UNIHIST: Fast histogram and weighted random sampling with replacement for data-parallel devices

*****

*****

***Abstract* — We introduce the fast histogram computing algorithm for non-uniform bins that avoids bin-search and, instead, uses predictor-corrector technique for bin indices refinement. Approximate bin indices are computed using parallel map operation, which makes the algorithm suitable for implementation for data-parallel devices. We demonstrate superior performance of the method using uniform map functions, but the method is easily extensible to other maps. It is also extensible to multi-dimensional histogram computation. Finally, we demonstrate how the same technique can be applied for generalized discrete random number generation. Our implementation leverages oneAPI DPC++ compiler to run the code on both CPU and GPU devices and provides Python wrappers for comparison with NumPy, CuPy, and easy integration with Data Parallel Extensions for Python.**

Keywords — histogram, discrete random number generation, data-parallel computing, GPU, SYCL, oneAPI, Data Parallel Extensions for Python, NumPy, dpctl, dpnp

I. Introduction

Histograms are used for decades as a tool for compact representation of data, statistical analysis, and quantization in variety of applications and became increasingly popular in recent years as a component of data science algorithms [2], [3], [4], [5], [6], [7].

Many research papers deal with uniform bins due to simplicity of binning points into a given bin (e.g., [10] and references therein). For this reason, the research was traditionally focused on how to select the optimal number of bins to avoid over-smoothing but still capture meaningful frequency peaks.

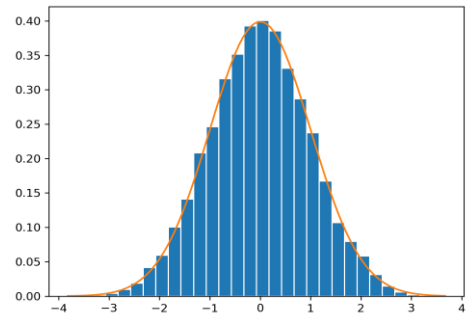


Fig. 1 An example of uniform bins histogram

The counterapproach is to use non-uniform bins aimed to adjust bin size to capture local peaks and, when needed, to catch tail distribution effects [12].

While the selection of optimal number of bins remains an important subject, the equally important problem becomes bin widths optimization with respect to *a priori* distribution or to actual observations [20], [21].

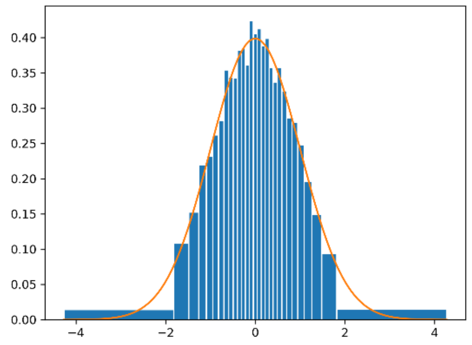


Fig. 2. An example of non-uniform bins histogram

Another equally important aspect of dealing with histograms of varying bin width is fast binning techniques [8], [9]. In general case the binning is equivalent to a search problem, which has $n \cdot \log k$ complexity, where n is the number of points in a dataset and k is the number of bins. In turn, various techniques suggest k to be proportional to $\sqrt{n}$, $n^{2/5}$, $\sqrt[3]{n}$, $\log n$, *etc*. and can be large so that the search time cannot be neglected [13], [14], [15], [16], [17], [18], [19].

The focus of this paper is a novel method for binning many points in a dataset with non-uniform bins. The novelty is in substituting the search with the predictor-corrector approach, where a simple map function provides an initial guess of a correct bin combined with a procedure for an iterative refinement of the guess until the correct bin is found. We will demonstrate that when non-uniform bin widths have not much variation the uniform-map function is the most preferred. We will also compare the performance of the method for arbitrary bin widths generated randomly. We will also discuss the generalization of the map function to substantially non-uniform distributions.

All results will be presented on a range of data-parallel hardware ranging from a laptop setting with multi-core CPU with integrated graphics to a powerful server with a large core-count CPU supporting AVX512 instructions and server-class discrete GPUs from different vendors.

While the scope of this paper is 1D histogram, the suggested method is easily extended to multi-dimensional case. Also, we will show how the histogram binning problem relates to generalized discrete distribution random number generation.

II. The Method

When the bins are all equal width, the bin localization problem is expressed by the simple map function:

$$i = \left\lfloor k\frac{x-b_0}{b_k-b_0}\right\rfloor \qquad (1)$$

where i is the bin index for a given sample point x, b_0 and b_k are the leftmost bin left edge and the rightmost bin right edge respectively, k is the number of bins, and $\lfloor \quad \rfloor$ denotes the floor (round down) operation.

