Distributed kNN using vp-tree

Filosidis Stavros 9456, Gitopoulos Giorgos 9344

Parallel & Distributed Systems @ ECE AUTh

Abstract

K-Nearest-Neighbors is an algorithm used for various classification and segmentation purposes, especially in clustering and data grouping. When dealing with big data, especially datasets that do not entirely fit inside computer's random access memory, performance degradation is particularly visible when performing multiple queries.

Two alternative implementations are included in this report, the first being the naïve algorithm using **matrix multiplication** and the other is a **vantage point tree** approach, to optimize the required visits for the knn result. Note that both approaches implement all-knn and not the approximate version of the algorithm.

The repository for the code can be found here: Github Link

1 Naïve algorithm

The first algorithm is implemented using the Euclidean distance matrix in order to calculate each possible distance from a point to another in the Euclidean space.

1.1 Distance matrix calculation

To calculate an $m \times n$ Euclidean distance matrix D between two sets points X and Y of m and n points respectively, we use formula 1.

$$D = (X \odot X)\, e\, e^T - 2\, X\, Y^T + e\, e^T (Y \odot Y)^T \qquad (1)$$

For corpus points $x_1, x_2, \ldots, x_m$ and query points $y_1, y_2, \ldots, y_n$ (X and Y respectively) in k-dimensional space $\mathbb{R}^k$, the elements of their Euclidean space matrix A are given by squares of distances between them. That is:

$$a_{ij} = \sqrt{d_{ij}^2} = \sqrt{\|x_i - x_j\|^2} \qquad (2)$$

for each element d_{ij} in matrix D.

In order to calculate matrix A we utilize the **OpenBLAS** library, an optimized version of **BLAS**, and in particular the routine **cblas_dgemm** which multiplies two row major-formatted matrices (double precision).

A particular problem to this approach is the amount of system memory used by the routine, since it does not handle queries of $n \gtrapprox 20.000$ points very well, and also produces a significant amount of cache misses, as we saw in our analysis with the *perf* tool. To solve this problem, the elements of the query Y had to be processed in small chunks, one at a time, and to be placed into the final distance matrix. This not only increased performance significantly, but also allowed us to process orders of magnitude more points without memory problems. We found that a value of *1000 points / block* works particularly well for our implementation.

1.2 Distributed implementation

For the distributed version of the algorithm, we use **MPI** *(Message Passing Interface)* to create multiple independent *processes* and to achieve direct data communications between them.

Distributing the data to multiple processes efficiently is a complex problem and just sending all data chunks from one process is not the ideal solution. To solve this we used the custom *MPI_Scatterv* routine that implements all the peer to peer communications under the hood decreasing the initial distribution times.

Since X and Y are stored in a row-major format, we compute the *chunk size* and the corresponding *index displacements* for the initial query indices respectively in order to distribute each chunk in multiple threads and for each process to be aware of the correct initial point indices, which are needed for the final answer. We only send the corpus data and keep the queries to reduce the bandwidth and complexity needed to send all the queries, indices and k nearest points each time.

For example, consider matrix X with $m = 8$ and $d = 2$, (for the *2-dimensional* space), with points $(0, 0)$, $(1, 0)$, $(2, 0)$, $(1, 2)$, $(3, 1)$ stored in row major format as follows:

0	1	2	3	4	5	6	7	8	9
0	0	1	0	2	0	1	2	3	1

The displacement matrix and the chunk size matrix for 2 processes will be the following:

0	6

6	4

1.3 KNN

To efficiently perform KNN search, and since we have calculated all-to-all distances, we append the k closest previously selected neighbors into the new calculated ones and we perform a partial sort. Partial sort has a complexity of $\mathcal{O}(len(local\_corpus) \cdot \log(k))$, and we perform

this operation in every process with a total cost of $\mathcal{O}(p \cdot len(local\_corpus) \log(k))$.

