xbar Req Cycles Active [%]     12.17
SOL SM: Mio Pq Write Cycles Active [%]              2.41    SOL L2: T Sectors [%]                          11.68
SOL SM: Mio Pq Read Cycles Active [%]               2.41    SOL L2: Xbar2lts Cycles Active [%]             10.42
SOL SM: Issue Active [%]                            1.64    SOL L1: Lsuin Requests [%]                      8.74
SOL SM: Inst Executed [%]                           1.64    SOL L1: Data Pipe Lsu Wavefronts [%]            5.52
SOL SM: Mio2rf Writeback Active [%]                 0.91    SOL L2: D Sectors [%]                           4.97
SOL SM: Pipe Alu Cycles Active [%]                  0.70    SOL L1: M Xbar2l1tex Read Sectors [%]           4.93
SOL SM: Pipe Fma Cycles Active [%]                  0.31    SOL L2: Lts2xbar Cycles Active [%]              4.22
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]      0.00    SOL L2: D Sectors Fill Device [%]               4.20
SOL SM: Inst Executed Pipe Uniform [%]              0.00    SOL L2: T Tag Requests [%]                      3.23
SOL SM: Inst Executed Pipe Adu [%]                  0.00    SOL L1: Lsu Writeback Active [%]                1.83
SOL SM: Inst Executed Pipe Ipa [%]                     0    SOL L1: Data Bank Reads [%]                     1.67
SOL SM: Inst Executed Pipe Tex [%]                     0    SOL L1: Data Bank Writes [%]                    1.66
SOL SM: Inst Executed Pipe Xu [%]                      0    SOL L2: D Atomic Input Cycles Active [%]        0.00
SOL SM: Inst Executed Pipe Fp16 [%]                    0    SOL L2: D Sectors Fill Sysmem [%]               0.00
SOL IDC: Request Cycles Active [%]                     0    SOL L1: Texin Sm2tex Req Cycles Active [%]      0.00
SOL SM: Pipe Fp64 Cycles Active [%]                    0    SOL L1: F Wavefronts [%]                        0.00
SOL SM: Pipe Shared Cycles Active [%]                  0    SOL L1: Tex Writeback Active [%]                   0
SOL SM: Pipe Tensor Cycles Active [%]                  0    SOL L1: Data Pipe Tex Wavefronts [%]               0

▸ Launch Statistics
Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel.
Choosing an efficient launch configuration maximizes device utilization.
Grid Size                  8 | Registers Per Thread [register/thread]              52
Block Size             1,024 | Static Shared Memory Per Block [byte/block]      4,096
Threads [thread]       8,192 | Dynamic Shared Memory Per Block [byte/block]         0
Waves Per SM            0.20 | Shared Memory Configuration Size [byte]         32,768
▸ Occupancy
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process
warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation.
Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.
Theoretical Occupancy [%]                      100 | Block Limit Registers [block]      1
Theoretical Active Warps per SM [warp/cycle]    32 | Block Limit Shared Mem [block]    16
Achieved Occupancy [%]                       99.49 | Block Limit Warps [block]          1
Achieved Active Warps Per SM [warp]          31.84 | Block Limit SM [block]            16
```

Fig 8: Utilization of GPU with K-nn Gpu Shared