# UltraSparseBERT: 99% Conditionally Sparse Language Modelling

**Peter Belcak**
NVIDIA
pbelcak@nvidia.com

**Roger Wattenhofer**
ETH Zürich
wattenhofer@ethz.ch

## Abstract

Language models only really need to use a tiny fraction of their neurons for individual inferences.

We present UltraSparseBERT, a BERT variant that uses 0.3% of its neurons during inference while performing on par with similar BERT models. UltraSparseBERT selectively engages just 12 out of 4095 neurons for each layer inference. This is achieved by reorganizing feedforward networks into fast feedforward networks (FFFs).

To showcase but one benefit of high sparsity, we provide an Intel MKL implementation achieving 78x speedup over the optimized feedforward baseline on CPUs, and an OpenAI Triton implementation performing forward passes 4.1x faster than the corresponding native GPU implementation. The training and benchmarking code is enclosed.

## 1 Introduction

Feedforward layers hold the majority of the parameters of language models (Brown et al., 2020; Anil et al., 2023). However, not all of their neurons need to be engaged in the computation of the feedforward layer output at inference time for every input.

A growing body of work is taking advantage of this fact in a top-down fashion, making use of a method commonly referred to as "mixture of experts" (Shazeer et al., 2017; Lepikhin et al., 2020; Fedus et al., 2022). This method consists of subdividing a large feedforward network into blocks ("experts"), designating some blocks to form a gating network, and jointly training both the experts and the gating network to produce the layer's outputs while using only a fraction of layer parameters, conditionally on the input.

The covariant approach, dubbed "fast feedforward networks", is to introduce conditional execution in a bottom-up fashion, utilizing individual neurons rather than blocks to perform gating and be executed conditionally (Belcak and Wattenhofer, 2023). We employ this approach and produce UltraSparseBERT, a variant of the BERT architecture (Devlin et al., 2018) that reorganizes feedforward networks into simplified fast feedforward networks (FFFs). In terms of downstream performance, UltraSparseBERT performs on par with other BERT-like models that are similar in size and undergo similar training procedures. The intermediate layers of UltraSparseBERT are, however, effectively much sparser by design: given a feedforward (FF) and a fast feedforward (FFF) network, each with $n$ neurons, the FFF uses the parameters of only $\mathcal{O}(\log_2 n)$ neurons instead of $\mathcal{O}(n)$ as for FF. This is a consequence of the fact that FFFs organize their neurons into a balanced binary tree, and execute only one branch of the tree conditionally on the input. In terms of output produced by the intermediate layers, such a method of execution is equivalent to treating the weights of all unused neurons as zeroes and manifests itself as conditional sparsity, since the choice of effectively non-zero neurons is conditional on the layer input.

Performing inference on an FFF amounts to performing conditional matrix multiplication (CMM), in which the rows of the input dot with the columns of neural weights one at a time, and the weight column to proceed with is chosen depending on the output of the previous dot-product operation. In this manner, all neurons are used only by some inputs and no input needs more than just a handful of neurons to be handled by the network. This is in contrast with dense matrix multiplication (DMM), which lies at the heart of the traditional feedforward networks, and which computes the dot products of all rows with all columns.

Recent advances in deep learning infrastructure have made it possible to produce efficient implementations of conditional matrix multiplication based on both popular computational frameworks as well as custom kernel code. We showcase and provide three implementations of FFF forward pass based on advanced PyTorch compilation, the OpenAI Triton framework, and the Intel MKL routines. In a later section, we give a comparison of each implementation to the corresponding optimized baseline and note that while there is already clear evidence of significant acceleration, there is potential for more.

**Reproducibility.** We share our training, finetuning, and benchmarking code as well as the weights of our best model. For a quick conceptual verification, the fact that only 12 neurons are used in the inference of UltraSparseBERT can be verified simply by zeroing the output of all but the chosen neurons, and we also give the code for this.

**Contributions.**

- We present UltraSparseBERT, a BERT-like model that has 4095 neurons but selectively uses only 12 (0.03%) for inference.
- We finetune UltraSparseBERT for standard downstream tasks and find that it performs on par with its BERT peers.
- We provide three implementation that make use of the high level of sparsity in UltraSparseBERT to perform faster feedforward layer inference.
- Through UltraSparseBERT and the already considerable speedups by early FFF implementations, we demonstrate the potential of bottom-up conditional neural execution in language modelling.

## 2 Model

### 2.1 Architecture

Our architectural starting point is the crammedBERT architecture (Geiping and Goldstein, 2023), which we implement to the letter in all but the nature of intermediate layers. There, the feedforward networks contained in the intermediate layers of the crammedBERT transformer encoder are replaced with fast feedforward networks (Belcak and Wattenhofer, 2023).