2 Vantage Point Tree (vpt)

To improve the execution speed of the KNN queries, we will use a vantage point tree, a specialized data structure used for space partitioning problems. VP trees are more specifically metric trees, which are trees with the ability to efficiently partition data in n-dimensional metric space. The tree is constructed from the available corpus points and then we query new points against it.

2.1 Tree Structure & Construction

Each node contains a *vantage point*, μ, and a *left* and *right* child nodes. Additionally, tree leaves also contain a list of $\geq \beta$ points that belong to that vantage point, meaning that no more children are created if there are no available points in the node radius.

To construct the tree, we begin by randomly selecting a corpus point as the root node and then we compute μ. This value represents the radius at which half the points are inside the node radius, and the other half are out. For each segmented space created, we recursively select new vantage points and continue the same procedure. We select a vantage point that lays inside μ as the left child, and one that lays outside it as the right child. We continue until we cannot further partition space, because of the artificial bound β, so we assign remaining points to the leaves. For our tree, after experimentation we decided to use $\beta = 0.3 \log_2(\frac{m}{p})$ where p is the number of processes.

2.2 Search

We recursively search the tree by reaching to the deepest node (leaf) that is closest to the query point. Then by backtracking to a higher level each time, we check if any subtree must be searched.

2.3 Complexity analysis

The time complexity to build the tree is $\mathcal{O}(n \log n)$. Time cost to search for a single nearest neighbor is $\mathcal{O}(\log n)$, since there are $\approx \log n$ levels, each involving l distance calculations, where l denotes the number of vantage points in a particular node. Space complexity is $\mathcal{O}(n)$.

To find the k nearest neighbors of a query point, we utilize a modified max heap during the tree traversal. The modification is that the size of the heap is limited to k, so each time we only keep the *k-th* closest points.

Each time the distance is calculated, a comparison $\mathcal{O}(1)$ to the top element is performed, and if the new one is closer than the furthest, we pop the latter $\mathcal{O}(\log(n))$ and insert the former $\mathcal{O}(\log(n))$. The overall complexity is $\mathcal{O}(nodes_{visited} \log(n))$ for each query point, so $\mathcal{O}(n \cdot nodes_{visited,avg} \log(n))$ for all query points.

2.4 Distributed approach

As previously, we initially distribute data into p processes and each process builds the corresponding $vp-tree$ locally, keeping a chunk of query points privately as well. After the initialization is done, the computation follows and the first local knn results are generated by searching the local vpt one time for each local query point. Then, each *vp-tree* is sent to the neighboring process in a circular fashion. **(SEE REFERENCE)**

In order to send the vantage point trees using mpi, it was needed to serialize them in two arrays of indices and coordinates of the vantage points and right after receiving, each process uses this data to reconstruct the vpt. Finally, we search the new vpt for the local query points kNN and the procedure goes on, until every tree has passed all the processes.

By sending the tree and not the data, we reduce the needed bandwidth. To be exact, with our implementation each process sends $(1 + \frac{1}{d}) \cdot chunk\_size$ instead of $(1 + 2k) \cdot chunk\_size$.

Algorithm 1 VPT KNN-Search

```
1: procedure SEARCH(Node node, Point qp)
2:     vp ← node.vp
3:     μ ← node.mu
4:     dist ← DISTANCE(vp, qp)
5:     UPDATEKNN(vp, qp)
6:     τ ← heap.top
7:     if IS_LEAF(node) then
8:         LEAFKNN(node, qp)
9:         τ ← heap.top
10:    else
11:        if dist < μ then
12:            SEARCH(node.left, qp)
13:            if dist > μ − τ then
14:                SEARCH_SUBTREE(node.right, qp)
15:            end if
16:        else
17:            SEARCH(node.right, qp)
18:            if dist < μ + τ then
19:                SEARCH_SUBTREE(node.left, qp)
20:            end if
21:        end if
22:    end if
23: end procedure
```

2.4.1 Serialization

Since mpi does not natively support sending complex data structures, there is the need to serialize the $vp-tree$ into arrays of primitive data types. To achieve this, we only need two 1D arrays, containing the point indices & coordinates.

