SICK BIDE: SOFTMAX IMPLICIT COMPUTE KERNEL ALLOWS BINARY IMPLICIT DISTRIBUTION ENCODING

Théo Boyer
Duon Labs
theo@duonlabs.com

Santiago Malter-Terrada
Duon Labs
santi@duonlabs.com

ABSTRACT

Deep learning has achieved remarkable progress by minimizing domain-specific assumptions in favor of data-driven learning. In this paper, we introduce the Binary Implicit Distribution Encoding (BIDE), a novel framework for assumption-free distribution modeling that unifies continuous, categorical, and ordered distributions. By leveraging binary representations, BIDE implicitly encodes probability distributions without relying on predefined structures.

Complementing BIDE is the Softmax Integral Compute Kernel (SICK), a GPU kernel for efficiently computing softmax normalization over combinatorial binary spaces, addressing the computational challenges of this approach. Together, BIDE and SICK form a scalable, memory-efficient, and expressive alternative for generative modeling, probabilistic inference, and large language models (LLMs).

Experimental results demonstrate BIDE's ability to accurately model complex distributions without prior knowledge, highlighting its potential to redefine distribution modeling. This framework opens new avenues for generative AI, probabilistic forecasting, and other applications requiring flexible and assumption-free distribution representations.

***K*eywords** Probabilistic Modeling · Implicit Representations · GPU kernel

1 Introduction

Modeling distributions is a cornerstone of machine learning, especially in the deep learning based generative era with countless applications as diverse as natural language processing (NLP), Image / Video generation or time series. While categorical distributions can be trivially addressed, continuous distributions are not as trivial. Traditionally, two classes of approaches dominate continuous distribution modeling: parametric modeling, which requires *a priori* knowledge about distributions family, and discretization, which maps continuous spaces into discrete ones. While these methods have proven effective in many contexts, they are often constrained by their inherent trade-offs in flexibility, expressiveness, and computational efficiency. Furthermore, deep learning has consistently improved as the methods were more general and carried less assumptions about the problems it solves ([1], [2])

Binary Implicit Distribution Encoding (BIDE) introduces a paradigm shift in how we approach distribution modeling. By directly leveraging binary representations of data, BIDE provides a unified framework to model continuous, categorical, and ordered distributions. Instead of relying on approximations or domain-specific assumptions, BIDE maps binary inputs to their likelihood logits using a lightweight neural network architecture, allowing for expressive and memory-efficient implicit distribution representations.

Complementing BIDE is the Softmax Integral Compute Kernel (SICK), a GPU-optimized kernel designed to compute the softmax denominator and gradients required for BIDE's operation. In the same spirit as FlashAttention[3] SICK's method of denominator computation ensures exact computation without materializing the full intermediate representations. Without this kernel, materializing the 2^n binary combinations and their intermediate computations would not be feasible because of both compute and memory bottlenecks, especially for high values of n.

This paper explores the design, implementation, and implications of the BIDE and SICK framework. We focus on the key use case of modeling continuous distributions without assumptions in a way that combines the expressiveness of categorical approaches with the flexibility of continuous methods. We demonstrate BIDE's ability to correctly approximate continuous distributions and SICK's significant improvements in memory efficiency and computational throughput.

To evaluate the viability of this framework, we present a thorough theoretical analysis of the computational and memory requirements of SICK under two variants. This is followed by experimental results of our implementation of SICK, showcasing the effectiveness of the 2 declinations. The findings underline the potential of BIDE and SICK as a transformative tool for distribution modeling, with applications extending to language modeling, probabilistic inference, and beyond.

This work contributes:

- A novel approach to distribution modeling: BIDE leverages binary representations to unify and extend traditional modeling paradigms.
- GPU-optimized computation: SICK efficiently handles normalization and gradient calculations.
- Insights into scalability and practicality: Through theoretical analysis and experimental validation, we highlight the trade-offs and potential optimizations of this framework.

The remainder of the paper is structured as follows. Section 2 provides background on distribution modeling and GPU optimizations. Section 3 details the methodology behind BIDE and SICK, including their architectural design and mathematical underpinnings. Section 4 presents experimental results and applications. Section 6 discusses the implications and limitations of the framework, and Section 7 concludes with future directions.

2 Background

2.1 Distribution Modeling

Modeling data distributions is a cornerstone of statistical and machine learning methods. Continuous distributions are often approximated using parametric models, such as Gaussian distributions or mixture of distributions, or discretized to apply categorical methods. While these approaches are computationally cheap, they both impose limitations on the flexibility and expressiveness of the modeled distributions. As a consequence their predictive performances directly depend on the correctness of the hypotheses made by the model developers which prevents any form of assumption-free prediction.