The Fig. 3 illustrates the uniform map function (1), where each bin is painted in a different color.

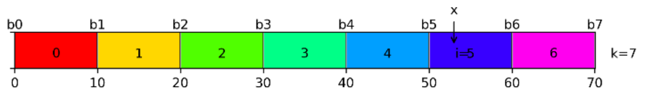
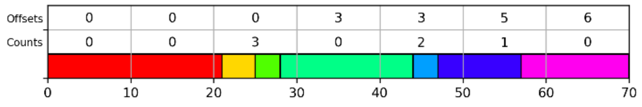


Fig. 3. Bin mapping in the case of uniform bins

We want to leverage the simplicity of the uniform map function but apply it to non-uniform bins. Imagine that our non-uniform bins partition is "almost" uniform. Then applying the formula (1) to a point x would produce the index i, which is "almost" always accurate. And only when it is inaccurate, we would need a correction procedure.

Let us formalize the idea. We now have non-uniform bins 0-7 specified by bin edges b_i = [0, 21, 25, 28, 44, 47, 57, 70], on which we map the uniform partitioning u_i = [0, 10, 20, 30, 40, 50, 60, 70] as illustrated on the Fig. 4. We again use different colors for different bins in the illustration.



Offsets	0	0	0	3	3	5	6
Counts	0	0	3	0	2	1	0

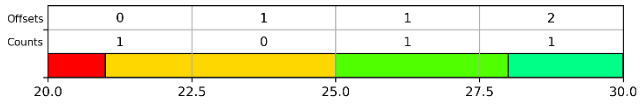
Fig. 4. Uniform map on non-uniform bins

We also associate with each uniform bin i two values, **Counts**[i] and **Offsets**[i]. The count represents the number of non-uniform bin edges in the interval i. (We do not count the left edge when it exactly matches with the left edge of the uniform interval, e.g., b_0=u_0=0). And the offset is the cumulative sum $\sum_{j=0}^{i-1}\text{Counts}[i]$.

Now for any point of interest x using formula (1) we determine the index i, and j = **Offsets**[i] would give us the non-uniform bin number. For example, if x is anywhere between 0 and 20 then the produced j = 0 indicates the correct non-uniform bin. If x is between 30 and 40 then j = 3 is accurate bin prediction.

The troubling case is when x is between 20 and 30. We cannot say, with a guarantee, whether x belongs to the bin 0, 1, 2, or 3. So, this is the case that will require the refinement procedure. And, we know we have this case because **Counts**[i] > 0.

The refinement procedure deals with the troubling interval only as illustrated on the Fig. 5.



Offsets	0	1	1	2
Counts	1	0	1	1

Fig. 5. Iterative refinement deals with the troubling interval only

We use the same procedure for the interval (20, 30], and map the uniform grid onto the non-uniform one. The number of uniform bins corresponds to the number of non-uniform bins in this interval. In this step the number of troubling intervals is 3, (20, 22.5], (25, 27.5], and (27.5, 30], for which we don't need the refinement procedure because the interval can be determined by a binary decision rule $\boldsymbol{if\ x \le b_i}$.

III. Convergence

The method will converge if on each iteration the number of troubling intervals is reduced by at least one. In practice the number of troubling intervals decreases exponentially until the refinement procedure is reduced to a binary decision rule.

More formally, the worst-case scenario is when all non-uniform bins (except the wide one) are concentrated in one uniform bin on its edge, as shown on the Fig. 6. Let us denote:

$$h_i = b_i - b_{i-1}$$
$$h_{\min} = \min_i h_i$$
$$h = b_k - b_0$$

The worst case is when $h_1 = \cdots = h_{k-1} = h_{\min}$ and $h_k = h - (k-1)h_{\min}$.

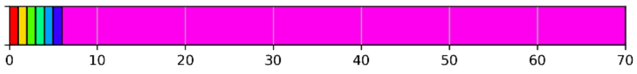


Fig. 6. Worst case bin setting

That will require multiple refinements for the first interval. Let us denote h_k^i the length of the rightmost non-uniform interval on i-th refinement, which is

$$h_k^i = h \cdot k^{-i} - h_{\min} \cdot (k-1)$$

Refinement will continue while $h_k^i > h_{\min}$, i.e., while $h > h_{\min} \cdot k^{i+1}$. The refinement will stop after n iterations, where:

$$n = \left\lfloor \frac{\log h/h_{\min}}{\log k} \right\rfloor - 1 \qquad (5)$$

One can note that in the worst-case on each step the interval is shrinking by a factor of k. The binary search instead reduces the number of bins in half, which may find the required bin in less iterations than using our approach. In section VI.C we will discuss effective ways to address convergence for substantially non-uniform cases.