2.4.2 Reconstruction

All the information needed to reconstruct the tree from the two arrays is contained in the way the points are placed in it. That is, when the tree build is performed, the points are partitioned in a way that we can immediately recover each node's μ by finding the median inside the transmitted array.

The rebuild process is significantly faster ($\approx 20 - 100$ times) than building the tree each time new points are received, since the *nth_element* routine is not called at all, and μ is embedded in the data placement.

3 MPI

As mentioned, the *Message Passing Interface* is used for all the distribution and communication purposes. The chosen architecture is a circular ring where each node is a process, and data exchanges happen between the next, current and previous node.

A particular danger using this architecture is *deadlocks*. If every node is sending and none is receiving, a deadlock is the result and the application never terminates if we use blocking routines because every node tries to send data to the next one and blocks until the recipient accepts those data. This can be fixed by setting every odd-indexed node to first receive data and then send them, or by asynchronous communication.

Asynchronous communication *deadlock* is the best solution to such problems, since no blocking is performed until we ask the application so, and we can perform all computations while data is transferred in the background.

Each node has a separate receiving and sending buffer for asynchronous data communications. During the communications, knn is performed and the loop starts again after all the procedures terminate (data is transferred successfully).

Algorithm 2 MPI vp-tree

procedure SEARCH(Node $node$, Point qp)
  $X \leftarrow Y$
  for i in p **do**
    MPI_IRECV($Z_{indices}, Z_{coords}, process_{prev}$)
    MPI_ISEND($X_{indices}, X_{coords}, process_{next}$)
    BUILDTREE(X)
    KNN(Y, X)
    MPI_WAITALL
    $X \leftarrow Z$
  end for
end procedure

4 Performance

We present in the following figures the speedup (sequential runtime / distributed runtime) of our implementations for specific numbers of processors. Also, we declare the v1 / v2 speedup, as a metric to express the improvement that vantage point tree implementation of v2 offers to kNN search against all to all kNN search of v1.

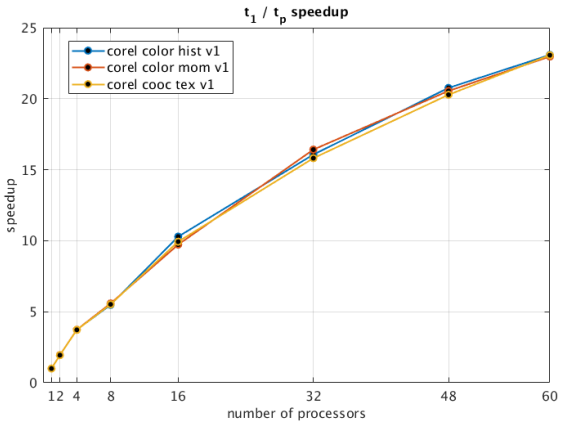
The results were recorded after our experiments in AUTh HPC (High Performance Computing). Our datasets characteristics are presented in table 1 and the results for v1 and v2 respectively in tables 2 and 3. All our experiments use $k = 50$.

4.1 Corel Datasets

These datasets have constant number of points and variable number of dimensions, so they allow us to directly compare the two algorithms for those attributes specifically.

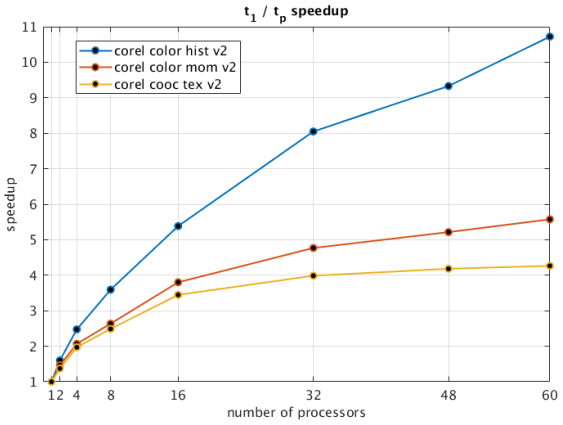
4.1.1 V1

- Sequential time almost the same for all datasets.
- Speedup almost the same as well. Seems that only n determines the operation time and the speedup in that case.