Recent advances in neural networks such as normalizing flows [4] have extended the ability to model complex, high-dimensional distributions. However, these approaches often come with trade-offs like memory and computational overhead, or constraints on the model architecture. Furthermore they are not completely assumption-free.

2.2 Challenges in Categorical Distribution Modeling with neural networks

Predicting a distribution over a categorical set using a neural network is a trivial problem as one just needs to apply a layer with its output dimensions being equal to the size of the set. While this is a viable approach for reasonable set sizes, it has two significant drawbacks when scaling to big numbers:

- The memory consumption of the weight of the output scales linearly with the set size. This directly impact Large models that also have a large hidden dimension, leading to huge weights of $O(ND)$. And it also impacts small models that target a big set, as the computation of the output layer dominates the ressources consumption of the total forward pass of the model.
- The states that are rare in the training set can cause problems as their associated weights are pushed towards zero [5]. A way to mitigate this problem would be to leverage structure between states for extrapolation.

The LLMs vocab size being constently growing, even reaching $256K$ for Gemma [6] we believe that any alternative methods that describe categorical distributions over big sets could be relevant in the future.

2.3 GPU Optimizations in Neural Computation

GPUs have become essential for accelerating deep learning workloads, offering massive parallelism and high memory bandwidth. Modern GPU programming frameworks, such as Triton and CUDA, enable fine-grained control over kernel-level operations. Leveraging these frameworks, researchers have developed optimized implementations for key operations like matrix multiplications, activation functions, and attention layer [3].

Using a vanilla deep learning framework like Pytorch [7] allow researchers and developers to utilize quite well the GPUs. However applying sequentially one kernel per operation is often suboptimal because of repeated memory transfers. Some features such as compilation allow for automated kernel fusion, increasing the GPU effectiveness. However compilers fail to optimize complex operations that require a mathematical rewrite of the formulas. As a consequence neural network layer kernels are still an active area of research [3] [8][9][10]. This is also why intermediary DSL like triton [11] were created to simplify the process of writing custom GPU kernels for complex neural network layers.

2.4 Binary Implicit Distribution Encoding and Softmax Integral Compute Kernel

The Binary Implicit Distribution Encoding (BIDE) introduces a novel neural network layer that models distributions directly from binary representations of data. Unlike traditional embedding methods, BIDE can efficiently map inputs to logits by leveraging their binary representations, and making it well-suited for applications like continuous distribution modeling and memory-efficient vocabulary distributions in LLMs.

The Softmax Integral Compute Kernel (SICK) complements BIDE by providing GPU-optimized implementations for forward and backward computations. SICK leverages GPU kernels to compute the deniminator for the softmax operation with exact computation. This is achieved by exploiting GPU parallelism, binary operations, memory-aware block-wise computation, tiling and the online-softmax algorithm [12].

Together, BIDE and SICK address the limitations of existing methods by combining the expressiveness of categorical distributions with the precision of continuous models, all while optimizing for modern GPU hardware.

3 Methodology

3.1 BIDE: Binary Implicit Distribution Encoding

Binary Implicit Distribution Encoding (BIDE) is a novel neural network layer designed to model distributions over arbitrary inputs. The layer takes as input a binary representation of any data types, such as numbers (e.g., float16) or categories (e.g., uint16), and predicts their likelihood logits. This binary representation enables BIDE to generalize across data types with no modifications while maintaining computational efficiency.

3.1.1 Architecture

The BIDE layer is 1 hidden layer MLP. The input value is converted to its binary representation and mapped to -1 for zeros and 1 for ones. Afterwards, we apply the input layer of the MLP with a gemm on the input vector followed by a ReLU. Finally, the output layer is applied again with a gemm outputing one value as the logit.

3.1.2 Applications

- Continuous distributions: Unlike traditional methods that discretize continuous spaces or rely on parametric distributions, BIDE offers a flexible framework that combines expressiveness with computational tractability.
- Memory-efficient vocabulary distributions: BIDE can replace conventional embeddings in large language models (LLMs), reducing memory overhead without compromising performance.
- Tokenizer-free LLMs: Some tokenizer-free methods already exists by directly mapping input bytes to output bytes [13][14] and lead to slower inference as more bytes are needed to express the equivalent of a token. BIDE would allow a native way to combine bytes together without architectural wrappers, either replacing the need for byte patching or being used on top of it.
- Probabilistic Foundation model for time series: The assumption-free approach that SICK BIDE provides allow the creation of a new class of foundation models for time series based on probabilistic modeling and going beyond single point forecasts [15] with a-posteriori uncertainty estimation methods like conformal prediciton [16] and allow for a better expressivity than probabilistic models based on mixtures of distributions [17]

3.2 SICK: Softmax Integral Compute Kernel

Softmax Integral Compute Kernel (SICK) is a GPU-optimized component that computes the normalization constants required for BIDE. Naïve numerically-stable softmax computation involves multiple passes over the logits, which can be computationally expensive when dealing with the 2^n binary combinations of n-bit inputs. But it's possible to compute it using the online softmax algorithm [12], with only one pass, and without the need to materialize the entire logits vector in memory.

