LookupFFN: Making Transformers Compute-lite for CPU inference

Zhanpeng Zeng [1] **Michael Davies** [1] **Pranav Pulijala** [1] **Karthikeyan Sankaralingam** [1 2] **Vikas Singh** [1]

Abstract

While GPU clusters are the de facto choice for training large deep neural network (DNN) models today, several reasons including ease of workflow, security and cost have led to efforts investigating whether CPUs may be viable for inference in routine use in many sectors of the industry. But the imbalance between the compute capabilities of GPUs and CPUs is huge. Motivated by these considerations, we study a module which is a workhorse within modern DNN architectures, GEMM based Feed Forward Networks (FFNs), and assess the extent to which it can be made compute- (or FLOP-) lite. Specifically, we propose an alternative formulation (we call it LookupFFN) to GEMM based FFNs inspired by the recent studies of using Locality Sensitive Hashing (LSH) to approximate FFNs. Our formulation recasts most essential operations as a memory look-up, leveraging the trade-off between the two resources on any platform: compute and memory (since CPUs offer it in abundance). For RoBERTa language model pretraining, our formulation achieves similar performance compared to GEMM based FFNs, while dramatically reducing the required FLOP. Our development is complemented with a detailed hardware profiling of strategies that will maximize efficiency – not just on contemporary hardware but on products that will be offered in the near/medium term future. Code is avaiable at `https://github.com/mlpen/LookupFFN`.

1. Introduction

CPU-based inference in the data-center is growing in importance as evidenced by recent server chip announcements from IBM, Intel, AMD and ARM (Lichtenau et al., 2022; Intel; Nassif et al., 2022; AMD; Bhat) and academic efforts (Liu et al., 2019b; Mittal et al., 2022; Nori et al., 2021; Zhang et al., 2019). Some of the technical and business motivations include latency, security, privacy and the fact that modern data-intensive workloads have AI tasks embedded in a pipeline of non-AI tasks. Further, CPUs are a generic platform common across servers and clients faithfully serving the compute needs of businesses, which makes it attractive. And finally – it comes down to the cost of running the full workload. Unfortunately, CPU chips lack the computational intensity of raw-FLOPS compared to GPUs. On the positive side, CPUs provide tremendously large caches in the range of 128MB to 192MB, and even larger (Burd et al., 2022), which is currently under-utilized. Furthermore, such caches made out of SRAMs are more than an order of magnitude more energy efficient to access compared to DRAMs (DDR, GDDR, or HBM) (Jouppi et al., 2021; Horowitz, 2014), while providing $4\times$ access bandwidth increase [1]. In this context, this paper revisits one of the fundamental building blocks of modern deep-learning: the Feed Forward Network (FFN), to examine algorithmic reformulations to make it FLOP-lite and CPU friendly.

FFNs are essential components in almost all deep neural networks, such as convolutional networks (Howard et al., 2017) or Transformers (Vaswani et al., 2017). They heavily rely on General Matrix Multiply (GEMM), which is extremely compute intensive, especially for large scale models common in the community today. Many alternatives (Chen et al., 2020; 2021; Fedus et al., 2022; Zhang et al., 2018; Moczulski et al., 2016) have been proposed to reduce the FLOP needs of FFNs. In the context of inference, one may use generic pruning/quantization techniques (Dong & Yang, 2019; Lee et al., 2019; Jacob et al., 2018; Zafrir et al., 2019; Kim et al., 2021), after training has concluded. This strategy is typically agnostic of specific modules in the architecture, and is applied to the entire model. Notice that if module-specific FLOP reduction is accomplished somehow, the full model will still benefit from a scheme like pruning, prior to deployment. Since such an idea would complement any reformulation of FFNs, it is more meaningful to instead focus the discussion on existing ideas that targets FLOP reduction further upstream.

[1]University of Wisconsin, Madison, USA [2]NVIDIA Research. Correspondence to: Zhanpeng Zeng <zzeng38@wisc.edu>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

[1]AMD Zen2 for example, allows 64 bytes per cycle into each core: with 32 cores running 3.2 GHz that amounts to 6 TB/sec.

LSH can make FFNs FLOP-lite. One popular line of work shows how to use Locality Sensitive Hashing (LSH), to address the computational bottleneck of feed-forward via adaptive sparsity. For example, Slide (Chen et al., 2020) uses LSH to retrieve a small subset of units that omit high activation via maximum inner product search (MIPS) and only computes the outputs of these units, resulting in a sparse network. However, LSH poses certain difficulties. Due to the randomness of hash functions, a large number of hash functions are needed to get good MIPS results (see Fig. 1, left). Also, the skewed hash bucket distribution makes LSH-based FFNs harder to parallelize since the computational load of different inputs can be very different (see Fig. 1, right). Considering training, due to the constantly evolving parameters, the hash tables need to be constantly updated to adapt for the changing parameter matrices (rehash) creating a large overhead during training (Chen et al., 2021).

Extensions/modifications of Slide. The Slide result shows that an approach based on LSH can be effective at reducing the FLOP count in FFNs. Motivated by this observation, the authors in Mongoose (Chen et al., 2021) proposed strategies for improvements. Specifically, by making the LSH component learnable and introducing a special update scheduler, Mongoose reduces the number of hash functions and the frequency of hash table updates (although the need for rehash is not eliminated). Separately, the skewed bucket distribution still limits the parallelism of LSH. Nonetheless, the similarities with Slide in the use of LSH make Mongoose applicable for FLOP reductions in FFNs. A distinct use of LSH was shown in YOSO (Zeng et al., 2021) for approximating the self-attention matrix. Although a reduction in FLOP count did not motivate that work, the YOSO algorithm (Zeng et al., 2021) specifically adjusts LSH so that the skewness of bucket distribution does not affect the computational load – to enable easy parallelism/efficiency gains. Therefore, YOSO is also potentially applicable for FLOP reductions in FFNs, like Slide/Mongoose. However, a large number of hash functions (and to a lesser extent, rehash) cannot be avoided. In summary, the effectiveness of these ideas for commonly used architectures remains unclear, which we will discuss in more detail later.

This work and its contributions. Motivated by the aforementioned limitations of LSH-based FFNs, in this paper, we provide a formulation of end-to-end learnable memory lookup for FFNs: LookupFFN. Specifically, we propose to directly view the hash tables as learnable modules. Projections are handled via a specialized module based on the fast Hadamard transform, which may be of independent interest. We show that the skewness of bucket distribution becomes irrelevant in our proposal. Since there are no parameter matrices as in (Chen et al., 2021; 2020), rehashing can be completely avoided, and the gradient updates are performed directly on the hash functions and hash tables (during model training). The proposed formulation is differentiable, and no special optimization on the hash modules is needed. In practice, LookupFFN can simply be integrated into common DNN models, optimization flows, and software frameworks.

