Flash-SD-KDE: Accelerating SD-KDE with Tensor Cores

1 Introduction

Score-debiased kernel density estimation (SD-KDE) [1] improves the statistical efficiency of standard kernel density estimators: for sufficiently smooth densities, SD-KDE attains better bias and mean-squared error rates than vanilla KDE while retaining a simple, nonparametric form. These accuracy gains come with a significant computational drawback: the empirical score used for debiasing introduces an additional $O(n^2)$ pass over the data, so a naive implementation has roughly the same quadratic cost as KDE itself but with a larger constant factor. At realistic sample sizes, this puts strong pressure on both the algorithmic structure and the hardware implementation.

Given samples x_i in one dimension and a bandwidth h, a Gaussian KDE is

$$\hat{p}(x) = \frac{1}{nh} \sum_{i=1}^{n} \varphi\left(\frac{x - x_i}{h}\right),$$

and SD-KDE forms debiased samples $x_i^{\mathrm{SD}} = x_i + \frac{h^2}{2}\,\hat{s}(x_i)$ where $\hat{s}$ is an estimate of the score. In this work we always use an empirical score computed from the KDE itself, i.e. no parametric model for the underlying density is assumed. Writing the KDE explicitly in terms of the samples,

$$\hat{p}(x) = \frac{1}{nh} \sum_{i=1}^{n} \varphi\left(\frac{x - x_i}{h}\right),$$

we estimate the score as

$$\hat{s}(x) = \frac{\partial_x \hat{p}(x)}{\hat{p}(x)} = \frac{\sum_{i=1}^{n} \left[-\frac{x-x_i}{h}\,\varphi\left(\frac{x-x_i}{h}\right)\right]}{h^2 \sum_{i=1}^{n} \varphi\left(\frac{x-x_i}{h}\right)}.$$

2 Task

The goal of Flash-SD-KDE is to implement a fast, GPU-accelerated SD-KDE (with empirical score) on an NVIDIA RTX A6000 using Triton. We:

- implement a Gaussian KDE and KDE-based score estimator in Triton,
- use these to construct an empirical SD-KDE on the GPU, and
- benchmark against CPU Silverman KDE and scikit-learn's KDE over a range of sample sizes $(n_{\mathrm{train}}, n_{\mathrm{test}})$.

2.1 Hardware

All experiments run on a workstation equipped with an NVIDIA RTX A6000 (GA102). This GPU exposes 84 streaming multiprocessors (SMs); each SM contains 128 FP32 ALUs and 16 special function units (SFUs). Because the ratio of FP32 ALUs to SFUs is 128:16, we treat one exp (issued on an SFU) as costing the equivalent of $128/16 = 8$ FP32 flops in our models. The card delivers roughly 40 TFLOP/s of peak FP32 throughput and approximately 770 GB/s of GDDR6 bandwidth.

The host CPU is a dual-socket AMD EPYC 7763 system ("Milan") configured with 2×64 cores and two hardware threads per core (256 logical CPUs reported by `lscpu`). The processors boost up to 3.53 GHz, expose 512 MiB of shared L3 cache across the sockets, and provide modern ISA extensions (AVX/AVX2, FMA, BMI1/2, SHA, VAES, etc.). Unless otherwise noted, all CPU benchmarks in this report run on this platform.

3 High-dimensional SD-KDE and Tensor Cores

The 1-D SD-KDE formulation provides useful intuition, but our primary goal is to understand how the method scales in higher dimensions and how to exploit Tensor Cores. Let $x_i, y_j \in \mathbb{R}^d$ denote training and query points, with bandwidth h and squared Euclidean distance

$$\|x_i - y_j\|^2 = \|x_i\|^2 + \|y_j\|^2 - 2\, x_i^\top y_j.$$

Stacking training samples into $X \in \mathbb{R}^{n_{\text{train}} \times d}$ and queries into $Y \in \mathbb{R}^{n_{\text{test}} \times d}$, the pairwise dot-product matrix

$$G = XY^\top \in \mathbb{R}^{n_{\text{train}} \times n_{\text{test}}}$$

dominates the arithmetic once d is moderately large ($d \gtrsim 16$). On modern NVIDIA GPUs this GEMM can be mapped to Tensor Cores and evaluated at 5–10× the throughput of standard FP32 SIMT arithmetic, particularly when we restrict to d that are multiples of 16 and use Triton's `tl.dot` interface. The remaining operations—vector norms, broadcasted additions, and exponentials—are all $O(n_{\text{train}} n_{\text{test}})$ but quickly become a small fraction of the total FLOPs as d grows.