IV. Implementation

The complete reference implementation along with optimized variants are available on the GitHub [1]. Compared to the reference schema outlined in the section II and implemented in Python, our optimized implementation relies on oneAPI Intel® DPC++ compiler [25], being able to compile the same code for a range of SYCL devices. Using `pybind11` [27] we implemented NumPy-friendly Python wrapper for performance comparisons with other popular implementations in Python (`numpy.histogram` and `cupy.histogram`).

In the optimized implementation of UNIHIST we use a slightly modified algorithm for better performance and memory footprint:

1. We do not store **Counts**[i], we only store **Offsets**[i]. Indeed, the **Offsets**[i] is a cumulative sum of **Counts**[i], and hence, **Counts**[i] are easily derived from **Offsets**[i].
2. We do not perform correction steps recursively. Instead, we pre-compute the entire mesh for the worst-case depth (5) discussed in the Section III. This is essential technique because SYCL kernels do not support recursion.
3. In the reference schema the correction procedure continues until we reduce the number of bins in the interval of interest to a binary decision rule $\boldsymbol{if\ x \leq b_i}$. In the optimized version we found that we can conclude the correction earlier, when the interval of interest consists of l bins (in our experiments $l = 4$ was found to be nearly optimal) and perform binning using vectorized linear search.
4. We also tuned algorithm by selecting optimal work-group size and performed data blocking to leverage local memory. We also apply certain optimization techniques to minimize the use of atomic reductions. Please refer to GitHub sources for details [1].

V. Results

Since our objective is to provide an optimized implementation targeting variety of hardware options, ranging from laptop CPUs with integrated graphics to server-class CPUs and powerful discrete GPUs, we present performance results for the following settings:

- **(LAPTOP CPU/GPU)** 12th Gen Intel® Core™ i7-1270P (2.5 GHz, 8 cores/16 threads) with Intel® Iris® Xe Graphics (1.4GHz, 96 execution units)
- **(SERVER CPU)** Intel® Xeon® Platinum 8480+ (2.0-3.8 GHz, 2x56 cores/224 threads)
- **(SERVER GPU)** Intel® Data Center GPU Max 1550 (0.9-1.6 GHz, 128 cores, 1024 vector engines)
- **(SERVER GPU)** NVIDIA A100 80GB PCIe GPU

We tested performance of the following histogram implementations:

- **UNIHIST**. Proposed histogram implemented in DPC++ and wrapped in Python with pybind11.
- **BINARY**. Binary search implemented in DPC++ and wrapped in Python with pybind11.
- **NUMPY**. The `numpy.histogram` implementation.
- **CUPY**. The `cupy.histogram` implementation.

The following software was installed with `conda` and used in performance measurements:

- `dpcpp_linux-64` 2023.1.0 (Intel channel) [31]
- `python` 3.10.8 (Intel channel)
- `numpy` 1.23.5 (Intel channel) [32]
- `dpctl` 0.14.2 (Intel channel) [33]
- `dpnp` 0.11.2dev0 (dppy/label/dev channel) [34]
- `cudatoolkit` 11.8.0 (Conda-Forge channel) [35]
- `cupy` 12.0.0 (Conda-Forge channel) [36]

For performance testing n single precision data points were uniformly generated on the interval (0, h), where h=1000. We choose n equal to 102,400,000 for LAPTOP CPU/GPU and 2,048,000,000 for SERVER CPU/GPU. Bins were generated using several methods:

- **Worst-case bins allocation** as described in the Section III. We present performance for different settings of $h_{\min}$.
- **Random bin widths.** Bin widths are random but each bin $h_i \geq h_{\min}$. We present performance for different settings of $h_{\min}$.
- **"Almost" uniformly allocated bins.** Bins are equal widths with small random variations $(-h_v, +h_v)$. We present performance for different settings of h_v.

We tested for the number of bins k equal to 100 and 1000.