Main features. Based on measurements and analytical calculations, we estimate $6\times$ (or more) reduction in FLOP compared to a vanilla FFN with almost the same accuracy. Even though our formulation requires somewhat large tables (16MB and more), with careful algorithm design, we can make the access pattern somewhat cache-friendly – achieving nearly 80% L1 cache hit rate. In particular, hardware-managed caches work well, avoiding the need for excessive optimization for the software-managed shared-memory of a GPU. In practice, this means that we are able to reduce the energy consumption of 80% of access to SRAMs (14 pico joules or pj per 64-bit access), versus 300 to 450 pj for DRAM-based access. We show that, on contemporary hardware, for inference, we are $2.51\times$ faster than a vanilla FFN. With new technology like 3D caches appearing in CPUs, we expect SRAM based bandwidth to increase even further, making LookupFFN integer factors faster as memory technology and packaging improvements continue. While this formulation has more memory lookups compared to GEMM-based FFN, the majority of the memory lookups are independent of each other. This means LookupFFN heavily relies on high memory throughput but has high tolerance to memory latency – making designing software (and potentially hardware) implementation easier.

2. Preliminaries

First, we briefly review the Transformer and then the Feed-Forward Network (FFN) within the Transformer, the use case we study in this paper. Then, we will discuss related works that motivated this paper and some salient limitations. We use $[\cdot]_i$ to denote the i-th row/entry of the matrix/vector, and use **BOLD** uppercase letters to denote matrices, **bold** lower case letters to denote vectors.

2.1. The Feed-forward Network in Transformer

Given an embedding matrix $\mathbf{X} \in \mathbb{R}^{n\times d}$ representing the embedding vectors of n tokens, a Transformer layer is

$$\mathbf{A} = \mathcal{G}(\mathbf{X}) + \mathbf{X}$$
$$\mathbf{Y} = \mathcal{F}(\mathbf{A}) + \mathbf{A}$$

where $\mathcal{G}(\cdot)$ is a multi-head attention and $\mathcal{F}(\cdot)$ is a two layer FFN. There are layer normalizations within the Transformer layer, but for notational simplicity, we omit them. The focus of this paper is the efficiency (FLOP) of $\mathcal{F}(\cdot)$.

What about efficiency of $\mathcal{G}(\cdot)$? There are various results studying how to improve the efficiency of $\mathcal{G}(\cdot)$ (Choromanski et al., 2021; Xiong et al., 2021; Beltagy et al., 2020;

Zaheer et al., 2020; Kitaev et al., 2020; Zeng et al., 2021; 2022). Of course, conceptually, our method (discussed later) can be extended to multi-head attention. But this work would be a little redundant. Such functionality is available within YOSO (Zeng et al., 2021), which provides a similar mechanism to support efficient self-attention calculation that works well but is inapplicable to FFN. Therefore, we focus on FFN $\mathcal{F}(\cdot)$.

FFN. The $\mathcal{F}(\cdot)$ is a point-wise operation applied to each row of $\mathbf{A} \in \mathbb{R}^{n\times d}$. Let $\mathbf{x} \in \mathbb{R}^d$ be any row of $\mathbf{A}$, t be the hidden dimension, and $\mathbf{W}$ and $\mathbf{V}$ be two parameter matrices in $\mathcal{F}(\cdot)$. Then, a FFN is

$$\mathbf{y} := \mathcal{F}(\mathbf{x}) = \sigma(\mathbf{x}\mathbf{W}^\top)\mathbf{V} = \sum_{i=1}^{t} \sigma(\langle \mathbf{x}, [\mathbf{W}]_i \rangle)\, [\mathbf{V}]_i \quad (1)$$

Here, biases are omitted. This operation is usually implemented via GEMM/matrix multiply. It takes $\mathcal{O}(dt)$ compute cost for one input and is usually the main bottleneck (in terms of FLOP) of a DNN.

The authors in Slide (Chen et al., 2020) observed that when σ is a softmax, the output of a FFN is dominated by only a few entries of $\sigma(\langle \mathbf{x}, [\mathbf{W}]_i \rangle)$ and proposed a sparse FFN, which uses LSH to perform a maximal inner product search (MIPS) among $[\mathbf{W}]_i$ for large $\sigma(\langle \mathbf{x}, [\mathbf{W}]_i \rangle)$ terms. Only activations of the search results $\mathcal{S}(\mathbf{x})$ are computed to approximate the full softmax with a reduced compute burden,

$$\mathbf{y} \approx \sum_{i\in\mathcal{S}(\mathbf{x})} \sigma(\langle \mathbf{x}, [\mathbf{W}]_i \rangle)\, [\mathbf{V}]_i \quad (2)$$

Constructing $\mathcal{S}(\mathbf{x})$ requires a pre-processing step that hashes $[\mathbf{W}]_i$ into multiple hash tables, and a querying step that hashes $\mathbf{x}$ to these hash tables and collects all $[\mathbf{W}]_i$ from the buckets that $\mathbf{x}$ is hashed to. There are some problems with this construction. **Rehashing:** Since $[\mathbf{W}]_i$ are constantly updated while training, the hash tables need to be constantly updated or re-constructed, referred to as rehashing. **Large #-hashes:** The LSH relies on the randomness of hash functions, so a large number of hash functions are used to obtain accurate MIPS result resulting in high query time (see left plot of Fig. 1). **Bucket skewness:** The LSH bucket distribution is skewed, so the number of $[\mathbf{W}]_i$ in different buckets are quite different and there is no control of how many $[\mathbf{W}]_i$ will be hashed into one bucket (see right plot of Fig. 1). Therefore, $|\mathcal{S}(\mathbf{x})|$ varies for different inputs. This skewness makes the workload difficult to be parallelized.

Mongoose (Chen et al., 2021) proposed a scheduler to reduce the frequency of rehashing and learnable hash functions to learn data-dependent hashing. So, the number of hash functions can be reduced without sacrificing MIPS quality, thereby (Chen et al., 2021) partially reduces the **Rehashing** and **large #-hashes** issues but introduces an additional auxiliary learning component for learnable hashing. Further, the **bucket skewness** still persists. Zeng et al. (2021) proposed a method for approximating self-attention in Transformer models, which can be extended to approximating FFNs. Zeng et al. (2021) shows that when σ is similar to the collision probability of LSH, instead of keeping track of S (as in Slide), one can store the summation of $[\mathbf{V}]_i$'s in hash buckets where $[\mathbf{W}]_i$ are hashed to, which is analogous to the LSH pre-processing step. Let f_k be a hash function, and $\mathbf{T}_k \in \mathbb{R}^{2^\tau \times d}$ be a hash table representing 2^τ d-dimensional buckets.

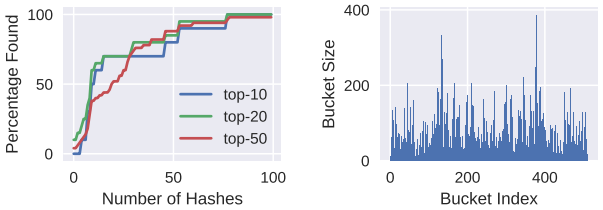

Figure 1. The left plot shows the number of hash functions used versus the percentage of top-x nearest neighbors found using these hashes. A large number of hash functions are needed for accurate MIPS result. The query time is linearly proportional to the number of hash functions. The right plot shows the bucket size of each bucket. It visualizes the bucket skewness issue.

$$[\mathbf{T}_k]_j = \frac{1}{h} \sum_{f_k([\mathbf{W}]_i)=j} [\mathbf{V}]_i \qquad \mathbf{y} \approx \sum_{k=1}^{h} [\mathbf{T}_k]_{f_k(\mathbf{x})} \quad (3)$$

Then, in the LSH querying step, we can directly estimate $\mathbf{y}$ by computing an average of one bucket of multiple $\mathbf{T}_k$ with a consistent compute cost. Therefore, the **bucket skewness** issue is solved. However, the steps of (Zeng et al., 2021) relies on the randomness of f_k, so it requires a large number of hash functions for a good estimate. Further, since $\mathbf{W}$ and $\mathbf{V}$ are evolving during training, $\mathbf{T}_k$ needs to be recomputed after every parameter update, which is inefficient. The **rehashing** and **large #-hashes** problems remain open.