We make the following simplifying changes to the original fast feedforward networks:

1. *Remove all differences between leaf and non-leaf nodes.* In particular, we use the same (GeLU) activation function across all nodes, equip all nodes with output weights, and remove all output biases.
2. *Fix the leaf size to 1.*
3. *Allow multiple FFF trees in parallel.* We allow for multiple FFF trees to jointly compute the intermediate layer outputs. This is achieved by summing the outputs of the individual trees and presenting the sum as the intermediate layer output.

We denote a model with $K$ trees of depth $D + 1$ by appending a suffix to the model name, i.e. UltraSparseBERT-$K$x$D$. Note that for consistency, we consider a tree with no edges to have depth $0$. A BERT-base-sized model with the traditional feedforward layer of width 3072 is then just a special case of UltraSparseBERT, namely UltraSparseBERT-3072x0.

We train a full range of increasingly deeper and narrower models, starting from UltraSparseBERT-3072x0 and proceeding with UltraSparseBERT-1536x1, UltraSparseBERT-512x2, etc..

### 2.2 Training

We follow the final training procedure of crammedBERT (Geiping and Goldstein, 2023), namely disabling dropout in pretraining and making use of the 1-cycle triangular learning rate schedule. By default, we train every model for 1 day on a single A6000 GPU, except for the final UltraSparseBERT-1x11-long model, which we train 2 times longer using the same regime for slightly better downstream performance.

### 2.3 Downstream Performance

#### 2.3.1 Setup

We finetune all UltraSparseBERT models for the RTE, MRPC, SST, STS-B, MNLI, QQP, QNLI, and CoLA tasks of the GLUE benchmark (Wang et al., 2018) and report evaluation scores as in Geiping and Goldstein (2023) for consistency. In short, this approach amounts to finetuning for 5 epochs with learning rate $4 \times 10^{-5}$ across all tasks.

We find that UltraSparseBERT models finetuned in this manner for CoLA end up being undertrained if only 5 training epochs are used. Therefore, we extend the number of CoLA finetuning epochs to 15. This leads to little to no improvement for the

| Model | $N_\mathrm{T}$ | $N_\mathrm{I}/N_\mathrm{T}$ | RTE | MRPC | STSB | SST-2 | MNLI | QNLI | QQP | Avg | CoLA | Avg |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Baselines** | | | | | | | | | | | | |
| crammedBERT-3072 | 4095 | 100.0% | 58.8 | 87.6 | 85.2 | 91.9 | 82.8 | 90.4 | 89.0 | 83.6 | 45.0 | 79.3 |
| crammedBERT-4095 | 3072 | 100.0% | 57.6 | 89.1 | 85.9 | 91.9 | 81.3 | 90.9 | 87.6 | 83.2 | 47.9 | 79.3 |
| **UltraSparseBERTs** | | | | | | | | | | | | |
| UltraSparseBERT-3072x0 | 3072 | 100.0% | 56.7 | 88.9 | 86.3 | 92.3 | **82.9** | **92.3** | 88.0 | **83.8** | **48.4** | **79.9** |
| UltraSparseBERT-1536x1 | 4608 | 66.6% | 55.2 | **89.4** | 85.0 | 91.9 | 82.2 | 90.1 | 89.0 | 83.1 | 47.5 | 79.2 |
| UltraSparseBERT-512x2 | 3584 | 42.9% | 59.2 | 87.7 | 86.0 | 89.9 | 81.9 | 90.3 | 89.3 | 83.3 | 46.2 | 79.2 |
| UltraSparseBERT-256x3 | 3840 | 26.7% | 54.2 | 87.4 | 85.9 | 91.6 | 81.6 | 90.0 | 89.1 | 82.7 | 48.0 | 78.8 |
| UltraSparseBERT-128x4 | 3968 | 16.1% | 58.4 | 87.5 | **87.2** | **92.3** | 81.2 | 89.9 | **90.0** | 83.5 | 45.9 | 79.3 |
| UltraSparseBERT-64x5 | 4032 | 9.5% | 55.7 | 89.0 | **87.2** | 91.4 | 81.6 | 90.2 | 89.4 | 83.3 | 46.1 | 79.1 |
| UltraSparseBERT-32x6 | 4064 | 5.5% | 57.6 | 88.2 | 86.1 | 91.2 | 81.0 | 89.2 | 88.3 | 82.8 | 40.6 | 78.1 |
| UltraSparseBERT-16x7 | 4080 | 3.1% | 55.5 | 89.0 | 86.7 | 88.9 | 80.1 | 89.4 | 86.9 | 82.1 | 41.5 | 77.6 |
| UltraSparseBERT-8x8 | 4088 | 1.8% | 56.2 | 88.4 | 85.4 | 88.7 | 80.6 | 89.3 | 86.4 | 81.9 | 32.7 | 76.5 |
| UltraSparseBERT-4x9 | 4092 | 1.0% | 53.8 | 85.9 | 85.7 | 89.6 | 81.9 | 89.3 | 88.0 | 82.0 | 31.8 | 76.4 |
| UltraSparseBERT-2x10 | 4094 | 0.5% | **59.9** | 88.8 | 85.3 | 87.4 | 79.9 | 89.2 | 86.1 | 82.0 | 35.4 | 76.9 |
| UltraSparseBERT-1x11 | **4095** | **0.3%** | 57.8 | 88.1 | 86.1 | 89.7 | 80.2 | 89.3 | 87.1 | 82.3 | 37.1 | 77.3 |
| **Final Model** | | | | | | | | | | | | |
| UltraSparseBERT-1x11-long | 4095 | 0.3% | 60.7 | 87.5 | 86.4 | 89.9 | 81.3 | 89.7 | 87.6 | 83.0 | 35.1 | 77.7 |
| **External Baselines** | | | | | | | | | | | | |
| OpenAI GPT | 3072 | 100% | 56.0 | 82.3 | 80.0 | 91.3 | 81.4 | 87.4 | 70.3 | 78.8 | 45.4 | 75.1 |
| DistilBERT | 3072 | 100% | 59.9 | 87.5 | 86.9 | 91.3 | 82.2 | 89.2 | 71.3 | 81.2 | 52.1 | 77.6 |
| BERT-base | 3072 | 100% | 66.4 | 88.9 | 85.8 | 93.5 | 83.4 | 90.5 | 71.2 | 83.0 | 51.3 | 79.6 |