3.2.1 Forward Computation

SICK performs computations of the softmax normalization constant by combining multiples tricks:

- Mixed precision: Some computations such as the gemm can be made in lower precisions with minimal loss of precision
- Fused operation: Not materializing intermediates tensors in memory to avoid the peak memory bottleneck.
- On-device input generation (brute-force variant): The binary nature of BIDE's inputs allows for generation of inputs on-device instead of generating the inputs on the host and transfer them. The memory transfer economy is significant as it's often the bottleneck of modern workloads.
- Pre-computation trick (precomputed variant): It's possible to rewrite the formula of z to turn the workload from the compute bounded regime to the memory bounded one. More details in 3.4

3.3 Fusion of SICK and cross-entropy loss

3.3.1 Forward Computation

For the forward pass of the fused cross entropy loss, we simply evaluate BIDE with the labels y and use SICK to compute the normalization constant. There is no need to develop a edicated kernel for this operation.

3.3.2 Backward Computation

The backward pass of BIDE is a classical MLP backward pass, and the main difficulty is the same as SICK's forward computation, which is iterating over the full combinatorial input space. We show in appendix C that it's possible to compute the most part of the gradient in the forward pass, with the formulas becoming:

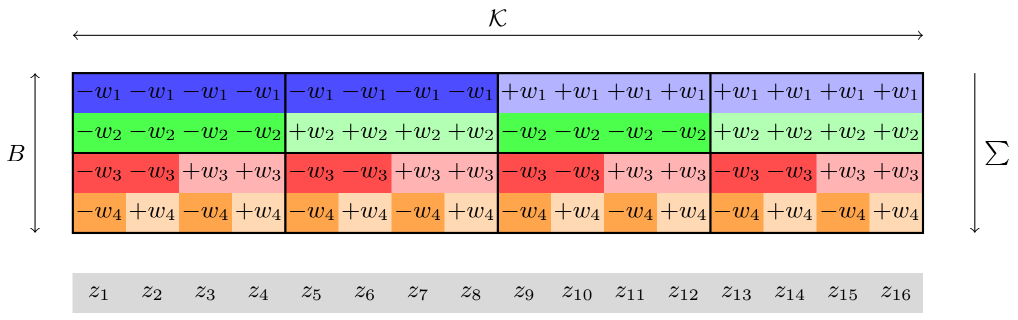
$$\frac{\partial \mathcal{L}}{\partial r_i} = \frac{1}{s}\sum_{k=1}^{K} h_{ki}\exp(l_k - m) - \frac{1}{N}\sum_{n=1}^{N} h_{y_n i} \tag{1}$$

$$\frac{\partial \mathcal{L}}{\partial W_{ij}} = \frac{1}{s}\sum_{k=1}^{K} b_{kj}\mathbb{1}_{h_{ki}>0} r_i \exp(l_k - m) - \frac{1}{N}\sum_{n=1}^{N} b_{y_n j}\mathbb{1}_{h_{y_n i}>0} r_i \tag{2}$$

Note that the loop over K can be done in the forward pass as it doesn't depend on y.

3.4 Hierarchical Pre-Computation for Efficiency

We present the rewrite of the z formula that allows to significantly reduce the compute capacity needed to achieve our objective. Let's isolate one neuron of the hidden layer whoes pre-activation value z is computed as such:



$-w_1$	$-w_1$	$-w_1$	$-w_1$	$-w_1$	$-w_1$	$-w_1$	$-w_1$	$+w_1$	$+w_1$	$+w_1$	$+w_1$	$+w_1$	$+w_1$	$+w_1$	$+w_1$
$-w_2$	$-w_2$	$-w_2$	$-w_2$	$+w_2$	$+w_2$	$+w_2$	$+w_2$	$-w_2$	$-w_2$	$-w_2$	$-w_2$	$+w_2$	$+w_2$	$+w_2$	$+w_2$
$-w_3$	$-w_3$	$+w_3$	$+w_3$	$-w_3$	$-w_3$	$+w_3$	$+w_3$	$-w_3$	$-w_3$	$+w_3$	$+w_3$	$-w_3$	$-w_3$	$+w_3$	$+w_3$
$-w_4$	$+w_4$	$-w_4$	$+w_4$	$-w_4$	$+w_4$	$-w_4$	$+w_4$	$-w_4$	$+w_4$	$-w_4$	$+w_4$	$-w_4$	$+w_4$	$-w_4$	$+w_4$
z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9	z_{10}	z_{11}	z_{12}	z_{13}	z_{14}	z_{15}	z_{16}

Figure 1: Computation of z in BIDE

In figure 1 we decompose the computation of $z = bW^T$ into the multiply and sum reduction steps. Elementwise multiplication of W and b effectively result in taking the weights w_i when $b_i = 1$ and $-w_1$ when $b_i = 0$. We can see the structure in the computations of z_i visually once we pack the matrices in blocks.