None of the foregoing methods can resolve all issues. In particular, no method solves **rehashing** – all of them require rehashing when the parameters are updated. One of our goals is to completely eliminate the need for rehashing, and remove the dependency of workload on the bucket size. Further, we also hope to obtain a scheme that, if desired, can be trained end-to-end via back-propagation.

3. FFN as Lookups

Here, we present an end-to-end construction for differentiable table lookups as an efficient alternative to GEMM for FFNs where most operations are memory lookups.

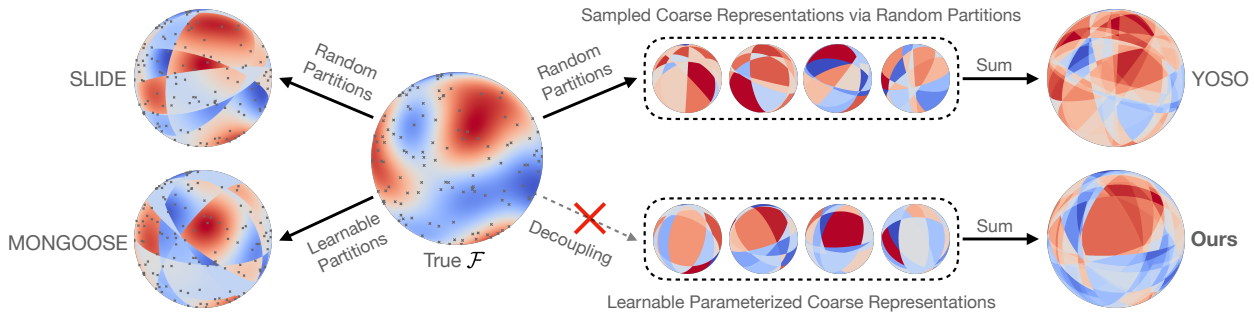

Figure 2. High level comparison of each method. The true $\mathcal{F}(\cdot) = \sum_{i=1}^{h} \sigma(\langle \cdot, [\mathbf{W}]_i \rangle) [\mathbf{V}]_i$ is constructed as a function in S^2. Here, $[\mathbf{W}]_i \in S^2$ and $[\mathbf{V}]_i \in \mathbb{R}$. The points $[\mathbf{W}]_i$ are marked in the left three figures. SLIDE, MONGOOSE, and YOSO try to construct an approximation of the true $\mathcal{F}(\cdot)$ via different uses of LSH partitions, so whenever $\mathcal{F}(\cdot)$ is updated, the LSH partitions need to be updated. Rather than approximating the function $\mathcal{F}$, our proposed method is plugged into a deep learning model and uses the back-propagated gradient to learn appropriate transformation similar to a vanilla FFN.

3.1. Differentiable Lookup

To avoid the impact of skewed bucket distribution on efficiency, we start from (3) and attempt to adjust the formulation in the setting where it is used as a FFN. The randomness of f_k in (3) is the key ingredient of (Zeng et al., 2021), but at the same time, this randomness introduces the need for a large number of hash functions to get an accurate approximation. This issue must be handled. Separately, we try to completely avoid any pre-processing steps or rehashing for evolving parameters $\mathbf{W}$ and $\mathbf{V}$.

Main idea. Observe that for YOSO, the f_k is a partition of the $\mathbb{R}^d$ space and each hash table $\mathbf{T}_k$ is a coarse representation of $\sum_{i=1}^{h} \sigma(\langle \cdot, [\mathbf{W}]_i \rangle) [\mathbf{V}]_i$ associated with f_k. Whenever f_k, $\mathbf{W}$, or $\mathbf{V}$ are updated, $\mathbf{T}_k$ needs to be updated. This is inefficient. But $\mathbf{T}_k$ is a coarse representation of a parameterized function, so we hypothesize that we might be able to directly optimize the coarse representation $\mathbf{H}_k$ and f_k to minimize the loss of the model. If possible, we also want to make it differentiable. If this is achieved, this strategy helps avoid any rehashing necessary in (2) and (3). Therefore, we consider the formulation

$$[\mathbf{T}_k]_j := \frac{1}{h} \sum_{f_k([\mathbf{W}]_i)=j} [\mathbf{V}]_i \qquad \mathbf{y} := \sum_{k=1}^{h} [\mathbf{T}_k]_{f_k(\mathbf{x})} \quad (4)$$

where $\mathbf{T}_k$ and f_k are learnable modules. Here, the dependency of $\mathbf{T}_k$ on f_k, $\mathbf{W}$, $\mathbf{V}$, as in (3), is removed. Fig. 2 is a visualization of the difference comparing LSH-based FFNs. This decoupled dependency creates a problem in that the resultant formulation is not differentiable. Zeng et al. (2021) uses the fact that (3) is an estimate of a differentiable function, and uses the gradient of this function as an estimate of the gradient of (3), however, this estimate relies on the randomness of f_k which is not available after decoupling in (4). So, the challenge is how we can train f_k and $\mathbf{T}_k$, and backpropagate to shallower layers.

Making (4) differentiable again. To figure out a solution, we need to first dive into how f_k is computed. Zeng et al. (2021) uses the hyperplane hash (Charikar, 2002) to compute the hash code. Specifically, define $\mathbf{z}_k = \mathbf{x}\mathbf{R}_k$, referred to as the "soft hash code", where $\mathbf{R}_k \in \mathbb{R}^{d \times \tau}$ is a random projection associated with f_k where τ is the length of binary representation of the hash code.

$$f_k(\mathbf{x}) = \text{decimal}(\text{sign}(\mathbf{z}_k))$$

Here, decimal is a function that maps the binary representation $\{\pm 1\}^\tau$ to a decimal representation $\{0, \cdots, 2^\tau - 1\}$. This form does not directly suggest a method for backpropagation, but observe that f_k can be expressed as

$$f_k(\mathbf{x}) = \arg\max_i (\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle)$$

where $\mathbf{S} \in \{\pm 1\}^{2^\tau \times \tau}$ is a structured matrix whose row vector

$$[\mathbf{S}]_i = \text{decimal}^{-1}(i)$$

is the binary representation of the integer i. While, by itself, this does not solve our problem, a common differentiable relaxation of $\arg\max$ is the softmax activation, and the resultant formulation for (4) is,

$$\hat{y}^* := \sum_{k=1}^{h} \sum_{i=1}^{2^\tau} \frac{\exp(\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle) [\mathbf{T}_k]_i}{\sum_{j=1}^{2^\tau} \exp(\langle \mathbf{z}_k, [\mathbf{S}]_j \rangle)} \quad (5)$$

Then, by replacing the random matrix $\mathbf{R}_k$ with a learnable parameter matrix, this formulation makes f_k a learnable hash function and $\mathbf{T}_k$ a learnable coarse representation of a function in $\mathbb{R}^d$ in an end-to-end manner.