Table 1: The results of various language models on the GLUE-dev test sets. $N_\mathrm{T}$ denotes the number of neurons available for training, $N_\mathrm{I}/N_\mathrm{T}$ the proportion of neurons that are used for a single inference. "Avg" denotes the average score of all the task results to the left of the column. **Emphasis** marks the best crammed 1-day UltraSparseBERT performance for the given column. OpenAI GPT, DistilBERT, and BERT-base refer to models reported in Radford et al. (2018); Sanh et al. (2019); Devlin et al. (2018). Experimentation conducted according to the instructions in Wang et al. (2018) and the precedent of Geiping and Goldstein (2023).

baseline crammedBERT models but has a significant impact on the CoLA performance of UltraSparseBERTs.

### 2.3.2 Results

The results of our finetuning are listed in Table 1.

We see that UltraSparseBERT variants trained for 1 day on a single A6000 GPU all retain at least 96.0% of the GLUE downstream predictive performance of the original BERT-base model (Devlin et al., 2018). We also observe that the performance decreases with the increasing depth of the FFFs. Note, however, that the majority of the performance decrease due to the increasing depth is caused by only a single task – CoLA. This behaviour has previously been observed in the literature and is in line with other work trying to compress BERT behaviour into smaller models (Sun et al., 2019; Turc et al., 2019; Mukherjee et al., 2021). If we disregard CoLA, at least 98.6% of the predictive performance is preserved by all UltraSparseBERT model.

Furthermore, we see that save from CoLA, our best model – UltraSparseBERT-1x11-long – performs on par with the original BERT-base model while using only 0.3% of its own neurons, which amounts to a mere 0.4% of BERT-base neurons. We share the weights of this model.

## 3 Inference

FFFs as a part of large language models have a considerable acceleration potential. At the center of their promise sits the operation of conditional matrix multiplication.

### 3.1 Algorithm

Belcak and Wattenhofer (2023) gives recursive pseudocode for FFF inference. We list the pseudocode for CMM and the consecutive inference for FFFs, with modifications as per Section 2.1. In Algorithm 1, $B$ denotes the batch size, $H$ the layer input width (transformer hidden dimension), $2^D - 1$ is the number of neurons, and $M_{\star,k}$, $M_{l,\star}$ denote the $k$-th column and $l$-th row of $M$, respectively. The result of the $>$-comparison in CMM is assumed to be an integer $\in \{0, 1\}$.

| | | CPU Implementation | | | GPU Implementation | | |
|---|---|---|---|---|---|---|---|
| Model | Limit | Level 1 | Level 2 | Level 3 | Native fused | BMM | Triton |
| BERT-base-4095 | 1.0x | 1.0x | 1.0x | 1.0x | 1.0x | 1.0x | 1.0x |
| UltraSparseBERT-1x11 | **341.2x** | **130.7x** | **255.1x** | - | - | **1.3x** | **5.5x** |

Table 2: The results of the feedforward inference acceleration evaluation. **Emphasis** highlights the better "fair comparison" performance.