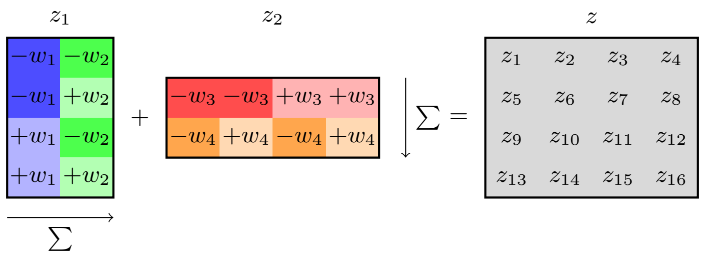


Figure 2: Computation of z in BIDE

We can thus take advantage of this structure and isolate two blocks z_1 and z_2 such that adding all pairs of entries of z_1 and z_2 give back z the pre-activation value of the neuron. Note that multiple choices of z_1 and z_2 are possible depending on the location where you split W.

4 Experimental Results

This section presents an evaluation of the SICK BIDE framework across various scenarios using our implementation of SICK. We first examine performance metrics for the different variants, followed by an analysis of SICK's applicability to the modeling of continuous distributions. The results are summarized in tables and figures.

4.1 Experimental Setup

To evaluate the performance of SICK BIDE, we measured the execution time for two isolated settings. The first setting is a batch size of 64 and the second is a simulation of the final layer of a transformer-based model with a batch size of 32, a context size of 512, and a hidden dimension of 32. The experiments were conducted on an NVIDIA GeForce RTX 4060 Laptop GPU, which provides:

- 11.61 TFLOPS of peak compute performance.
- 256.0 GB/s of memory bandwidth.

We compared three variants in this setting:

- Torch implementation with automated kernel fusion: An implementation of BIDE without using any manually crafted kernel. We use torch.compile whenever we can for automated kernel fusion.
- Bruteforce Variant: Direct computation of the integral kernel without optimizations.
- Pre-computated Variant: Leveraging pre-computed intermediate values described in 3.4 to reduce computational overhead.

The results focus on kernel execution time, throughput, effective FLOPs, and memory bandwidth utilization.

Finally we tested sick in a practical setting, by learning an embedding for 4 distributions at the same time:

- A Normal distribution with $\mu = 3, \sigma = 1$
- A Gamma distribution with $\alpha = 2, \beta = 1$
- A Mixture of two normal distributions with $p = 0.2, \mu_1 = 120, \sigma_1 = 1, \mu_2 = 132, \sigma_2 = 1.5$
- a Pareto distribution with $x = 1, k = 1$

We run the training for 2000 steps with batch size 64 and using mixed precision. This final experiment showcase the modeling power of BIDE while providing an good end to end test for the SICK kernel.

4.2 Performance Metrics

4.2.1 Variant results

The results highlight a spectrum of computational trade-offs across implementations for all operations in Table 1. The "torch.compile" implementation, while straightforward, is lacking memory for larger batch sizes due to its reliance

Operation	**Batch size**	**torch.compile**		**Bruteforce**		**Precomputed**	
		fp32	**fp16**	**fp32**	**fp16**	**fp32**	**fp16**
Normalisation constant	64	11.71	7.09	0.86	0.69	0.44	**0.36**
	16384	OOM	OOM	201.54	145.61	93.53	**63.29**
Fused cross-entropy forward	64	11.72	6.59	0.88	0.62	0.45	**0.34**
	16384	OOM	OOM	199.23	148.43	91.57	**63.48**
Fused cross-entropy backward	64	-	-	**0.90**	0.95	1.18	1.14
	16384	OOM	OOM	235.13	235.06	**227.38**	228.73

Table 1: Compute time for: vanilla PyTorch implementation, bruteforce, and precomputed variants of the SICK kernel

on materializing full intermediate results. For all kernels except the backward pass we observe a significant reduce in runtime when comuting in float16 compared to float32. The precomputed variant wins in all tested cases.

In contrast, the results for the backward kernel are more mitigated. The bruteforce variant wins for the small batch sizes while the precomputed variant is prefered for heavy batch sizes but with a small margin. Furthermore we observe that the float16 computation doesn't improve significantly the kernel runtime, even degrading it sometimes.