Remaining difficulties and solutions. A naive implementation of this operation is extremely inefficient and has a runtime complexity of $\mathcal{O}(h 2^\tau d)$, which is not practical. A common choice of efficient softmax approximations is to use a small subset of softmax numerators (we denote the set of corresponding indices as $\mathcal{N}(\mathbf{z}_k)$) to approximate the full

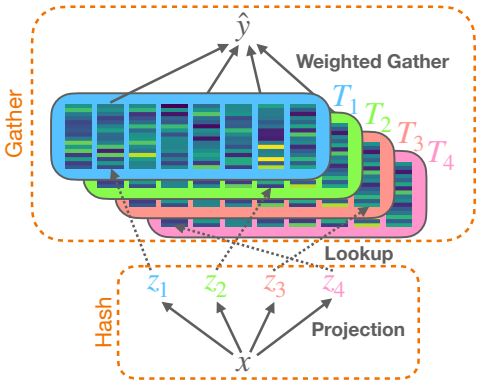

Figure 3. Illustration of LookupFFN operations.

softmax since the softmax is usually dominated by only a few entries within it (Spring & Shrivastava, 2017; Charikar & Siminelakis, 2017). Non-uniform sampling, such as LSH-based importance sampling, can be used to lower the estimation variance. However, we found that the structured matrix $\mathbf{S}$ used in (5) offers several properties that actually enables efficient approximation. Due to the structure of $\mathbf{S}$, the denominator of (5) can be rewritten as

$$\sum_{j=1}^{2^\tau} \exp(\langle \mathbf{z}_k, [\mathbf{S}]_j \rangle) = \prod_{j=1}^{\tau} (\exp([\mathbf{z}_k]_j) + \exp(-[\mathbf{z}_k]_j))$$

which only involves a $\mathcal{O}(\tau)$ cost. For calculating the numerator, we use a simple non-uniform sampling scheme for a better approximation of the softmax with a small number of samples. Due to the structure of $\mathbf{S}$, we easily know the approximate sorting order of $\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle$ among different i. Specifically, note that

$$\begin{aligned} \arg\max_i (\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle) &= \text{decimal}(\text{sign}(\mathbf{z}_k)) \\ \arg\min_i (\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle) &= \text{decimal}(-\text{sign}(\mathbf{z}_k)) \end{aligned} \tag{6}$$

When the order of magnitudes for different entries of $\mathbf{z}_k$ are not too different, the $\|[\mathbf{S}]_i - \text{sign}(\mathbf{z}_k)\|_0$ term roughly indicates the magnitude of $\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle$. A smaller distance means a larger value.

Therefore, we use the approximation

$$\hat{\mathbf{y}} = \sum_{k=1}^{h} \sum_{i \in \mathcal{N}(\mathbf{z}_k)} \frac{\exp(\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle) [\mathbf{T}_k]_i}{\prod_{j=1}^{\tau} (\exp([\mathbf{z}_k]_j) + \exp(-[\mathbf{z}_k]_j))} \tag{7}$$

where $\mathcal{N}(\mathbf{z}_k)$ can be easily sampled according to ℓ_0 difference $\|[\mathbf{S}]_i - \text{sign}(\mathbf{z}_k)\|_0$. It is much easier compared to other non-uniform sampling based softmax approximations (Spring & Shrivastava, 2017; Charikar & Siminelakis, 2017) since we can sample large numerators based on the number of sign flips away from $\text{sign}(\mathbf{z}_k)$. Further, we empirically found that in most cases, just using the largest numerator, i.e., let

$$g(\mathbf{z}_k) := \arg\max_i (\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle) \tag{8}$$

computed via (6), then $\mathcal{N}(\mathbf{z}_k) = \{g(\mathbf{z}_k)\}$ is sufficient for performance. This is our default choice for experiments.

Two main operations. The proposed learnable lookup consists of two operations: **(a)** Hash: we compute multiple $\mathbf{z}_k = \mathbf{x}\mathbf{R}_k$ for $k = 1, 2, \cdots, h$. Then, **(b)** Gather: we use $g(\mathbf{z}_k)$ (defined in (8)) for memory lookup and calculate a weighted (based on $\mathbf{z}_k$) accumulation of the lookup results $[\mathbf{T}_k]_{g(\mathbf{z}_k)}$. This procedure is illustrated in Fig. 3.

Remark 3.1. While (7) might look unfamiliar, it is closely connected to two commonly used FFNs. When σ in (1) is the sigmoid activation, let $\mathbf{z}_k = 0.5\langle \mathbf{x}, [\mathbf{W}]_k \rangle$, we note that (1) can be rewritten as

$$y = \sum_{k=1}^{h} \frac{\exp(\mathbf{z}_k) [\mathbf{V}]_k}{\exp(\mathbf{z}_k) + \exp(-\mathbf{z}_k)}$$

which is just a special case of (7) with $\tau = 1$. When σ is a GELU (Hendrycks & Gimpel, 2016) commonly used in Transformer models, let $\mathbf{z}_k = 0.851\langle \mathbf{x}, [\mathbf{W}]_k \rangle$, then a fast approximation of GELU can be written as

$$y = \sum_{k=1}^{h} \frac{1.175\mathbf{z}_k \exp(\mathbf{z}_k) [\mathbf{V}]_k}{\exp(\mathbf{z}_k) + \exp(-\mathbf{z}_k)}$$

This is again a special case of (7) with $\tau = 1$ and an additional linear scaling $1.175\mathbf{z}_k$. This scaling can be incorporated in (7) by an extra term $1.175\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle$ in the numerator.

3.2. BH4: Efficient and Expressive Projection

The problem. In practice, we compute the "soft hash code" $\mathbf{z}_k$ for multiple hash tables at once by computing $\mathbf{x}\mathbf{R}$ where $\mathbf{R} \in \mathbb{R}^{d \times (h\tau)}$. Here, $\{\mathbf{z}_1, \cdots, \mathbf{z}_h\}$ are computed at once by partitioning $\mathbf{x}\mathbf{R}$ into h τ-dimensional vectors. The time complexity for this projection is $\mathcal{O}(h\tau d)$. This is not desirable since it is compute heavy.