TABLE I. Worst-case bin allocation. Laptop CPU

Variant	Performance (throughput Mpts/sec)			
	k	$h_{min} = 0.1$	$h_{min} = 0.01$	$h_{min} = 0.001$
NUMPY	100	22	22	22
	1000	21	22	22
BINARY	100	628	458	477
	1000	333	328	329
UNIHIST	100	1783	1298	1285
	1000	860	1267	1395

TABLE II. RANDOM BIN WIDTHS. LAPTOP CPU

Variant	**Performance (throughput Mpts/sec)**			
	$\boldsymbol{k}$	$\boldsymbol{h_{min} = 0.1}$	$\boldsymbol{h_{min} = 0.01}$	$\boldsymbol{h_{min} = 0.001}$
NUMPY	100	22	21	20
	1000	20	21	21
BINARY	100	667	675	470
	1000	338	333	349
UNIHIST	100	1336	1340	920
	1000	885	886	900

TABLE III. "ALMOST" UNIFORMLY ALLOCATED BINS. LAPTOP CPU

Variant	**Performance (throughput Mpts/sec)**			
	$\boldsymbol{k}$	$\boldsymbol{h_v = 0.1}$	$\boldsymbol{h_v = 0.01}$	$\boldsymbol{h_v = 0.001}$
NUMPY	100	22	22	22
	1000	21	21	22
BINARY	100	675	680	446
	1000	341	340	344
UNIHIST	100	2106	2105	1432
	1000	1517	1336	1427

NUMPY performance does not depend on bin allocation and the number of bins, but its performance is an order of magnitude lower than our implementations for BINARY and UNIHIST. BINARY performance somewhat sensitive to h_{min} and h_{v} and sensitive to the number of bins (it is nearly twice slower for 1000 bins than for 100 bins). UNIHIST behavior is similar to BINARY but performance is measurably better, especially for "almost" uniform bins, which is expected by design.

TABLE IV. WORST-CASE BIN ALLOCATION. LAPTOP GPU

Variant	**Performance (throughput Mpts/sec)**			
	$\boldsymbol{k}$	$\boldsymbol{h_{min} = 0.1}$	$\boldsymbol{h_{min} = 0.01}$	$\boldsymbol{h_{min} = 0.001}$
BINARY	100	2846	2837	2827
	1000	2032	2031	2049
UNIHIST	100	5270	5005	5030
	1000	4017	5889	6361

TABLE V. RANDOM BIN WIDTHS. LAPTOP GPU

Variant	**Performance (throughput Mpts/sec)**			
	$\boldsymbol{k}$	$\boldsymbol{h_{min} = 0.1}$	$\boldsymbol{h_{min} = 0.01}$	$\boldsymbol{h_{min} = 0.001}$
BINARY	100	1151	1151	1151
	1000	1280	1234	1289
UNIHIST	100	4614	4607	4611
	1000	2455	2449	2446

TABLE VI. "ALMOST" UNIFORMLY ALLOCATED BINS. LAPTOP GPU

Variant	**Performance (throughput Mpts/sec)**			
	$\boldsymbol{k}$	$\boldsymbol{h_v = 0.1}$	$\boldsymbol{h_v = 0.01}$	$\boldsymbol{h_{min} = 0.001}$
BINARY	100	1113	1113	1122
	1000	1312	1268	1281
UNIHIST	100	4631	4693	4723
	1000	3986	3985	3983

GPU performance depends on the number of bins but does not vary much with h_{min} and h_{v}. The "almost" uniform bins setting is not the performance leader for UNIHIST, rather the worst-case bin allocation performs better. This is explained by the fact that, even such a bin allocation represents the deepest correction mesh, in most cases points fall into the widest bin, for which no correction procedure is required. It's likely that if we sample most points within thinnest intervals then UNIHIST behavior will be much worse. The Section VI.C discusses the substantially non-uniform bins case.

TABLE VII. WORST-CASE BIN ALLOCATION. SERVER CPU

Variant	**Performance (throughput Mpts/sec)**			
	$\boldsymbol{k}$	$\boldsymbol{h_{min} = 0.1}$	$\boldsymbol{h_{min} = 0.01}$	$\boldsymbol{h_{min} = 0.001}$
BINARY	100	14167	15837	19098
	1000	11569	14887	15453
UNIHIST	100	23664	28312	28510
	1000	21161	27341	32480

TABLE VIII. RANDOM BIN WIDTHS. SERVER CPU

Variant	**Performance (throughput Mpts/sec)**			
	$\boldsymbol{k}$	$\boldsymbol{h_{min} = 0.1}$	$\boldsymbol{h_{min} = 0.01}$	$\boldsymbol{h_{min} = 0.001}$
BINARY	100	13137	17655	17587
	1000	10640	10510	10562
UNIHIST	100	21224	23087	23515
	1000	21829	21846	21910