The empirical SD-KDE score inherits the same GEMM structure. Its numerator involves terms of the form

$$\sum_j -(x_i - y_j)\, \varphi_{ij}, \qquad \varphi_{ij} = \exp\!\left(-\frac{\|x_i - y_j\|^2}{2h^2}\right),$$

which naively suggests $O(n_{\text{train}} n_{\text{test}} d)$ additional elementwise arithmetic. Using the identity

$$\sum_j (x_i - y_j)\, \varphi_{ij} = x_i \sum_j \varphi_{ij} - \sum_j \varphi_{ij}\, y_j,$$

we can decompose the numerator into a second GEMM:

$$T = \Phi Y.$$

Thus both the KDE evaluation and the SD-KDE score numerator reduce to Tensor-Core-accelerable matrix multiplies, plus $O(n^2)$ scalar work for the norms and exponentials. In this note we focus on the $d = 16$ case, which aligns naturally with the Tensor Core tile sizes on the RTX A6000 and offers a clean contrast to the 1-D baseline (summarized in the appendix).

3.1 Arithmetic intensity in d dimensions

To understand when the high-dimensional SD-KDE becomes compute-bound, we estimate FLOPs, bytes moved, and arithmetic intensity for the d-dimensional case. Let $n_{\text{train}} = k$ and set $n_{\text{test}} = k/8$ as in the experiments.

Total FLOPs. The 16-D implementation consists of three main matrix-multiply stages:

1. Score Gram matrix $G = XX^\top$: $2dk^2$ FLOPs.
2. Score numerator $T = \Phi X$: $2dk^2$ FLOPs, plus $4k^2$ scalar FLOPs for norms and distance terms and $8k^2$ FLOPs for exponentials (counting each exp as 8 FLOPs due to the SFU/FP32 ratio).
3. Final KDE Gram matrix on debiased data: $2dk(k/8)$ FLOPs, plus $4k(k/8)$ scalar FLOPs and $8k(k/8)$ FLOPs for exponentials.

Aggregating these terms yields

$$\text{FLOPs}_d(k) \approx 4dk^2 + 12k^2 + 2d\frac{k^2}{8} + 12\frac{k^2}{8} = \Big(4d + 12 + \tfrac{d}{4} + \tfrac{3}{2}\Big)k^2.$$

Substituting $d = 16$ gives

$$\text{FLOPs}_{16}(k) \approx 81.5\, k^2,$$

which is on the order of 10^{11} FLOPs for $k = 32\text{k}$.

Bytes moved. In the idealized case where each sample and query is read once and each output is written once, the leading-order memory traffic is

$$\text{Bytes}_d(k) \approx 4\big(dk + d(k/8) + dk + k/8\big) = 4\Big(\tfrac{9}{8}dk + \tfrac{k}{8}\Big),$$

where we count reads of the training data X, debiased data, queries Y, and writes of the $k/8$ outputs, up to lower-order terms.

Arithmetic intensity. Dividing FLOPs by bytes gives

$$I_d(k) = \frac{\text{FLOPs}_d(k)}{\text{Bytes}_d(k)} \approx \frac{\big(4d + 12 + \frac{d}{4} + \frac{3}{2}\big)k^2}{4\big(\frac{9}{8}dk + \frac{k}{8}\big)} \sim C(d)\, k \quad \text{for large } k,$$

with

$$C(d) \approx \frac{4d + 12 + d/4 + 3/2}{4 \cdot (9d/8)} = \frac{(17/4)\, d + 27/2}{9d/2}.$$

For $d = 16$ this simplifies to

$$C(16) \approx 2.0, \qquad I_{16}(k) \approx 2.0\, k \text{ flops/byte}.$$

Thus, as in the 1-D setting, the arithmetic intensity grows linearly with problem size; even moderate k values place the 16-D SD-KDE well into the compute-bound regime on the RTX A6000.