Some existing solutions yield unsatisfactory results. For simplicity, we assume $h\tau = d$ and d is power of 2. When computing hash codes in the LSH setting, a common efficient alternative is efficient random projections implemented via a fast Hadamard transform with $O(d \log(d))$ cost. For example, Zeng et al. (2021); Andoni et al. (2015) use

$$\mathbf{R} := \mathbf{D}_1\mathbf{H}\mathbf{D}_2\mathbf{H}\mathbf{D}_3\mathbf{H} \tag{9}$$

where $\mathbf{D}_i$ are matrices whose entries are $\{\pm 1\}$ for random sign flipping and $\mathbf{H}$ is Hadamard transform. A simple learnable extension would be to replace $\mathbf{D}_i$ with parameterized diagonal matrices. This belongs to a large family of structured efficient linear layers (SELLs) (Cheng et al., 2015; Le et al., 2013; Yang et al., 2015; Moczulski et al., 2016) For example, Moczulski et al. (2016) proposes a deep SELL,

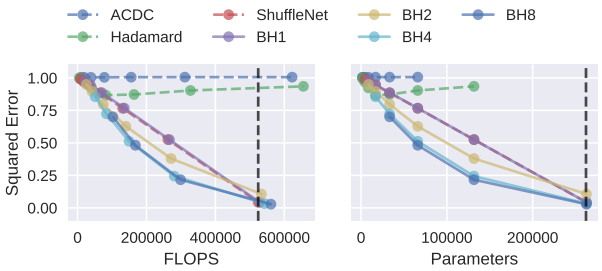

Figure 4. Approximation capacity vs FLOPs and parameters for each efficient projections. Hadamard denotes a variant of ACDC by replacing discrete cosine transform with Hadamard transform. The vertical dash lines are the FLOPs and parameters of the vanilla projection. Any results to the right of the vertical dashed lines are not meaningful, as there is no efficiency gain.

named ACDC, to increase the representation power:

$$\mathbf{R} := \prod_{i=1}^{k} \mathbf{A}_i \mathbf{C} \mathbf{D}_i \mathbf{C}^{-1} \tag{10}$$

where $\mathbf{A}_i, \mathbf{D}_i$ are parameterized diagonal matrices and $\mathbf{C}$ is the discrete cosine transform and k is a hyper-parameter. A similar construction, but using the Hadamard transform would involve replacing $\mathbf{C}$ with $\mathbf{H}$. This can be viewed as a generalization of (9). To empirically evaluate the representation power of each efficient projection, we use a toy problem of approximating a randomly generated matrix using these ideas. The results are shown in Fig. 4. We find that the representation power of (10) and its Hadamard transform variant for small k is extremely limited, but for large k, the efficiency is low and the optimization becomes difficult. We can verify this optimization difficulty from the fact that as k increases, the FLOP and parameter count increases, but the squared errors do not monotonically decrease.

A simple yet highly effective scheme. To address the optimization difficulty for large k, we propose that instead of scaling k, we can replace the diagonal matrices with their block diagonal counterparts, and scale the block size for the trade-off between efficiency and expressiveness.

$$\mathbf{R} := \prod_{i=1}^{m} \mathbf{B}_i \mathbf{H} \tag{11}$$

Here, $\mathbf{B}_i$'s are parameterized block diagonal matrices with an adjustable block size. We refer to (11) as BH{m} for different m. Fig. 5 is a visualization of the projections discussed. We empirically verify that (11) with $m = 4$ has a better trade-off between the expressiveness and efficiency compared to other m values. When the FLOP or parameter counts are similar, larger m does not increase its expressiveness, but a smaller m decreases its expressiveness.

Remark 3.2. Grouped convolution followed by channel shuffling in ShuffleNet (Zhang et al., 2018) can be directly expressed as $\mathbf{BF}$: applying a block diagonal matrix followed by a structure transform $\mathbf{F}$. ShuffleNet can be viewed as a BH1. We empirically verified that BH1 has near-identical approximation accuracy as ShuffleNet as shown in Fig. 4.

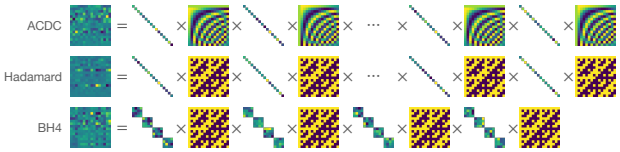

Figure 5. Visualization of efficient projections. ACDC and its Hadamard variant increase their capacity by increasing the depth. BH4 increases its capacity by increasing its block size.

4. Experiments

In this section, we will present our empirical results evaluating the benefits/limitations of replacing VanillaFFN with a LookupFFN in a Transformer, and conduct a detailed performance profiling of LookupFFN.

Note. Slide and Mongoose report experiments on a two layer neural network whose last dimension is 200K (or even 670K) (Chen et al., 2020; 2021). This is a synthetic setting to extract peak practical speedup of FFN in Slide/Mongoose because more than 99% of compute occurs in the final layer. In practical situations, such models are rare and might not reflect the actual workload of commonly used models, such as the Transformer. We are more interested in how these baselines and our method can be applied in more challenging commonly used DNN models, and what the corresponding performance impact is. As a result, we use the Transformer as a testbed to evaluate the effect of replacing VanillaFFN with baselines and our LookupFFN.

Outline. In §4.1, we compare LookupFFN's performance and FLOP reduction to baselines, and check that our formulation scales without difficulty to full size (12-layer) models. Then, in §4.2, to better understand its behavior, we perform an ablation study to study the effects of different hyper-parameters specific to lookup-based FFN. Finally, in §4.3, we analyze the performance characteristics for LookupFFN. Since our goal is to reduce the FLOP count, for comparison or even individual assessment of LookupFFN, we include the estimated FLOP count next to the model performance for each table (for comparisons). FLOPs are estimated as the number of floating operations of processing a single instance (a single token in the context of a Transformer).

Two variants are discussed in §3.1 corresponding to two different activations. To align well with the GELU activation used in Transformer, we use the linearly scaled variant of (7) with $\mathcal{N}(\mathbf{z}_k) = \{g(\mathbf{z}_k)\}$:

$$y = \sum_{k=1}^{h} \frac{\langle \mathbf{z}_k, [\mathbf{S}]_{g(\mathbf{z}_k)} \rangle \exp(\langle \mathbf{z}_k, [\mathbf{S}]_{g(\mathbf{z}_k)} \rangle) [\mathbf{T}]_{g(\mathbf{z}_k)}}{\prod_{j=1}^{r} (\exp([\mathbf{z}_k]_j) + \exp(-[\mathbf{z}_k]_j))}$$

Method	h	τ	MFLOP	Log Perplexity
VanillaFFN	-	-	4.19	1.78
Switch Transformer	-	-	2.11	1.85
Channel Shuffle	-	-	2.10	1.96
Slide	-	-	1.32	1.98
Mongoose	-	-	3.21	1.87
YOSO	-	-	0.35	2.13
LookupFFN	256	8	1.38	1.74
	128	8	0.69	1.81

Table 1. Log perplexity of each baseline. (lower is better) LookupFFN was tested with two different hyper-parameter configurations specified in h and τ columns.

Method	h	τ	MFLOP	Log Perplexity
VanillaFFN	-	-	9.44	1.37
LookupFFN	170	9	1.39	1.41

Table 2. Log perplexity when scaling to a RoBERTa-base model.

Type	Block Size	MFLOP		Log Perplexity
		Hash	Gather	
Dense	-	1.05	0.13	1.79
BH4	64	0.56	0.13	1.81
BH4	32	0.30	0.13	1.83
BH4	16	0.17	0.13	1.85

h	τ	$h\tau$	MFLOP	Log Perplexity
32	8	256	0.31	1.94
64	8	512	0.35	1.88
128	8	1024	0.69	1.81
256	8	2048	1.38	1.74

h	τ	$h\tau$	MFLOP		Log Perplexity
			Hash	Gather	
64	4	256	0.28	0.07	1.98
32	8	256	0.28	0.03	1.94
20	13	260	0.28	0.02	1.87

Table 3. Ablation study evaluating the effects of different hyper-parameters on model performance. The MFLOP columns in the top and button tables are broken down into two column showing the FLOP for Hash and Gather steps separately.