TABLE IX. "ALMOST" UNIFORMLY ALLOCATED BINS. SERVER CPU

Variant	**Performance (throughput Mpts/sec)**			
	$\boldsymbol{k}$	$\boldsymbol{h_v = 0.1}$	$\boldsymbol{h_v = 0.01}$	$\boldsymbol{h_v = 0.001}$
BINARY	100	13018	14807	18953
	1000	9872	9744	9764
UNIHIST	100	26933	29735	29939
	1000	27296	27530	27667

In the SERVER CPU setting we increase the number of sample points by a factor of 20, which makes NumPy run time very long. That's why we excluded it from the measurements. Both BINARY and UNIHIST scale well to the large number of CPU cores, but UNIHIST is about twice faster.

In the SERVER GPU setting we use different server-class GPUs, the one from NVIDIA to benchmark CUPY, and the one from Intel to benchmark BINARY and UNIHIST. While DPC++ can target compilation for SYCL devices from NVIDIA, it was not the goal of our study.

TABLE X. WORST-CASE BIN ALLOCATION. SERVER GPU

Variant	**Performance (throughput Mpts/sec)**			
	$\boldsymbol{k}$	$\boldsymbol{h_{min} = 0.1}$	$\boldsymbol{h_{min} = 0.01}$	$\boldsymbol{h_{min} = 0.001}$
CUPY	100	72445	79220	79874
	1000	86057	89117	90921
BINARY	100	34810	34698	34625
	1000	35114	33705	33405
UNIHIST	100	37800	37076	37215
	1000	40581	37900	38212

TABLE XI. RANDOM BIN WIDTHS. SERVER GPU

Variant	**Performance (throughput Mpts/sec)**			
	$\boldsymbol{k}$	$\boldsymbol{h_{min} = 0.1}$	$\boldsymbol{h_{min} = 0.01}$	$\boldsymbol{h_{min} = 0.001}$
CUPY	100	66940	66925	66983
	1000	49424	49434	49428
BINARY	100	58709	58893	58922
	1000	41929	41951	41909
UNIHIST	100	121040	121009	120420
	1000	95137	95271	95505

TABLE XII. "ALMOST" UNIFORMLY ALLOCATED BINS. SERVER GPU

Variant	**Performance (throughput Mpts/sec)**			
	$\boldsymbol{k}$	$\boldsymbol{h_v = 0.1}$	$\boldsymbol{h_v = 0.01}$	$\boldsymbol{h_v = 0.001}$
CUPY	100	80492	79950	80227
	1000	49851	49675	50030
BINARY	100	58560	58646	58600
	1000	42311	42067	42242
UNIHIST	100	150968	151138	151218
	1000	100518	101992	98827

The worst-case allocation is favorable to CUPY. Our implementation of BINARY and UNIHIST are not performing well. Perhaps, additional parameter tuning is required for improving the worst-case behavior.

BINARY is almost on-par with CUPY on random bins and somewhat behind on "almost" uniform bins.

UNIHIST is a clear winner for both random bins and "almost" uniform bins, especially when the number of bins is large ($k = 1000$).

Overall additional study is required to better understand the performance behavior on different types of hardware. Currently we have vague understanding of performance bottlenecks and whether significant optimization opportunities exist. Programming in DPC++ is relatively new experience for us. We will extend this study as DPC++ and Data Parallel Extensions for Python tools mature.

VI. GENERALIZATIONS

A. *Sampling from the discrete uniform distribution*

One can notice that the map function (1) corresponds to the Inverse Cumulative Distribution Function (ICDF) for the discrete uniform distribution [24]. Sampling from that distribution is trivial:

1. Generate the random number x, uniformly distributed in the interval $[u_0, u_k]$
2. Apply (1) to get random integer $0 \leq i \leq k$.

B. *Sampling from the generalized discrete distribution*

Many applications require random number sampling from a population of size k, where the probability to draw a particular member i is p_i, where $\sum p_i$=1. This probability distribution is called the generalized discrete distribution. It is also known as a weighted random sampling with replacement. There are many effective methods developed for sampling from this distribution [22], [23], [24]. We adapt the histogram bin refinement procedure to suggest another effective algorithm for random number j from the generalized discrete distribution.

1. Let us construct bins b_i in the following way:
   - $b_0 = 0$
   - $b_i = b_{i-1} + p_i,\ i = \overline{1, k}$

   Note that $b_k = 1$.
2. Accumulate **Counts**[i] and **Offsets**[i] according to the original schema presented in the Section II.
3. Generate a random number x uniformly distributed in the in the interval $[u_0, u_k]$, i.e. [0, 1].
4. Apply formula (1) to get i.
5. For the bins with **Counts**[i] > 0 we need a refinement, otherwise we know that j = **Offsets**[i] gives us the correct bin. This j will be a population member generated with the probability p_j.