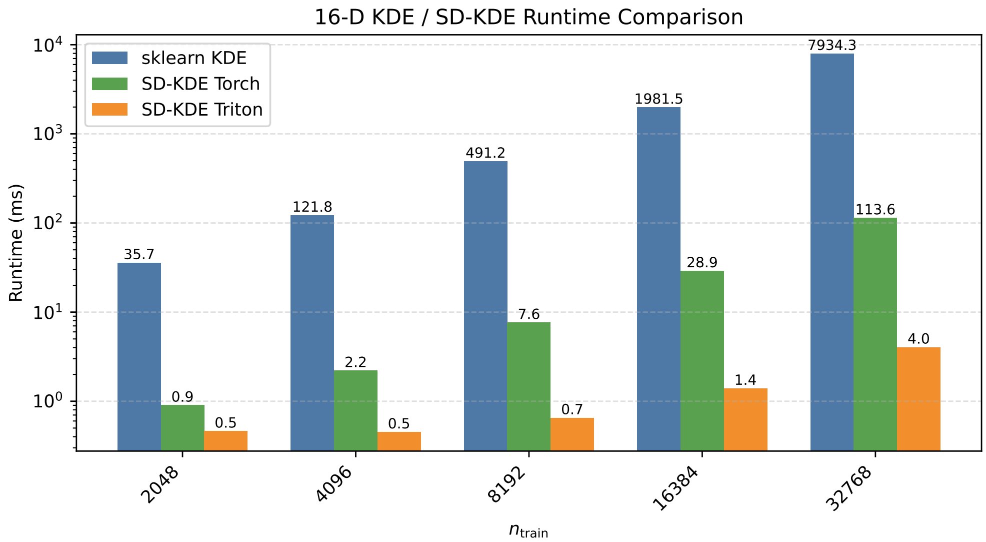

Figure 1: Runtime comparison for 16-D KDE/SD-KDE across n_{train} up to 32,768 ($n_{\text{test}} = n_{\text{train}}/8$). The Triton SD-KDE enjoys both the Tensor-Core acceleration (relative to sklearn) and the GEMM-based score computation (relative to the Torch baseline).

4 16-D experiments

We now summarize the empirical behavior of the 16-D implementation on the RTX A6000. For each n_{train} in $\{4\text{k}, 8\text{k}, 16\text{k}, 32\text{k}\}$ we set $n_{\text{test}} = n_{\text{train}}/8$, draw data from a simple 16-D Gaussian mixture, and run three baselines: scikit-learn KDE, Torch SD-KDE (GEMM-based), and Triton SD-KDE (Tensor-Core GEMMs for both KDE and score). Figure 1 shows that the Triton SD-KDE rapidly outpaces both baselines as n grows, while remaining numerically close to the Torch SD-KDE reference.

We also measure utilization by combining the flop model above with the measured runtimes. As shown in Figure 2, the Torch SD-KDE baseline reaches only a few percent of the A6000 Tensor Core peak, whereas the Triton implementation climbs into the multi-digit range once n_{train} exceeds 8k. This confirms that the 16-D Tensor-Core formulation is firmly compute-bound and that additional tuning effort should focus on kernel fusion and occupancy rather than memory traffic.

5 Performance tuning

We performed a sweep over the Triton launch parameters to improve utilization for the $n_{\text{train}} = 32\text{k}$ case (with $n_{\text{test}} = 4\text{k}$). Specifically we varied:

- $BLOCK\_M \in \{32, 64, 128, 256\}$,
- $BLOCK\_N \in \{32, 64, 128, 256\}$,
- $num\_warps \in \{1, 2, 4, 8\}$,
- $num\_stages \in \{1, 2, 4\}$,

and selected the combination that minimized runtime. For this workload we found that $BLOCK\_M = 64$, $BLOCK\_N = 128$, $num\_warps = 1$, and $num\_stages = 2$ gave the best overall performance,

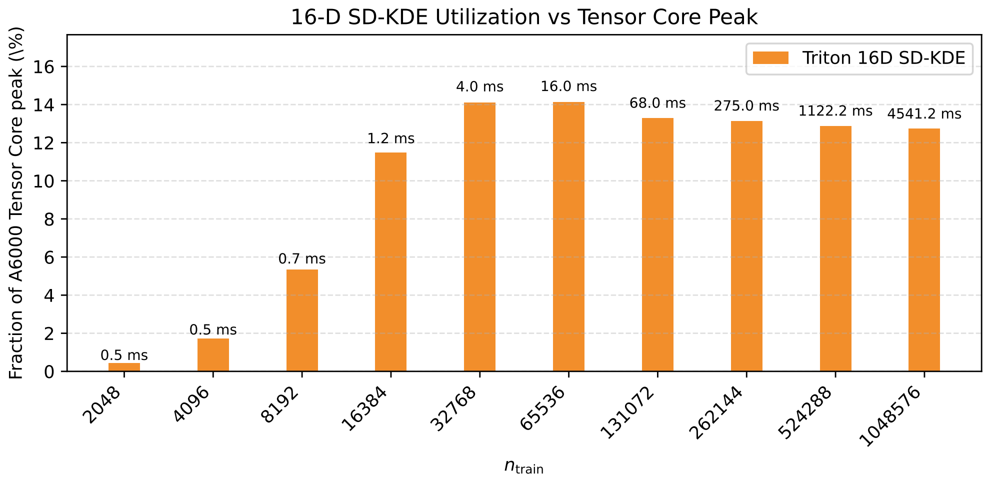

Figure 2: Utilization (percentage of RTX A6000 Tensor Core peak, taken as 155 TFLOP/s FP32-equivalent) for the 16-D SD-KDE pipeline, using the flop estimate $4dn^2+(4+8)n^2+2dmn+(4+8)mn$ with $d = 16$ and $m = n/8$. Bars are annotated with the observed runtime (ms).

increasing measured FLOP utilization by more than $2\times$ relative to the initial settings. The same sweep can be repeated for different problem sizes if further tuning is needed.