4.1.2 V2

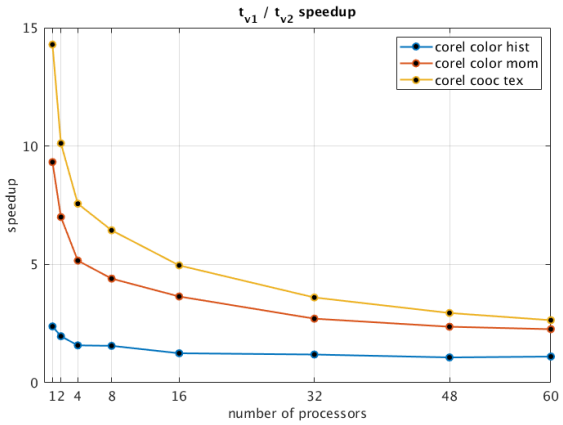
- Sequential algorithm is faster on *cooc* texture, color moments follows and color histogram comes last. We cannot export any 'rule' with respect to n and d.
- Distributed speedup is by far better on color histogram. We observe that the slower a matrix is with 1 processor, the better acceleration it shows for increasing number of processes. For 60 processors, the order remains the same.

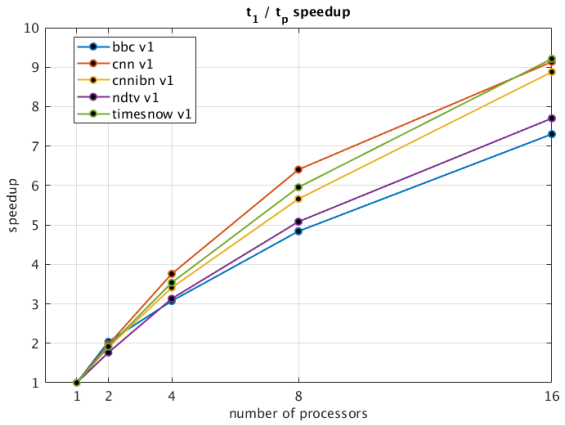


4.1.3 V1 vs V2

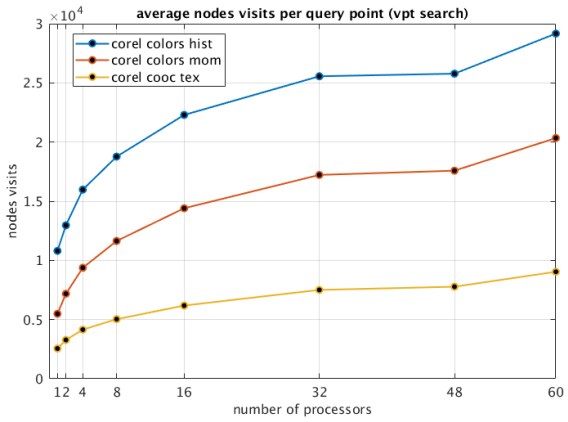
This is a metric that expresses the improvement that vantage point tree kNN algorithm offers against all-to-all kNN, for each number of processors. We observe that the greatest improvement takes place in the case of 1 processor for all matrices. The improvement follows the order of speedup of the previous case. *Histogram* has the best performance, followed by *moments* and *texture*.

Meanwhile, it is clear that the increase of the number of processes, leads v2 operation time to approach v1 time (As speedup approaches value 1, v1 and v2 time are getting closer).





This means that the improvement of vpt kNN algorithms decreases while we distribute the dataset to more processors. Trying to explain this behavior, we counted the average number of nodes, that each query point needed to visit (and use them to update its kNN), searching the tree (note that every leaf point here was counted as a node too). The results show that using more processors - that means creating more and smaller local trees - makes the algorithm search more nodes, as the average nodes that a query point visits increases. This seems reasonable, as every process has a smaller vpt, so less vantage points, that leads to less and bigger sub-spaces. As a result, the kNN search is less directed and more random in comparison with a case of a bigger vpt and the searches increase.