Method	h	τ	MFLOP	MNLI-m/mm
VanillaFFN	-	-	4.19	75.0/76.3
LookupFFN	256	4	0.82	74.1/74.7

Table 4. Downstream performance of RoBERTa-small models.

Implementation Details. We used PyTorch (Paszke et al., 2019) for the majority of the implementation. On the GPU, our fast Hadamard Transform and weighted gather operators are not supported by PyTorch so we implemented custom CUDA kernels to support the operators for training. For CPU, we implemented these kernels in C++ using OpenMP for inference which uses AVX2 vector instructions.

Evaluation Task. For empirical evaluations, we use RoBERTa language modeling pretraining (Liu et al., 2019a) as our evaluation tool to measure the method performance, since it is a challenging task. The models are pretrained using masked language modeling (Devlin et al., 2019) on the English Wikipedia corpus. We pretrain each model for 250K steps with a batch size of 256, where each sequence is of length 512. We use an Adam optimizer with 1e-4 learning rate, 10,000 warm-up steps, and linear decay. To keep compute reasonable, we use RoBERTa with 4 layers and 512 embedding dimensions for model evaluation except one stress-test experiment checking the scaling behavior of LookupFFN to a full size RoBERTa-base.

4.1. Performance Comparison

Comparing to Baselines. We compare our method to Vanilla FFN, Slide (Chen et al., 2020), Mongoose (Chen et al., 2021), and YOSO (Zeng et al., 2021) based FFNs discussed previously. Additionally, for comparison to more baselines, we include Switch Transformer (Fedus et al., 2022) and the grouped convolution + channel shuffling introduced in ShuffleNet. Others have identified that the original implementation of Slide, which is implemented from scratch in C++, is difficult to be adopted (Chen et al., 2021), and have propose optimized variants, which we use here (Chen et al., 2021). Instead of each instance in a batch retrieving its own subset of weights, the union of the retrieved subsets is used for computation. This strategy is used to avoid an irregular workload due to the skewed bucket distribution, as discussed earlier. The size of $\mathcal{S}(\cdot)$ is larger for larger batches. We note that in a Transformer model, the effective batch size for a FFN is the number of sequences $\times$ the sequence length. The union of retrieved subsets will simply contain the entire set of weights. For a more reasonable size of set $\mathcal{S}(\cdot)$, we partition the effective batch into smaller mini-batches (128 tokens for Slide and 2048 tokens for Mongoose) and feed them into the FFN sequentially. This would severely increase the runtime of training on GPUs. The size of mini batch is set such that it is small enough but running the experiment is still feasible. Further, since the performance of Slide (Chen et al., 2020), Mongoose (Chen et al., 2021), and YOSO (Zeng et al., 2021) based FFN largely depends on the frequency of rehashing, we perform rehashing after every parameter updates (this overhead is not counted towards FLOP) to ensure their optimal performance. The results are summarized in Tab. 1. Our method achieves lower perplexity using fewer FLOP compared to the baselines. Further, the FLOP of our method can be significantly reduced with some loss in performance (but still better than the baselines except for the VanillaFFN) as shown in the last row of Tab. 1. Additional results are discussed in §4.2.

Downstream finetuning. Further, we evaluate the quality of the pretrained language models for VanillaFFN and our

LookupFFN on MNLI downstream task (Williams et al., 2018) in the GLUE benchmark (Wang et al., 2019). The result is shown in Tab. 2. We note that there is a small gap between our method and vanilla FFN, but the FLOP of our method is much lower.

Scaling to Full Size Models. We check whether LookupFFN scales to a larger model, so we evaluate our method on a RoBERTa-base pretraining. All pretraining hyper-parameters remain the same as before. Due to the compute burden of training a full size model, we only perform one experiment comparing LookupFFN with $h = 170$ and $\tau = 9$ to VanillaFFN. The results are shown in Tab. 4. Our method achieves $6.8\times$ FLOP reduction while the log perplexity is only higher by 0.04 in a RoBERTa-base model.

Memory use. Our LookupFFN requires more memory (for large h and τ) since we directly parameterize the hash tables, but we believe this is not a key issue. In a GPU setting, memory use is critical since the GPU memory is usually much more expensive and limited. On the other hand, CPU memory is much cheaper and larger compared to GPU memory.

4.2. Ablation Study

Reducing FLOP for Hash. We note that the projection in the Hash step has a complexity of $O(h\tau d)$ when using a vanilla dense projection and will generate the majority of FLOP for our LookupFFN. In §3.2, we propose an efficient alternative, BH4 and verify that the block size of B_i has a direct impact on the representation power of BH4. But will this impact the final performance of a model? To evaluate the trade-off between efficiency and performance, we study the effect of block size of B_i on the log perplexity of RoBERTa. The results are summarized in the top table of Tab. 3. When using a vanilla projection, the FLOP for the Hash step accounts for 89% of the total FLOP. It is critical to reduce the FLOP need for this projection, else it becomes the main bottleneck. When using BH4, the performance decreases and efficiency increases as the block size decreases, which is expected, but it is surprising that while offering a large FLOP reduction, the performance drop compared to the vanilla dense projection is quite small.

Scaling Effect of LookupFFN. The number of hash tables h and the length of the hash code τ (or log of table size) control the scaling behavior. As a preliminary step, we first verify that our method can indeed be scaled for better accuracy by increasing the number of hash tables h. Therefore, we compare the model perplexity for different h when τ is fixed, as shown in Tab. 3 (middle). The model performance monotonically increases as h increases. When $h = 256$, our method achieves a lower perplexity compared to the VanillaFFN in Tab. 1. Then, we evaluate the effect of scaling τ. Instead of fixing h while scaling τ, we increase τ but at the same time reduce h such that $h\tau$ is roughly the same. As shown in Tab. 3 (bottom), this comparison reveals a surprising scaling effect of our method: when $h\tau$ is fixed (the FLOP of Hash step remains the same), by increasing τ (increasing the table size), we can reduce the number of hash tables and reduce the FLOP count for the Gather step while achieving better performance.

Technique	Avg. Latency (ms)	Speedup
VanillaFFN	403	$1.00\times$
SLIDE	428	$0.94\times$
Mongoose	878	$0.46\times$
LookupFFN	328	$1.23\times$
LookupFFN (Opt1)	160	$2.51\times$

Table 5. Average latency for LookupFFN compared to baselines.

4.3. Throughput and Latency Study

In this subsection, we isolate the FFN from RoBERTa to examine the runtime and throughput of each style of FFN. We compare throughput of LookupFFN, Slide and Mongoose to VanillaFFN (YOSO is excluded since it does not have a CPU implementation at this time), then provide an in-depth analysis of LookupFFN's hardware-level behavior and potential for future throughput scalability. For all empirical results here, we use batch size 64 and sequence length 512, so the effective batch size is 32768 (side note: multi-head attention takes 274ms in this setting).

Latency Comparison. In order to demonstrate the latency improvement afforded by our LookupFFN for CPU inference, we compare runtime to alternatives. Tab. 5 shows the average per-iteration time for vanilla, Slide, and Mongoose-based FFN which is sized to match typical hyperparameters for a standard Transformer model on a modern AMD EPYC-7452 (Zen 2) 32-core Server. Our basic implementation for LookupFFN uses OpenMP without additional software-engineering optimization along with a naive implementation of BH4 and achieves 23% speedup over VanillaFFN. We did further optimization of both the hash function and gather operation (*opt1*), achieving $2.51\times$ speedup over VanillaFFN. Overall we see that both Slide and Mongoose perform worse than vanilla, while our optimized LookupFFN provides good performance improvement.