4.2.2 Implementation efficiency

Variant	**Execution Time (ms)**	**Throughput (Hz)**	**FLOPs (Tops/s)**		**Memory Bandwidth (Gb/s)**	
			Value	**Peak FLOPS (%)**	**Value**	**Peak Mem BW (%)**
Bruteforce	145.61	6.87	8.29	71.39	0.12	0.05
Precomputed	63.29	15.80	2.38	20.46	8.50	3.32

Table 2: Performance comparison for bruteforce and precomputed variants in fp16 with batch size 16384.

Table 2 provides an analysis of memory and compute requirements for the integral kernel. We infer the flops and memory bandwidth based on our model described in appendix B and compare them to the theoretical figures of the GPU. As expected, the precomputed variant utilizes less computational power but demanded more memory bandwidth compared to the bruteforce implementation.

Despite leveraging more than 70% of the GPU's theoretical max computing power the bruteforce variant is more than 2x slower than the precomputed one.

While it is clear that the bruteforce kernel is saturated by heavy computing pipelines utilization, we note that out implementation of the precomputed kernel struggle to fully utilize the potential of the GPU.

4.3 Modeling Continuous Distributions

We also evaluated SICK on a practical example end to end to assess for its viability

Method	**Runtime (s)**	**Train Loss (EMA)**	**Peak Memory (MB)**
Vanilla	37.38	7.227 091	418.42
Bruteforce	6.88	7.256 562	163.43
Precompute	7.62	7.195 776	163.43

Table 3: Comparison of runtime, loss (EMA), and peak memory usage for Vanilla, Bruteforce, and Precompute methods.

The vanilla method, while straightforward, suffered from excessive runtime and memory overhead. In contrast, both the bruteforce and precomputed variants significantly reduced both memory usage and runtime, demonstrating the efficiency of SICK BIDE while reaching a comparable train loss.

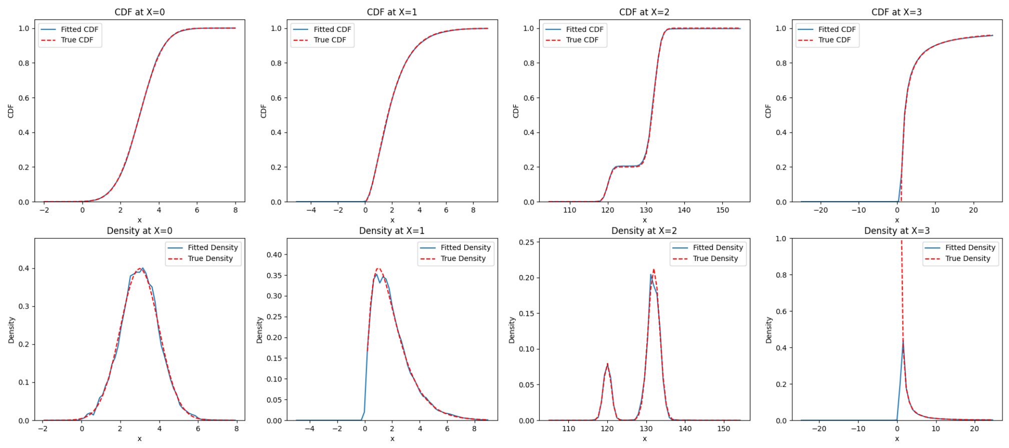


Figure 3: Continuous distribution estimations using SICK BIDE.

Figure 3 shows the obtained cdfs and density functions for the precompute variant compared to the true cdfs and densities. We observe that the CDFs match closely their targets. The Pareto CDF is a bit off for values close to one. For the densities however we can note some disparities:

- For the Normal distribution we see some perturbations around zero (probably because the concentration of float16 numbers is higher at this spot)
- For the Gamma distribution the tip is a bit off and some of the probability mass ends up in negative numbers which is theoretically impossible for a gamma distribution but can be attributed to approximations artifacts.
- For the Mixture of normales the tip of the stronger normal component is a bit off
- For the Pareto distribution we observe that what should be a density going to infinity the closest it get to zero at some point go back to zero in practice. This is to be expected as BIDE's density function is defined over the full float16 set and thus include impossible values such as negative ones for which the theoretical density should be zero.

4.4 Insights

- Trade-Off Analysis: The precomputed variant manage to significantly reduce the runtime of the kernel. The bruteforce variant achieved an honorable runtime by extracting most of the GPU's computing capacity.
- Practical Applicability: SICK BIDE effectively handled continuous distributions and approximated well the theoretical densities
- Integration Potential: The framework's GPU-optimized nature makes it well-suited for integration into modern AI systems.

In summary, SICK BIDE demonstrates strong potential as a scalable and efficient solution for complex distribution modeling. Future optimizations in hardware, algorithm design, and implementations will further enhance its viability across a broader range of applications.

5 Discussion

The results and analyses presented in this paper highlight the practical and theoretical potential of the SICK BIDE framework for distribution modeling. Below, we discuss key insights, trade-offs, limitations, and avenues for future exploration.