4.2 TV News Commercial Datasets

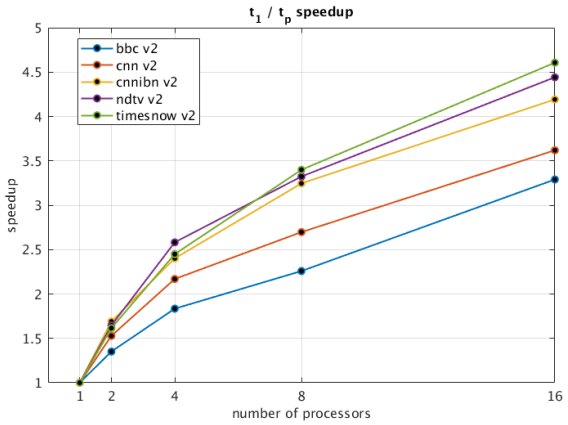
Variable number of points, constant number of dimensions.

4.2.1 V1

- As expected, higher number of samples comes with higher operation time.
- Seems that the bigger matrices perform a little better in distributed operation in comparison to smaller ones, as they present higher values of speedup. We could say that the overall performance of the distributed algorithm is very satisfying and for small numbers of processors approaches linear increase.

4.2.2 V2

- Speedup is random with respect to n.
- Quite worse than the one of v1, for the reason we explained in section 4.1.
- However, the performance of v2 algorithm is great. The maximum v1 / v2 speedup is above 17 (cnnibn, 1 processor). For increasing number of processors, v2 time approaches v1 time as previously.



4.3 FMA Datasets

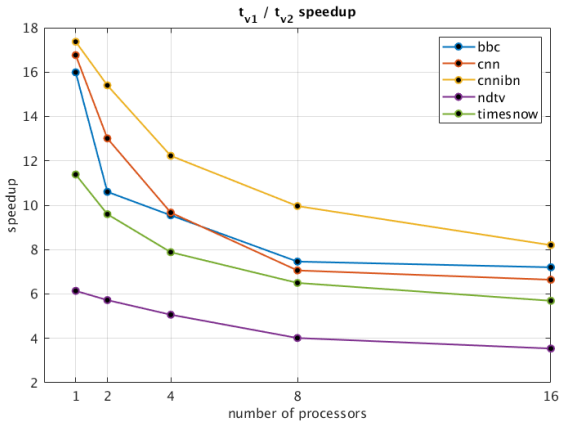
Very large number of points and dimensions.

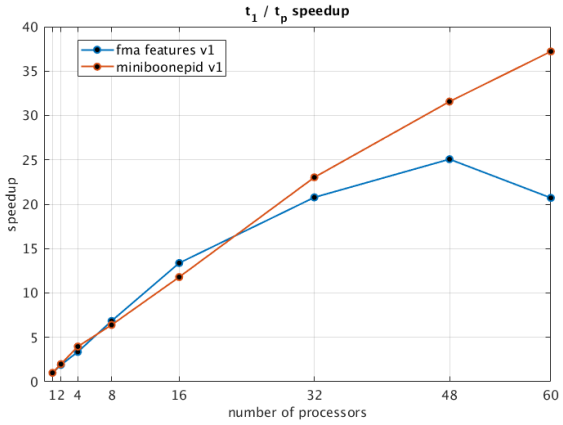
4.3.1 V1

- Great speedup for $p \leq 48$, almost linear (probably due to runtime anomaly).
- BLAS routines seems to do great job for high values of d and the performance is not affected during distance matrix D computation.