Discrepancy between FLOP and Latency. We note that there is difference in the speed up between FLOP and Latency in Tab. 1 and Tab. 5. The latency not only depends on FLOP, but also the memory access pattern or arithmetic intensity. The GEMM used in VanillaFFN has a very structured memory access pattern and its arithmetic intensity is high, but the gather operator used in LookupFFN has a more random memory access pattern that depends on the input and its arithmetic intensity is lower. As a result, while the FLOP of LookupFFN is drastically lower than VanillaFFN,

LookupFFN		naive	opt1
Gather	Latency	100 ms	35 ms
	Compute Utilization	1.21%	3.53%
	Sustained L1 BW	79.7 GB/s	231.5 GB/s
	Sustained LLC BW	9.5 GB/s	52.5 GB/s
	L1 Miss %	11.87%	22.68%
	LLC Miss %	69.56%	12.77%
Hash Latency		208 ms	106 ms
Other Latency		20 ms	20 ms
Total Latency		328 ms	160 ms

Table 6. Analysis of Performance Characteristics for Lookup FFN. We keep volume of work and working set constant for both so instruction count and FLOPs are constant.

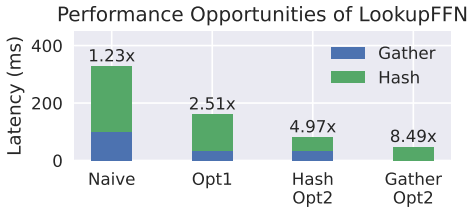

Figure 6. Future performance opportunities of Lookup FFN. Each bar shows the time breakdown of LookupFFN by GatherAdd and Hash performance. Each version of LookupFFN is annotated with its speedup over Vanilla FFN.

the corresponding speed up in latency is not as drastic.

Performance analysis. To further elucidate how we are able to achieve speedup, we detail architectural level performance statistics in Tab. 6 We notice the speedup primarily is afforded by achieving a higher sustained cache bandwidth. The lower LLC miss rate suggests this is at least in part due to extracting more reuse from the LLC, thereby making internal cache management including MSHRs, and on-chip network traffic perform better.

Future Performance Opportunities. Future performance gains for LookupFFN are possible through a combination of software and upcoming hardware optimizations which we summarize in Fig. 6. *naive* and *opt1* are our two configurations reported previously. Additional tuning of block size for the BH4 projection (64 to 16) could provide a $4\times$ speedup for our Hash step (*hash-opt2*). Future cache technology such as 3D stacking (Wuu et al., 2022) will likely provide a rather generous boost in high-speed LLC cache capacity, enabling larger table size LookupFFN weights to be entirely cache-resident. In combination with this, careful cache optimization through the use of modern prefetching techniques (Georganas et al., 2018), as well as hardware improvements such as Intel's wide 128 B L1 cache interface (Rotem et al., 2022) can improve hit rates and overall bandwidth. If 90% of bandwidth from a 128 B cache interface could be sustained, Gather step can be improved by a factor of $35\times$ (*gather-opt2*). Overall, these improvements could yield $8.49\times$ improvement over VanillaFFN. Datatype precision reduction could potentially afford a further multiplicative runtime improvement of $2\times$, achieved by switching from float32 to float16 – however, VanillaFFN would also gain a $2\times$ execution time reduction from float16.

Discussion of Compiler Techniques for FFN. Deep Learning specific compilers, such as TVM and XLA (built into TensorFlow) have been introduced, aiming to optimize operations such as a feed-forward network. We evaluated TVM and XLA on VanillaFFN to compare LookupFFN's latency to these two compiler frameworks. TVM performs 21% worse than PyTorch, with a latency of 488ms. Switching to TensorFlow gives $2.39\times$ improvement over the PyTorch VanillaFFN, with XLA yielding an additional 37% performance boost (absolute latency of TF+XLA: 123ms). Our optimized LookupFFN already provides competitive performance compared to TensorFlow, and our additional hash optimization, when fully implemented/integrated, LookupFFN will provide performance improvement over TF+XLA.

5. Conclusions

We conclude with a contemplative remark followed by a practical comment. Given the rapid improvements in compute capabilities of GPUs we have seen over the last decade (and a mature software support for different kernel shapes), there was little incentive for algorithm designers to use non-GEMM operations. In fact, any modern deep learning algorithm that is not heavily reliant on GEMM may have little chance of broader adoption, assuming it even sees the light of day. The ideas described here (and in Slide, Mongoose, YOSO), in some sense, lie at the other extreme. LookupFFN almost operates as if GEMM is forbidden. While compute capability improvements are slowing down, new memory technologies are already available and others in the development pipeline, we hypothesize that novel yet-undiscovered DNN architectures must hit a sweet-spot to delicately balance the trade-off between these two resources. Doing so also offers other benefits including potential energy savings. Now, specific to the model in this work, we expect that the LookupFFN benefits can translate to other DNN models. This will complement the server chip developments in §1, and enable AI models to serve a broader cross-section of industries.

Acknowledgments

This work was supported in part by funding from the Vilas Board of Trustees and UW–Madison Office of the Vice Chancellor for Research and Graduate Education.

References

AMD. Amd zen deep neural network (zendnn). URL https://www.amd.com/en/developer/zendnn.html.

Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I. P., and Schmidt, L. Practical and optimal LSH for angular distance. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pp. 1225–1233, 2015.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Bhat, A. Machine learning on - arm servers an update. linaro virtual connect 2021.

Burd, T., Li, W., Pistole, J., Venkataraman, S., McCabe, M., Johnson, T., Vinh, J., Yiu, T., Wasio, M., Wong, H.-H., Lieu, D., White, J., Munger, B., Lindner, J., Olson, J., Bakke, S., Sniderman, J., Henrion, C., Schreiber, R., Busta, E., Johnson, B., Jackson, T., Miller, A., Miller, R., Pickett, M., Horiuchi, A., Dvorak, J., Balagangadharan, S., Ammikkallingal, S., and Kumar, P. Zen3: The amd 2nd-generation 7nm x86-64 microprocessor core. In *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 65, pp. 1–3, 2022. doi: 10.1109/ISSCC42614.2022.9731678.

Charikar, M. Similarity estimation techniques from rounding algorithms. In Reif, J. H. (ed.), *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pp. 380–388. ACM, 2002. doi: 10.1145/509907.509965.

Charikar, M. and Siminelakis, P. Hashing-based-estimators for kernel density in high dimensions. In Umans, C. (ed.), *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pp. 1032–1043. IEEE Computer Society, 2017. doi: 10.1109/FOCS.2017.99.

Chen, B., Medini, T., Farwell, J., gobriel, s., Tai, C., and Shrivastava, A. Slide : In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 291–306, 2020. URL https://proceedings.mlsys.org/paper/2020/file/65b9eea6e1cc6bb9f0cd2a47751a186f-Paper.pdf.