5.1 Limitations of Current Optimizations

Despite the gains achieved through pre-computation, several limitations remain:

- Implementation: Our implementation utilizes triton which is a DSL and still under development. We know from our theoretical analysis that the overhead of our implementation prevent the kernel to fully utilize all the performances of the GPU. Maybe a better triton implementation or even a CUDA implementation could reach better computing resources utilization.
- Modeling precision: Our experiments showed that BIDE can match quite well the theoretical distributions, but not perfectly. There are probably other implicit representations that could better model the theoretical distributions and match the performances of non assumption-free methods.
- Number of bits: The current implementation of BIDE allow to represent distributions over 16bits numbers in a LLM-like setting (reasonable batch size * context size). However to be able to fully represent UTF-8 encoded text, we need to upgrade BIDE to 64bits numbers. This is out of the reach of the current state of development of BIDE but could change in the future with both software and hardware innovations.

5.2 Opportunities for Future Optimizations

To address these limitations, several strategies warrant exploration:

- Hierarchical Decomposition: Other hierarchical decompositions methods could improve scalability, leveraging effectively patterns in binary combinations inputs.
- Lower-Precision Computing: Integrating low-precision approaches could balance memory and computational efficiency by leveraging modern low-precision hardware features, viabilizing the approach for BIDE with higher number of bits.
- Moore's law: As the GPUs become more performant with time, the SICK BIDE framework designed to maximaly use computing ressources will directly benefit from hardware advances and become naturally viable for higher numbers of bits as time passes.

6 Conclusion and Future Work

In this paper, we introduced Binary Implicit Distribution Encoding (BIDE) and the Softmax Integral Compute Kernel (SICK) as a novel paradigm for assumption-free distribution modeling by directly leveraging binary representations. By focusing on binary encoding, BIDE enables the modeling of both categorical and continuous distributions with high expressiveness and efficiency. SICK, a GPU-optimized kernel, complements this framework by making the computation of the softmax normalization constants tractible, avoiding approximations.

Our theoretical and experimental analysis demonstrated the feasibility of integrating SICK BIDE into modern machine learning pipelines, to efficiently model conditional distributions over vocabulary or real numbers. We provided a comprehensive breakdown of memory and computational requirements, showing that the approach achieves competitive GPU throughput. The empirical results on an NVIDIA GeForce RTX 4060 Laptop GPU validate these trade-offs, offering practical insights into kernel optimization for real-world deployment.

Despite its strengths, this work highlights areas for further improvement. For example, while the pre-computation strategy mitigates the computational overhead, its current implementation struggles to utilize GPUs to their full capabilities. Moreover, as the current implementation relies heavily on GPU compute capacity, optimizing SICK BIDE for other hardware accelerators (e.g., TPUs or custom ASICs) could unlock additional performance gains.

Looking ahead, several exciting directions emerge for future research. These include:

- Hierarchical and Approximation Techniques: Developing hierarchical computation strategies or approximation methods to further reduce memory and FLOP requirements without sacrificing accuracy.
- Mixed-Precision and Quantization: Exploring lower-precision or quantized computation for SICK to improve efficiency on resource-constrained devices.
- Broader Application Domains: Extending BIDE to other domains, such as becoming an LLM's output layer
- Integration with AI Frameworks: Seamlessly integrating SICK BIDE into popular AI frameworks like PyTorch or TensorFlow, making it accessible to a broader audience.

- Scaling Beyond Current Constraints: Investigating ways to handle higher input dimensions or batch sizes by leveraging distributed GPU computation or advanced memory management techniques.

  In conclusion, the BIDE and SICK framework represents a significant step forward in leveraging binary representations for efficient assumption-free distribution modeling. By combining theoretical rigor with practical innovations, we hope this work inspires further exploration and application of binary-based neural architectures in diverse machine learning and AI challenges.

Acknowledgments

This was was supported in part by......

References

[1] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[3] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[4] Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows, 2016.

[5] Sander Land and Max Bartolo. Fishing for magikarp: Automatically detecting under-trained tokens in large language models, 2024.

[6] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Pier Giuseppe Sessa, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikuła, Mateo Wirth, Michael Sharman, Nikolai Chinaev, Nithum Thain, Olivier Bachem, Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruibo Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vlad Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali Ahmed, Zhitao Gong, Tris Warkentin, Ludovic Peran, Minh Giang, Clément Farabet, Oriol Vinyals, Jeff Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy. Gemma: Open models based on gemini research and technology, 2024.

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[8] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

[9] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024.

[10] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024.

[11] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. In *arXiv preprint-arXiv:1805.02867*, 2018.

[13] Linting Xue, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel. Byt5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306, 03 2022.