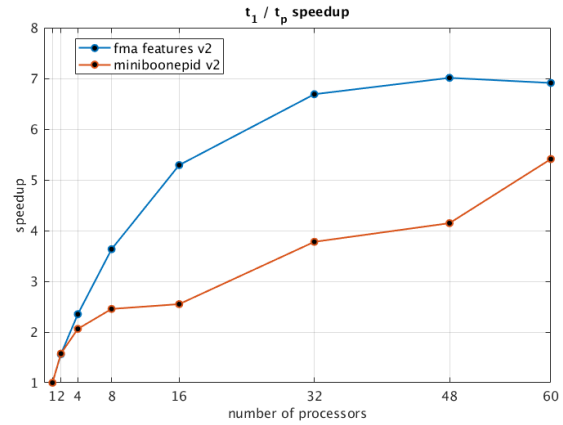
4.3.2 V2

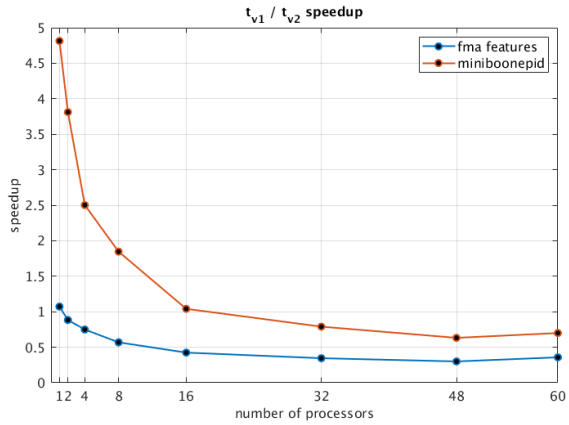
- Speedup is much worse in comparison to v1, seems that the high number of samples and dimensions affects the efficiency of the distributed algorithm as it is also noticed in the original vp-tree paper (more node searching).





- V1 seems to perform better as the number of processors increases. BLAS routines that only used in v1, perform better since multiple distances are calculated at once and the operation is not affected by the high number of dimensions, instead to discrete calculations in *vp-tree* implementation. As stated in the original paper, with more dimensions, the number of node searches increases, so our results seem correct.





4.4 MinibooNE PID Datasets

4.4.1 V1

- Great speedup, almost linear scaling.
- Most probably keeps improving after 60 processors.

4.4.2 V2

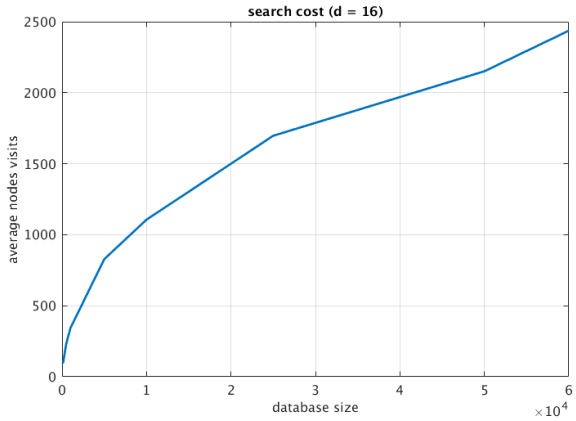
- Worse speedup than v1 due to many node searches. Should scale similarly for more than 60 processes too.
- Mediocre improvement against v1, $max = 4.8$ for 1 processor, after 16 processors v1 performs better (speedup < 1).

5 Analysis of *vp-tree* KNN search cost

The purpose of using a vp-tree vp-tree, is to decrease the number of the comparisons that take place during the kNN computation for every single query point. However, as we noticed in 4.1.3, our try to distribute the dataset to p processes, leads to the increase of the number of these comparisons (node visits). At this point we will define the search cost, as the average number of comparisons that take place per query point during kNN computation, as a metric of the data structure efficiency. The focus of this section is to compare search cost of a global vpt against the overall search cost of smaller trees, distributed in p processors. Our test was the following: for increasing dataset size, we measured the average nodes that a single query point visited during the kNN computation. We noticed that the search cost increased logarithmically with respect to dataset size (as mentioned in the original paper). The results are presented in fig **??**:

As we mentioned earlier, for increasing processes search cost increases too. In most cases, parallelism outweighs this drawback and the execution time keeps benefiting albeit with slower acceleration due to the increase of the search cost.