Here we present very preliminary performance results only, which should be considered with a "grain of salt". We tested RNG performance for n equal to 102,400,000 for LAPTOP CPU/GPU and 2,048,000,000 for SERVER CPU/GPU. Population size was chosen $k = 1000$, $p_i = 0.001$.

TABLE XIII. RANDOM SAMPLING WITH REPLACEMENT

Variant	**Performance (throughput Mpts/sec)**			
	LAPTOP CPU	**LAPTOP GPU**	**SERVER CPU**	**SERVER GPU**
NUMPY random choice	25	N/A	N/A	N/A
UNIHIST random choice	494	336	246	899
CUPY random choice	N/A	N/A	N/A	Out-of-memory

NUMPY is the slowest in our measurements, which can be explained that it does not take advantage of multiple cores. UNIHIST-based implementation of the `random.choice` relies on `dpnp.random.random` [34] to produce uniformly distributed random numbers on the unit interval. While UNIHIST performance numbers look impressive compared to NUMPY, our measurements show that we spend less that 1% of time in UNIHIST, where more than 99% of remaining time is spent in random number generation, which is a clear indication of performance gaps in `dpnp`. We were also unable to obtain results for CUPY because it runs out of memory on the selected problem size. The largest problem we successfully run with CUPY is $n = 1{,}048{,}576$. For this reason, we do not include CUPY measurements in the Table XIII.

C. *Better guessing for "substantially" non-uniform bins*

By taking this idea further, what if our data follows "closely" some other probability distribution? Then taking bin widths according to this distribution will allow us to effectively map data points to respective non-uniform histogram bins.

Let's take the exponential distribution as an example shown on the Fig. 7. We select k bins using the following approach. Create the uniform mesh $0 \le v_0 < \dots < v_k \le 1$, where:

$$v_i = v_0 + i \cdot \frac{v_k - v_0}{k} \quad (6)$$

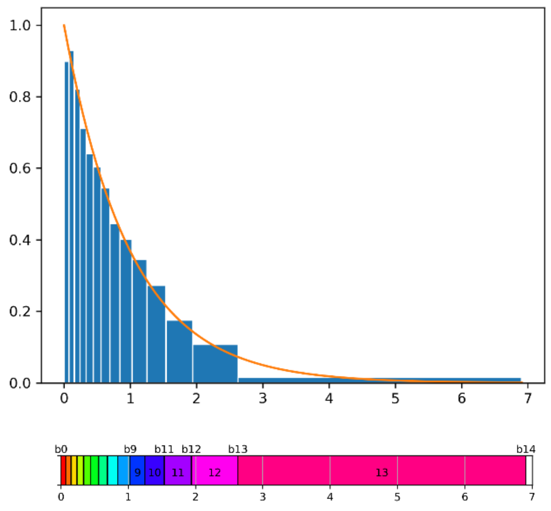


Fig. 7. Non-uniform bins that follow exponential distribution ICDF

Select non-uniform bins as follows:

$$b_i = \mathrm{F}^{-1}(v_i),$$

where $\mathrm{F}^{-1}(v)$ is the ICDF. For exponential distribution $\mathrm{F}^{-1}(v) = -\log(1-v)$. Now if we have a set of points x, which we want to place in bins, and which follow exponential distribution "to some degree", then we can adapt the non-uniform bins refinement procedure as follows. Instead of (1) we now will use the following rule for the bin guess:

$$i = \left\lfloor k \cdot \frac{\mathrm{F}(x) - v_0}{v_k - v_0} \right\rfloor \quad (7)$$

Our uniform bins u_i are constructed the same way as before. And we still accumulate **Counts**$[i]$ and **Offsets**$[i]$. For uniform bins with **Counts**$[i] > 0$ we need a refinement, otherwise we know that j = **Offsets**$[i]$ gives us the correct bin. This method works for any discrete distribution $\mathrm{F}(x)$. One key consideration is the cost of computation of the Cumulative Distribution Function (CDF) $\mathrm{F}(x)$, which we will need to compute for each x. We also cannot ignore the cost of ICDF computation required for bins initialization.

Note, that the beauty of our method is in the loose requirement how accurate $\mathrm{F}(x)$ needs to be computed. That leaves a variety of options how it can be approximated using parametric and non-parametric techniques [26].