**Algorithm 1:** FFF inference forward pass.

```
Input: $B \times H$ input matrix $I$,
       $(2^D - 1) \times H$ weight matrix $W^{\text{in}}$,
       $(2^D - 1) \times H$ weight matrix $W^{\text{out}}$
Intermediate: $B \times D$ logit matrix $L$,
              $B \times D$ node index matrix $N$
Output: $B \times H$ matrix $O$

Function CMM($I, W^{in}$):
    for $d \in \{1, \dots, D-1\}$ do
        $L_{\star,d} \leftarrow I \left( W^{\text{in}}_{[N_{\star,d-1}],\star} \right)^{\mathrm{T}}$
        $N_{\star,d} \leftarrow 2N_{\star,d-1} + 1 + (L_{\star,d} > 0)$
    end
    return $L, N$

Function FFF_I($I, W^{in}, W^{out}$):
    $L, N \leftarrow$ CMM($I, W^{in}$)
    $L \leftarrow$ Activation($L$)
    for $d \in \{0, \dots, D-1\}$ do
        $O_{\star,d} \leftarrow L_{\star,d} \cdot W^{\text{out}}_{N_{\star,d},\star}$
    end
    return $O$
```

## 3.2 Inference Performance

**Implementations.** For CPU inference, we use the Math Kernel Library available as a part of the Intel oneAPI. Level 1-3 implementations are implementations that use Level 1-3 BLAS routines, respectively.

The native fused implementation uses the native fused feedforward layer kernel. Note that this is the fastest GPU implementation for FF layers but no such kernel currently exists for FFFs due to the nature of CMM. The BMM implementation uses the batched matrix multiplication and activation kernels for both FFs and FFFs. The support for this implementation without copying is currently only available on PyTorch nightly builds. Triton implementation is our custom OpenAI Triton kernel code for both FFs and FFFs, performing fused DMM/CMM and activation on the level of vector/matrix elements.

**Methodology.** For CPU inference, we perform 250 forward passes per entry on Intel(R) Core(TM) i7-6700HQ CPUs under Intel MKL v2023.2.0, using 64-bit variants of all routines. We report the mean time taken by single inference, noting that the value of the standard deviation always lay well under 2% of the mean. For GPU inference, we perform 1000 forward passes per entry on NVIDIA RTX A6000 GPUs under CUDA v12.1 and PyTorch 2.1.1-nightly. We measure the GPU time and report the mean time taken, with the standard deviation again well under 2% of the mean in all cases. We take batch size $B = 128 \times 128$ (equivalent to the BERT pretraining context token batch size) and hidden dimension $H = 768$.

**Results.** Table 2 lists the performance comparison of feedforward and fast feedforward layers as they appear in BERT-base and UltraFastBERT-1x11. Each column of the table lists the relative inference FFF-over-FF implementation speedups *when using the same linear-algebraic routine primitives*. The two entries missing Table 2 are for the unavailable BLAS Level 3 and Native fused implementations of FFFs.

The speedups reported in Table 2 give "fair comparisons", meaning that in each case, both the FF and FFF implementation used exactly the same primitive linear-algebraic operations. One may also be interested in knowing how the best implementations of FFF currently fare against the best implementations of FF, even though the ones for FF use primitives unavailable for FFF. On CPU, the Level 2 implementation of FFF performs inference **78x** faster than the fastest implementation of FF. On GPU, the Triton implementation of FFF delivers a **4.1x** speedup over the fastest (native fused) implementation of FF. In sum, there are attractive benefits to high-levels of conditional sparsity.

## 4 Limitations

A limitation of our training work is that for most FFF configurations, we only perform one training run. It is possible that the downstream performance of the individual configurations would vary across multiple training runs. This is partially mitigated by the use of multiple fine-tuning runs to find the downstream task score as per the precedent for BERT models on the GLUE benchmark.

A major weakness of inference speed measurements is that they depend heavily on the hardware used as well as the low-level optimization provided as the interface to the hardware. To illustrate how fast the landscape is changing: in October 2023, neither the non-copying BMM nor the Triton implementation leveraging local conditionality would have been possible. Our sparsity argument, however, remains intact, and is easily verifiable through the (default provided) implementation that zeroes out the contributions of all unused neurons.

Our work focuses on efficiency of existing models and inherits the risks of the models used, if any.

## References

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*.

Peter Belcak and Roger Wattenhofer. 2023. Fast feedforward networks. *arXiv preprint arXiv:2308.14711*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270.

Jonas Geiping and Tom Goldstein. 2023. Cramming: Training a language model on a single gpu in one day. In *International Conference on Machine Learning*, pages 11117–11143. PMLR.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*.

Subhabrata Mukherjee, Ahmed Hassan Awadallah, and Jianfeng Gao. 2021. Xtremedistiltransformers: Task transfer for task-agnostic distillation. *arXiv preprint arXiv:2106.04563*.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*.

Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. 2019. Patient knowledge distillation for bert model compression. *arXiv preprint arXiv:1908.09355*.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-read students learn better: On the importance of pre-training compact models. *arXiv preprint arXiv:1908.08962*.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.