Chen, B., Liu, Z., Peng, B., Xu, Z., Li, J. L., Dao, T., Song, Z., Shrivastava, A., and Re, C. {MONGOOSE}: A learnable {lsh} framework for efficient neural network training. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=wWK7yXkULyh.

Cheng, Y., Yu, F. X., Feris, R. S., Kumar, S., Choudhary, A., and Chang, S.-F. An exploration of parameter redundancy in deep networks with circulant projections. In *Proceedings of the IEEE international conference on computer vision*, pp. 2857–2865, 2015.

Choromanski, K. M., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlos, T., Hawkins, P., Davis, J. Q., Mohiuddin, A., Kaiser, L., Belanger, D. B., Colwell, L. J., and Weller, A. Rethinking attention with performers. In *International Conference on Learning Representations (ICLR)*, 2021.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Dong, X. and Yang, Y. Network pruning via transformable architecture search. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/a01a0380ca3c61428c26a231f0e49a09-Paper.pdf.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL http://jmlr.org/papers/v23/21-0998.html.

Georganas, E., Avancha, S., Banerjee, K., Kalamkar, D., Henry, G., Pabst, H., and Heinecke, A. Anatomy of high-performance deep learning convolutions on simd architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

Hendrycks, D. and Gimpel, K. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Horowitz, M. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, 2014. doi: 10.1109/ISSCC.2014.6757323.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Intel. Intrinsics for intel advanced matrix extensions (intel(r) amx) instructions. intel c++ compiler classic developer guide and reference.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

Jouppi, N. P., Hyun Yoon, D., Ashcraft, M., Gottscho, M., Jablin, T. B., Kurian, G., Laudon, J., Li, S., Ma, P., Ma, X., Norrie, T., Patil, N., Prasad, S., Young, C., Zhou, Z., and Patterson, D. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, 2021. doi: 10.1109/ISCA52012.2021.00010.

Kim, S., Gholami, A., Yao, Z., Mahoney, M. W., and Keutzer, K. I-bert: Integer-only bert quantization. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 5506–5518. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/kim21d.html.

Kitaev, N., Kaiser, L., and Levskaya, A. Reformer: The efficient transformer. In *International Conference on Learning Representations (ICLR)*, 2020.

Le, Q., Sarlos, T., and Smola, A. Fastfood - approximating kernel expansions in loglinear time. In *30th International Conference on Machine Learning (ICML)*, 2013. URL http://jmlr.org/proceedings/papers/v28/le13.html.

Lee, N., Ajanthan, T., and Torr, P. SNIP: Single-shot network pruning based on connection sensitivity. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=B1VZqjAcYX.

Lichtenau, C., Buyuktosunoglu, A., Bertran, R., Figuli, P., Jacobi, C., Papandreou, N., Pozidis, H., Saporito, A., Sica, A., and Tzortzatos, E. Ai accelerator on ibm telum processor: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, pp. 1012–1028, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450386104. doi: 10.1145/3470496.3533042. URL https://doi.org/10.1145/3470496.3533042.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach, 2019a. URL https://arxiv.org/abs/1907.11692.

Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V., and Wang, Y. Optimizing cnn model inference on cpus. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pp. 1025–1040, USA, 2019b. USENIX Association. ISBN 9781939133038.

Mittal, S., Rajput, P., and Subramoney, S. A survey of deep learning on cpus: Opportunities and co-optimizations. *IEEE Transactions on Neural Networks and Learning Systems*, 33(10):5095–5115, 2022. doi: 10.1109/TNNLS.2021.3071762.

Moczulski, M., Denil, M., Appleyard, J., and de Freitas, N. ACDC: A structured efficient linear layer. In Bengio, Y. and LeCun, Y. (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.05946.

Nassif, N., Munch, A. O., Molnar, C. L., Pasdast, G., Lyer, S. V., Yang, Z., Mendoza, O., Huddart, M., Venkataraman, S., Kandula, S., Marom, R., Kern, A. M., Bowhill, B., Mulvihill, D. R., Nimmagadda, S., Kalidindi, V., Krause, J., Haq, M. M., Sharma, R., and Duda, K. Sapphire rapids: The next-generation intel xeon scalable processor. In *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 65, pp. 44–46, 2022. doi: 10.1109/ISSCC42614.2022.9731107.

Nori, A. V., Bera, R., Balachandran, S., Rakshit, J., Omer, O. J., Abuhatzera, A., Kuttanna, B., and Subramoney, S. Reduct: Keep it close, keep it cool! : Efficient scaling of dnn inference on multi-core cpus with near-cache compute. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 167–180, 2021. doi: 10.1109/ISCA52012.2021.00022.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An

imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Rotem, E., Yoaz, A., Rappoport, L., Robinson, S. J., Mandelblat, J. Y., Gihon, A., Weissmann, E., Chabukswar, R., Basin, V., Fenger, R., Gupta, M., and Yasin, A. Intel alder lake cpu architectures. *IEEE Micro*, 42(3):13–19, 2022. doi: 10.1109/MM.2022.3164338.

Spring, R. and Shrivastava, A. A new unbiased and efficient class of lsh-based samplers and estimators for partition function computation in log-linear models. *arXiv preprint arXiv:1703.05160*, 2017.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. GLUE: A multi-task benchmark and analysis platform for natural language understanding. 2019. In the Proceedings of ICLR.

Williams, A., Nangia, N., and Bowman, S. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 1112–1122, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1101. URL https://www.aclweb.org/anthology/N18-1101.

Wuu, J., Agarwal, R., Ciraula, M., Dietz, C., Johnson, B., Johnson, D., Schreiber, R., Swaminathan, R., Walker, W., and Naffziger, S. 3d v-cache: the implementation of a hybrid-bonded 64mb stacked cache for a 7nm x86-64 cpu. In *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 65, pp. 428–429, 2022. doi: 10.1109/ISSCC42614.2022.9731565.

Xiong, Y., Zeng, Z., Chakraborty, R., Tan, M., Fung, G., Li, Y., and Singh, V. Nyströmformer: A nyström-based algorithm for approximating self-attention. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

Yang, Z., Moczulski, M., Denil, M., De Freitas, N., Song, L., and Wang, Z. Deep fried convnets. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1476–1483, 2015. doi: 10.1109/ICCV.2015.173.

Zafrir, O., Boudoukh, G., Izsak, P., and Wasserblat, M. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*, pp. 36–39, 2019. doi: 10.1109/EMC2-NIPS53020.2019.00016.

Zaheer, M., Guruganesh, G., Dubey, K. A., Ainslie, J., Alberti, C., Ontanon, S., Pham, P., Ravula, A., Wang, Q., Yang, L., and Ahmed, A. Big bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Zeng, Z., Xiong, Y., Ravi, S., Acharya, S., Fung, G. M., and Singh, V. You only sample (almost) once: Linear cost self-attention via bernoulli sampling. In *International Conference on Machine Learning (ICML)*, 2021.

Zeng, Z., Pal, S., Kline, J., Fung, G. M., and Singh, V. Multi resolution analysis (MRA) for approximate self-attention. In Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., and Sabato, S. (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 25955–25972. PMLR, 17–23 Jul 2022.

Zhang, C., Yu, M., Wang, W., and Yan, F. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pp. 1049–1062, USA, 2019. USENIX Association. ISBN 9781939133038.

Zhang, X., Zhou, X., Lin, M., and Sun, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.