[14] Pedro Rodriguez John Nguyen Benjamin Muller Margaret Li Chunting Zhou Lili Yu Jason Weston Luke Zettlemoyer Gargi Ghosh Mike Lewis Ari Holtzman† Srinivasan Iyer Artidoro Pagnoni, Ram Pasunuru. Byte latent transformer: Patches scale better than tokens. 2024.

[15] Azul Garza, Cristian Challu, and Max Mergenthaler-Canseco. Timegpt-1. In *arXiv:2310.03589*, 2023.

[16] Anastasios N. Angelopoulos and Stephen Bates. A gentle introduction to conformal prediction and distribution-free uncertainty quantification, 2022.

[17] Gerald Woo, Chenghao Liu, Akshat Kumar, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. Unified training of universal time series forecasting transformers, 2024.

A Notations

Numbers:

- B: Number of bits of BIDE
- H: Hidden size of BIDE
- F: Element size of BIDE dtype
- $\mathcal{B}$: Batch size
- $\mathcal{K}$: Input dimension (2^B)
- N: Batch dimension block size
- K: Input dimension block size

BIDE Tensors:

- W: Weights of BIDE 1st layer, shape (N, H, B)
- r: Weights of BIDE 2nd layer, shape (N, H)
- b: input values represented as bits mapped to $\{-1, 1\}$., shape (N, K, B)
- z: intermediate values of BIDE, shape (N, K, H)
- h: activations of BIDE's hidden layer, shape (N, K, H)
- l: logits (output) of BIDE, shape (N, K)
- m: max logit of BIDE, shape (N)
- s: sum of the exponential of the logits (normalization constant), shape (N)
- p: probabilities of BIDE, shape (N, K)

Precomputed Tensors:

- $\tilde{K} = 2^{\frac{\log_2 K}{2}} = \sqrt{2^{log_2 K}} = \sqrt{K}$: the number of combinations of inputs for half the bits of BIDE.
- $\tilde{b}_1$ and $\tilde{b}_2$: Respectively the first and last half of the input bits representation
- $\tilde{z}_1$ and $\tilde{z}_2$: The intermediate values of BIDE computed over respectively $\tilde{b}_1$ and $\tilde{b}_2$

B Theoretical Analysis for SICK

B.1 Bruteforce-variant

Memory:

- Load W: $FNHB$

- Load r: FNH
- Write m: FN
- Write s: FN

Total: $FN(H(B+1)+2)$

Compute:

- Compute b (», &, fma): $KB3$
- Compute z (gemm): $NKHB2$
- Compute h (relu): NKH
- Compute l (gemm): $NKH2$
- Compute m (max): NK
- Compute s (-m, exp, sum): $NK3$

Total: $K(3B+N(H(B2+3)+4))$

B.2 Precomputed-variant

- Load $\tilde{z}_1$: $FNH\tilde{K}$
- Load $\tilde{z}_2$: $FNH\tilde{K}$
- Load r: FNH
- Write m: FN
- Write s: FN

Total: $FN(H(2^{B/2+1}+1)+2)$

Compute:

- Compute $b1$ (», &, fma): $\tilde{K}B3$
- Compute $b2$ (», &, fma): $\tilde{K}B3$
- Compute $\tilde{z}_1$ (gemm): $N\tilde{K}HB2$
- Compute $\tilde{z}_2$ (gemm): $N\tilde{K}HB2$
- Compute z (+): NKH
- Compute h (relu): NKH
- Compute l (gemm): $NKH2$
- Compute m (max): NK
- Compute s (-m, exp, sum): $NK3$

Total: $6\tilde{K}B+N(4\tilde{K}HB+K(4H+4))$

B.3 Numerical application

We would like to estimate the throughput needed to viably run BIDE on a device. One particular use case that interests us is the use of BIDE as a LLM head. In particular let's consider as a baseline the nano-gpt repository which is a minimalistic implementation of GPT-2. Training this type of model on modern GPUs with mixed precision, we can expect roughly that the final layer + loss computation should be in the order of $10ms$ for a batch size of 32 with a context size of 512. If we want to use BIDE as a LLM head (either to serve as an implicit distribution over tokens or over a real number) we would like a reasonable precision like at least 16 bits. For this setting to be viable we can ask the simple question: How many memory bandwidth / FLOPs are required to run 16bits-BIDE with batch size 32 such that the loss computation in float16 is $10ms$?

- $B=16$

- $H = 32$ (2x the input size, default setting)
- $\mathcal{B} = 32 * 512 = 16384$
- $\mathcal{K} = 2^B = 65536$
- $F = 2$

Tensor/Operation	**Bruteforce Variant**	**Precomputed Variant**
Load W	16.78 Mb	-
Load r	1.05 Mb	1.05 Mb
Load $\tilde{z}_1$	-	268.44 Mb
Load $\tilde{z}_2$	-	268.44 Mb
Write m	32.77 kb	32.77 kb
Write s	32.77 kb	32.77 kb
Total	17.89 Mb	537.99 Mb
Required bandwidth	1.79GB/s	53.8GB/s

Table 4: Comparison of Memory Requirements for Bruteforce and Precomputed Variants.