The focus of this section is to compare search cost of a global vpt against the overall search cost of smaller trees, distributed in p processors. Our test was the following: for increasing dataset size, we measured the average nodes that a single query point visited during the kNN computation. We noticed that the search cost increased logarithmically with respect to dataset size (as mentioned in the original paper).



Now, we assume a dataset of $n = 60000$ and $d = 16$. We are going to analyze the search cost of the distributed algorithm. When we use p processors, p local trees are created and each one consist of $60000/p$ local corpus points. So, all we have to do is search p trees of $60000/p$ points.

The theoretical execution times for each number of processes, according to the figure, are the following:

$p = 1 \rightarrow cost = y(60000) = 2437$

$p = 2 \rightarrow cost = 2y(30000) = 2 * 1800 = 3600$

$p = 4 \rightarrow cost = 4y(15000) = 4 * 1300 = 5200$

$p = 8 \rightarrow cost = 8y(7500) = 8 * 900 = 7200$

We tested the distributed version of our code using this dataset and the experiment confirmed our theoretical calculations, as we noticed cost increase for increasing number of processes.

We have proven that searching more and smaller trees is more costly than searching a global one. However, as we have already mentioned, parallelism outweigh this behavior and speedup remains in our distributed version.

6 Discussion

Overall, our experiments were close to our theoretical calculations, and the tree showed a significant increase in performance, albeit not so much in scalability compared to the naive approach. They also generally follow the results written in the original *vp-tree* paper, as the pattern for node searches and database size (points number) is pretty similar to it.

	cc histogram	cc moments	cc texture	miniboone	fma	bbc	cnn	cnnibn	ndtv	timesnow
n	68040	68040	68040	130064	106574	17720	22545	33117	17051	39252
d	32	9	16	50	518	17	17	17	17	17

Table 1: Datasets

p	cc histogram	cc moments	cc texture	miniboone	fma	bbc	cnn	cnnibn	ndtv	timesnow
1	77562.7	74437.9	74844.9	238376	520072.6	5319.97	9657.02	18029.9	4969.14	25186
2	40002.1	38173.3	38582.3	119897	274190.53	2611.6	4904.23	9459.12	2815.15	13141.6
4	20790.6	19931.8	20105.7	60098.8	154592.4	1731.61	2567.88	5286.9	1587.57	7120.98
8	14148.7	13329.3	13556.3	37243	75923.2	1099.13	1508.05	3185.74	977.597	4229.33
16	7538.6	7653.96	7533.41	20216	38885.5	728.529	1057.11	2030.79	645.079	2733.85
32	4827.45	4534.36	4733.43	10350.8	25037.1					
48	3735.89	3622.79	3690.89	7551.66	20744.7					
60	3359.67	3240.8	3244.92	6405.85	25107.7					

Table 2: V1 Execution Time (ms)

p	cc histogram	cc moments	cc texture	miniboone	fma	bbc	cnn	cnnibn	ndtv	timesnow
1	32594.5	7984.91	5237.86	49515.9	484865	332.744	576.034	1038.02	809.125	2212.08
2	20356.9	5452.85	3814	31458.5	310285	246.289	376.915	614.14	491.827	1369.55
4	13169.3	3862.57	2658.13	24016.13	206237	181.389	265.583	432.348	313.382	903.358
8	9072.83	3029.52	2104.81	20169.15	133446	147.338	213.567	319.798	243.445	650.778
16	6053.36	2100.43	1519.6	19419.1	91637	101.142	159.161	247.616	182.167	480.224
32	4049.75	1674.84	1314.07	13105.2	72492					
48	3494.06	1531.26	1252.35	11941.7	69152					
60	3040.32	1432.47	1228.36	9151.75	70158					

Table 3: V2 Execution Time (ms)