A 1-D SD-KDE baseline

For completeness we briefly summarize the 1-D SD-KDE arithmetic intensity and empirical behavior. With $n_{\text{train}} = k$ and $n_{\text{test}} = k/8$, there are two steps:

1. **Score + shift:** For each training point we compute $\hat{s}(x_i)$ from all k points and then form $x_i^{\text{SD}} = x_i + \frac{h^2}{2}\hat{s}(x_i)$. This uses $O(k^2)$ pairwise kernel interactions. With the RTX A6000 hardware ratio (128 FP32 ALUs, 16 SFUs per SM) we budget an exp as 8 flop-equivalents. The score accumulation requires one exp and roughly eight additional arithmetic ops (subtraction, scaling, accumulation), yielding $c_1 \approx 16$ flops per (train, train) pair.

2. **KDE on debiased samples:** We then evaluate a standard Gaussian KDE at $k/8$ query points using the k debiased samples, which uses $O(k^2/8)$ interactions. Each pair requires one exp (8 flops) plus about six other operations (difference, square, scaling, accumulation), so we take $c_2 \approx 14$ flops per (train, test) pair.

The total work is therefore approximated by

$$\text{FLOPs}(k) \;\approx\; c_1 k^2 + c_2\, k\,(k/8) \;\approx\; 16k^2 + 14\frac{k^2}{8} \;=\; 17.75\, k^2.$$

For $k = 32\text{k}$ this is on the order of 2×10^{10} flops. When we fix $n_{\text{test}} = k/8$ we move approximately $5k$ bytes (counting one read of each train and test point and one write of each output), so the arithmetic intensity scales as

$$I(k) = \frac{\text{FLOPs}(k)}{\text{Bytes}(k)} \;\approx\; \frac{17.75\, k^2}{5\, k} \approx 3.55\, k \quad \text{flops/byte},$$

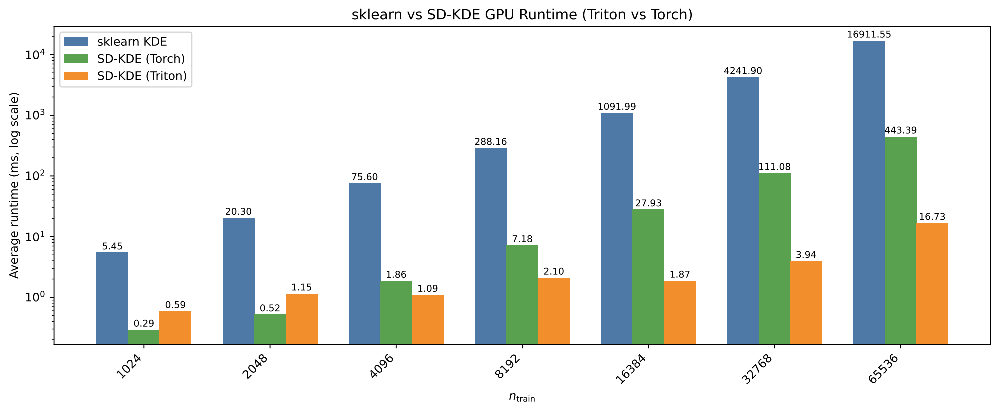

Figure 3: Average runtime (log-scale y-axis) of scikit-learn KDE and SD-KDE GPU across n_{train} in 1-D; annotations show the SD-KDE GPU speedup relative to sklearn.