Tensor/Operation	**Bruteforce Variant**	**Precomputed Variant**
Compute b	3.15 Mops	-
Compute $b1$	-	12.29 kops
Compute $b2$	-	12.29 kops
Compute z	1.1 Tops	34.36 Gops
Compute $\tilde{z}_1$	-	4.29 Gops
Compute $\tilde{z}_2$	-	4.29 Gops
Compute h	34.36 Gops	34.36 Gops
Compute l	68.72 Gops	68.72 Gops
Compute m	1.07 Gops	1.07 Gops
Compute s	3.22 Gops	3.22 Gops
Total	1.21 Tops	150.32 Gops
Required capacity	120.69TFLOPs	15.03TFLOPs

Table 5: Comparison of Memory and Compute Requirements for Bruteforce and Precomputed Variants.

C Formula for backward pass

Notation

- $\mathcal{L}$: Loss function
- n, N: index of the minibatch example and number of examples in the minibatch.
- k and K: index of the binary input value and number of binary input combinations.
- b_{kj}: representation of bit j (-1 if the bit is 0, else 1) for the k-th binary input value
- W: weights of the first layer of the BIDE MLP
- r: weights of the second layer of the BIDE MLP
- h_{ki}: value of the i-th hidden unit for the k-th binary input value
- l_k: logit of the k-th binary input value
- p_k: probability of the k-th binary input value
- y_n: index of the target value for the n-th example
- δ_{ky_n}: Kronecker delta, equal to 1 if $k = y_n$ and 0 otherwise

- m: maximum value of the logits (used for softmax numerical stability)
- s: sum of the exponentials of the logits (used for softmax normalization)

Calculations

We want to compute $\frac{\partial \mathcal{L}}{\partial r}$ and $\frac{\partial \mathcal{L}}{\partial W}$ in an efficient way. Here are the calculations assuming mean reduction for the loss over the minibatch:

Gradient of r

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial r_i} &= \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \frac{\partial l_k}{\partial r_i} \frac{\partial \mathcal{L}}{\partial l_k} \\
&= \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} h_{ki}(p_k - \delta_{ky_n}) \\
&= \frac{1}{N} \sum_{n=1}^{N} \left(\sum_{k=1}^{K} h_{ki} p_k - \sum_{k=1}^{K} h_{ki} \delta_{ky_n} \right) \\
&= \sum_{k=1}^{K} h_{ki} p_k - \frac{1}{N} \sum_{n=1}^{N} h_{y_n i} \\
&= \sum_{k=1}^{K} h_{ki} \frac{\exp(l_k - m)}{s} - \frac{1}{N} \sum_{n=1}^{N} h_{y_n i} \\
&= \frac{1}{s} \sum_{k=1}^{K} h_{ki} \exp(l_k - m) - \frac{1}{N} \sum_{n=1}^{N} h_{y_n i}
\end{aligned}$$

Gradient of W

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W_{ij}} &= \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \frac{\partial h_{ki}}{\partial W_{ij}} \frac{\partial l_k}{\partial h_{ki}} \frac{\partial \mathcal{L}}{\partial l_k} \\
&= \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} b_{kj} \mathbb{1}_{h_{ki}>0} r_i (p_k - \delta_{ky_n}) \\
&= \frac{1}{N} \sum_{n=1}^{N} \left(\sum_{k=1}^{K} b_{kj} \mathbb{1}_{h_{ki}>0} r_i p_k - \sum_{k=1}^{K} b_{kj} \mathbb{1}_{h_{ki}>0} r_i \delta_{ky_n} \right) \\
&= \sum_{k=1}^{K} b_{kj} \mathbb{1}_{h_{ki}>0} r_i p_k - \frac{1}{N} \sum_{n=1}^{N} b_{y_n j} \mathbb{1}_{h_{y_n i}>0} r_i \\
&= \sum_{k=1}^{K} b_{kj} \mathbb{1}_{h_{ki}>0} r_i \frac{\exp(l_k - m)}{s} - \frac{1}{N} \sum_{n=1}^{N} b_{y_n j} \mathbb{1}_{h_{y_n i}>0} r_i \\
&= \frac{1}{s} \sum_{k=1}^{K} b_{kj} \mathbb{1}_{h_{ki}>0} r_i \exp(l_k - m) - \frac{1}{N} \sum_{n=1}^{N} b_{y_n j} \mathbb{1}_{h_{y_n i}>0} r_i
\end{aligned}$$