D. *Multidimensional case*

We consider the most general case described in section V.C. In general case we have k_m bins in each dimension $1 \le m \le d$. For example, on Fig. 8 we illustrate 2D histogram $(d = 2)$, with $k_1 = 4$ and $k_2 = 2$. According to (6) we generate uniform mesh $0 \le v_o^m < \dots < v_k^m \le 1$ for each axis. Non-uniform bins will be constructed as follows:

$$b_i^m = F_m^{-1}(v_i^m),$$

Here $F_m(x) = F(+\infty, \dots, +\infty, x, +\infty, \dots, +\infty)$ and x is in the position of m-th dimension. With that formula (7) for m-th dimension would be as follows:

$$i^m = \left\lfloor k^m \cdot \frac{\mathrm{F}_m(x) - v_0^m}{v_k^m - v_0^m} \right\rfloor \quad (8)$$

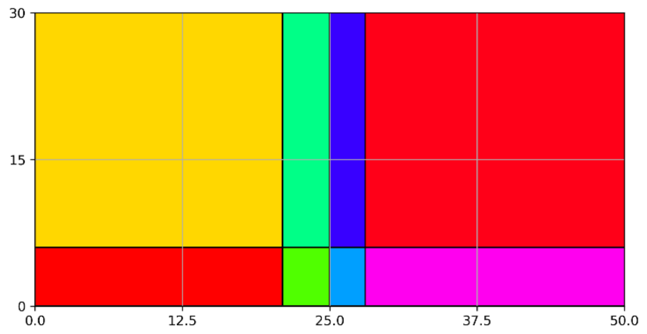


Fig. 8. 2D histogram with non-equal size bins create non-uniform 2D mesh

We will collect **Counts**$[i^1, \dots, i^d]$ and **Offsets**$[i^1, \dots, i^d]$ and apply refinement procedure for a given bin $i^1, \dots, i^d$ if **Counts**$[i^1, \dots, i^d] > 0$.

Conclusions

We presented novel methods for histogram computation with non-equal bin widths and for random sampling with replacement. Our corrector-predictor-based approach uses a simple map operation to map many sample points in parallel to many bins with a repetitive correction step, making the methods suitable for effective implementation of the UNIHIST on data-parallel devices.

Using DPC++ compiler we target the same code for both CPU and GPU SYCL devices, however device-specific parameter tuning (such as the work-group size) was needed for each device.

UNIHIST in most settings demonstrated superior performance compared to other library implementations (NUMPY and CUPY) as well as compared to our implementation of the binary search BINARY written in DPC++ language.

Future work

This work is an initial prototype to illustrate the idea of the histogram computation and random number sampling from the generalized discrete distribution with UNIHIST.

Our end objective is to contribute the implementations to Data Parallel Extensions for Python [28].

We also want to contribute the implementations to NumPy community as optimized `numpy.histogram()` and `numpy.random.choice()` functions for CPU devices [29], [30].

Additional study is required to understand the UNIHIST's worst-case behavior using different sampling methods. We also need to explore CDF approximation options for substantially non-uniform case (7).

A separate direction for additional studies is to apply the UNIHIST for random sampling from discrete probability distributions, such as Poisson, binomial, negative binomial, and multinomial distributions. For these distributions the widely used sampling technique is the acceptance-rejection method [24]. We think that the UNIHIST (1) and its substantially non-uniform variation (7) may be a viable alternative for these distributions.

2D and 3D histograms are of particular interest in many data science and engineering applications, which can be another direction of the UNIHIST exploration.

REFERENCES

[1] Python and DPC++ implementations algirithm available at https://github.com/pycoddiy/unihist

[2] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T-Y Liu, "LightGBM: a highly efficient gradient boosting decision tree", NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017, pp. 3149-3157.

[3] Z. Wang, H. Li, W. Quyang, X. Wang, "Learnable Histogram: Statistica Context Features for Deep Neural Networks", European Conference on Computer Vision, 2016, pp. 246-262.