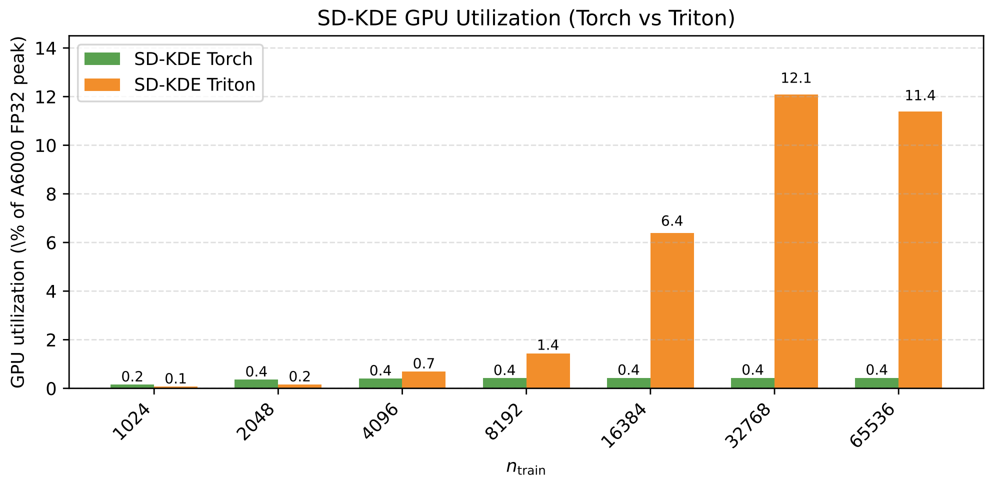

Figure 4: Estimated GPU utilization (as a percentage of A6000 FP32 peak) for 1-D SD-KDE implemented in Triton and optimized Torch, computed from the 1-D flop model and the measured runtimes.

placing realistic problem sizes firmly in the compute-bound regime.

Empirically we sweep k over powers of two from 512 to 32k, with $n_{\text{test}} = k/8$, and average over three seeds. For each configuration we record CPU Silverman KDE time, scikit-learn Gaussian KDE time, and SD-KDE GPU time. Across the entire range, the empirical SD-KDE GPU implementation is consistently faster than scikit-learn, with speedups growing with k; at the largest sizes the GPU achieves well over an order-of-magnitude speedup relative to the sklearn baseline.

We also run a Triton-only sweep over larger 1-D problem sizes, using powers of two for n_{train} up to $2^{22} \approx 4.2$ million (and $n_{\text{test}} = n_{\text{train}}/8$) with a single seed. The helper script `run_triton_scaling.sh` executes this experiment by invoking the `--emp-kernel-only` benchmark mode, producing a compact log that we feed to Nsight Systems when examining the multi-million-sample kernels.

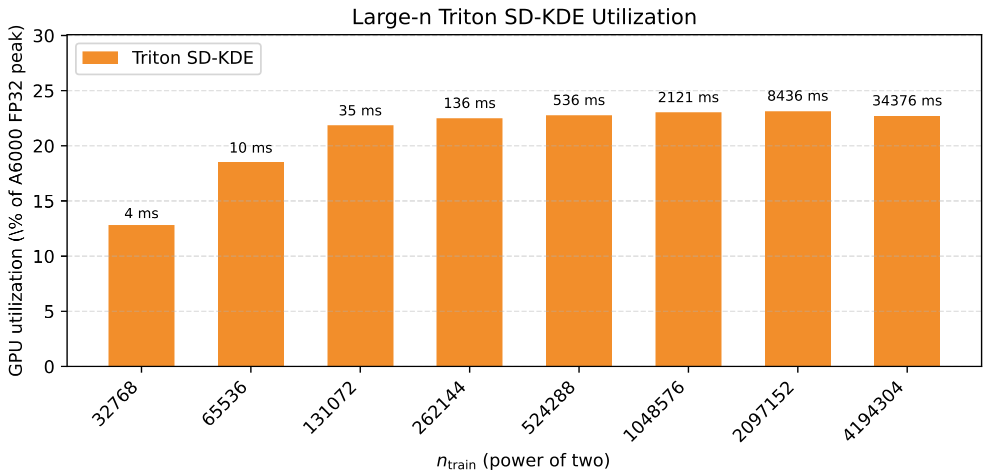

Figure 5: Utilization of the 1-D Triton SD-KDE kernel for large n_{train} (powers of two up to 2^{22}). Labels show the observed runtime for each data size; error bars are omitted because a single seed is used per point.

B Additional commands

For convenience we summarize the most frequently used scripts:

- `run_triton_scaling.sh`: sweeps 1-D SD-KDE with `--emp-kernel-only` for Nsight profiling.
- `run_triton_sd_kde_nd.sh`: runs Triton-only 16-D SD-KDE sweeps and logs runtimes for utilization plots.
- `run_nd_runtime_sweep.sh`: collects runtime data (sklearn vs Torch vs Triton) for 16-D KDE/SD-KDE up to $n = 32{,}768$.

References

[1] Elliot L. Epstein, Rajat Vadiraj Dwaraknath, Thanawat Sornwanee, John Winnicki, and Jerry Weihong Liu. SD-KDE: Score-debiased kernel density estimation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.