We thank the reviewers for their detailed reports and suggestions. We now provide answers to their main questions.

"usability" (R1) As showcased in our examples, users only have to cast their tensors as symbolic matrices: this is a one-line operation, just like with sparse matrices. We stress that our library runs out-of-the-box on *e.g.* a fresh `Google Colab` session: a simple call to "`pip install *******`" is all it takes to get started.

"only useful for distance computation", "limited applicability" (R2) We respectfully disagree. While fast distance computation is core in many geometric ML methods, we show multiple examples that go beyond this first problem. Our library has gained a sizeable and diverse userbase in fields ranging from theoretical ML to medical imaging and quantum chemistry. It has been downloaded over 25k times and is now mature to become a standard toolbox for ML.

"improvements only in a limited range" (R2) Our library targets computations on data samples that are made up of 10^3 to 10^6 points in dimension 1 to 100. This is relevant to problems such as geometric deep learning, 3D vision, shape analysis and optimal transport theory. In data sciences, we provide a sizeable performance boost to methods that are ubiquitous in applied ML and are often used through the `Scikit-learn` library. As pointed out by R1, R3 and R4, our efficient support for these problems is bound to have a stimulating impact on ML research. Going further, we would love to provide optimal performance in all settings – from low-resolution 3D shapes to Google-scale datasets – but note that the rigid structure of `CUDA` registers makes it a very challenging problem for generic symbolic computations.

"performance gap" (R1), "similarities and differences with deep learning compilers" (R4) Let us benchmark a matrix-vector product with an $\mathrm{N} \times \mathrm{N}$ Gaussian kernel matrix $k(x_i, x_j) = \exp(-\|x_i - x_j\|^2)$ for a point cloud $x_1, \dots, x_\mathrm{N}$ in $\mathbb{R}^3$. `Halide` and `TVM` implement the same streaming computations as our library, whereas `PyTorch` and `TF-XLA` attempt to optimize a tensorized code – as detailed in Table 5 of the supplementary materials, Python tiling with `PyTorch` or `XLA` is also inefficient. Our library is extremely competitive for kernel- and distance-related operations (timings performed on a `Google Colab` session with a K80 GPU, checked with `nvidia-smi`).

	PyTorch	PyTorch-TPU	TF-XLA	Halide	TVM	**Ours**
$\mathrm{N} = 10\mathrm{k}$	34 ms	10 ms	23 ms	5 ms	6 ms	**2 ms**
$\mathrm{N} = 100\mathrm{k}$	mem	mem	1,062 ms	360 ms	282 ms	**107 ms**
$\mathrm{N} = 1\mathrm{M}$	mem	mem	mem	41.3 s	26.5 s	**10.3 s**
Lines of code / interface	5 / arrays	5 / arrays	5 / arrays	15 / C++	17 / low-level	**5 / arrays**

"why modern toolboxes are not meant for geometric problems?" (R2) The example above provides an insight into this question. Roughly speaking, frameworks like `TVM` or `Halide` demand technical skills to be used efficiently, whereas `PyTorch` or `TF-XLA` do not prioritize the geometric computations of Section 2: these require significant investments on specific `CUDA` schemes to avoid memory issues and accelerate computations when the sample size $N > 10^4$.

"comparison to DGL" (R4) will be included in the final version.

"blocking on CPU" (R1) On CPU, our library implements the simple reduction scheme of Figure 2.a, with a direct parallelization via OpenMP. We are are working on explicit SIMD support for future releases. For approximate computations, our block-sparse scheme is available on both CPU and GPU backends.

"concerns regarding Table 2" (R1) All our benchmarks are done fairly. We chose FAISS since it is well established, performs well in the reference "ANN-Benchmarks" (some competitors such as ScaNN have been added after the NeurIPS deadline) and is one of the only NN-search libraries with an optimized GPU implementation. It comes with two main backends: `FAISS-GPU`, a bruteforce GPU search similar to ours (useful when $\mathrm{N} \leqslant 10^6$) and `FAISS-HNSW`, a CPU implementation of the graph-based `HNSW` algorithm, with a significant pre-processing time (useful when $\mathrm{N} > 10^6$). We never intended to compete with approximate, graph-based methods for NN-search on very large datasets and made this clear in our paper (see Sections 2, 4, 5.2 and the legend of Table 2, where we also state that `FAISS-HNSW` is run with a recall at 90%). We will stress this more in the final version.

"why FAISS runs out of memory at 10M points?" (R1) The backend that fails at 10M points is `FAISS-GPU` which is a bruteforce implementation, not an ANN method. We do not know exactly why it fails.

"compilation aspects" (R1), "implementation weakly described" (R2), "specific binary is compiled" (R2). Our engine handles all reductions of symbolic matrices through the same parallel schemes: `C++` code is generated for each formula $F(x_i, y_j)$ and is inserted in a templated `CUDA` kernel that implements the reduction scheme of Figure 2.b. As detailed in Section 4, compilation overheads are not a bottleneck in practice: binaries are stored on the hard drive for later use. Our engine is implemented in `C++`, which is key to performance: array-centric libraries prevent us from managing `CUDA` registers. Block-sparse reductions are also efficient: runtimes are proportional to the sparsity of the block-wise reduction mask, provided that the dimensions of the tiles exceed the `CUDA` block size (~ 100). Implementation is fully described on our website and will be included in the final version / supplementary materials.