[4] B. Xiao, Y. Xu, H. Tang, X. Bi, W. Li, "Histogram Learning in Image Contrast Enhancement", IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, 2019, pp. 1880-1889

[5] Y.J. Ong, Y. Zhou, N. Baracaldo, H. Ludwig, "Adaptive Histogram-Based Gradient Boosted Trees for Federated Learning", 2020, https://arxiv.org/abs/2012.06670

[6] H. Sadeghi, A.-A. Raie, "HistNet: Histogram-based convolutional neural network with Chi-squared deep metric learning for facial expression recognition", Information Sciences, Vol. 608, 2022, pp. 472-488

[7] B. Bhattarai, R. Subedi, R.R. Gaire, E. Vazquez, D. Stoyanov, "Histogram of Oriented Gradients meet deep learning: A novel multi-task deep network for 2D surgical image semantic segmentation", Medical Image Analysis, Vol. 85, 2023

[8] J.Gómez-Luna, J.González-Linares, J.Benavides, N.Guil, "An optimized approach to histogram computation on GPU", Machine Vision and Applications, 24(5), 2013

[9] C. Nugteren, G.J. Van Den Braak, H. Corporaal, B. Mesman, "High Performance Predictable Histograming on GPUs: Exploring and Evaluating Algorithm Trade-offs", Proceedings of the 4th Workshop on General Purpose Processing on GPU, 2011, pp.1-8.

[10] Kevin H. Knuth, "Optimal data-based binning for histograms and histogram-based probability density models", Digital Signal Processing, Vol. 95, 2019

[11] "Notes for Intel® oneAPI Math Kernel Library Vector Statistics. Chi-Squared Goodness-of-Fit Test".
https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-vector-statistics-notes/2021-1/chi-squared-goodness-of-fit-test.html

[12] L.Denby, C.Mallows. "Variations on the Histogram", Journal of Computational and Graphical Statistics, Vol 18, Issue 1, 2009, pp. 21-31

[13] H.A. Sturges, "The Choice of a Class interval", Journal of the American Statistical Association, Vol. 21, Issue 152, 1926, pp. 65-66

[14] D.P. Doane, "Aesthetic Frequency Classifications", The American Statistician, Vol. 30, Issue 4, 1976, pp. 181-183.

[15] D.W. Scott, "On optimal and data-based histograms", Biometrica, Vol. 66, Issue 3, 1979, pp. 605-610.

[16] D. Freedman, P. Diaconis, "On the Histogram as a Density Estimator: L_2 Theory", Z. Wahrscheinlichkeitstheorie verw. Gebiete 57, 453-476, 1981

[17] L. Wasserman, "All of Statistics", Springer, 2004, p.310

[18] C. Stone, "An asymptotically optimal histogram selection rule", 1984, Proceedings of the Berkley conference in homor of Jerzy Neyman and Jack Kiefer, 1984

[19] H. Shimazaki, S. Shinomoto, "A method for selecting the bin size of a time histogram", Neural Computation. 19 (6), 2007, pp. 1503–1527.

[20] J. Prins, D. McCormack, D. Michelson, K. Horrell, "Chi-square goodness-of-fit test", NIST/SEMATECH e-Handbook of Statistical Methods, 2019.

[21] D. Moore, "Tests of Chi-Squared Type", Goodness-of-Fit Techniques (Ed. R.B. D'Agostino, M.A. Stephens), Marcel Dekker Inc, 1986, pp. 63-93.

[22] G. Marsaglia, W. W.Tsang, J.Wang, (2004). "Fast Generation of Discrete Random Variables". Journal of Statistical Software, 11(3), 1–11.

[23] K.Schwarz. "Darts, Dice, and Coins: Sampling from a Discrete Distribution", https://www.keithschwarz.com/darts-dice-coins, 2011

[24] J. Gentle. "Random Number Generation and Monte Carlo Methods", 2nd Edition, Springer, 2015

[25] J. Reinders, B. Ashbaugh, J.Brodman, M.Kinsner, J.Pennycook, X.Tian, "Data Parallel C++. Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL", APRESS Open, 2021

[26] B.W. Silverman, ""Density Estimation for Statistics and Data Analysis. Monographs on Statistics and Applied Probability", Taylor & Francis, 1986

[27] PyBind11, pybind11.readthedocs.io

[28] Introduction to Data Parallel Extensions for Python https://intelpython.github.io/DPEP

[29] NumPy Histogram:
https://numpy.org/doc/stable/reference/generated/numpy.histogram.html

[30] NumPy Random Choice:
https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html

[31] Intel® oneAPI DPC++/C++ Compiler
https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html

[32] Intel Distribution for Python – What's Included
https://www.intel.com/content/www/us/en/developer/articles/tool/whats-included-distribution-for-python.html

[33] Data Parallel Control https://intelpython.github.io/dpctl/

[34] Data Parallel Extension for NumPy https://intelpython.github.io/dpnp/

[35] CUDA Toolkit https://developer.nvidia.com/cuda-toolkit

[36] CuPy - NumPy & SciPy for GPU https://docs.cupy.dev/