DECA: A Near-Core LLM Decompression Accelerator Grounded on a 3D Roofline Model

Gerasimos Gerogiannis
Intel, University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
gg24@illinois.edu

Stijn Eyerman
Intel
Brussels, Belgium
stijn.eyerman@intel.com

Evangelos Georganas
Intel Labs
Santa Clara, California, USA
evangelos.georganas@intel.com

Wim Heirman
Intel
Brussels, Belgium
wim.heirman@intel.com

Josep Torrellas
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
torrella@illinois.edu

Abstract

To alleviate the memory bandwidth bottleneck in Large Language Model (LLM) inference workloads, weight matrices are stored in memory in quantized and sparsified formats. Hence, before tiles of these matrices can be processed by in-core generalized matrix multiplication (GeMM) hardware engines, they need to be dequantized and de-sparsified. This is currently performed in software with vector operations. Unfortunately, this approach delivers only modest performance. Moreover, it is hard to understand how to improve the system, as the overall GeMM performance depends on the interaction between memory resources, vector units, and hardware matrix engines.

This paper addresses the problem of improving the performance of these environments. For this, it first develops the novel *Roof-Surface* performance model, a 3D model that provides clear insights into how memory resources, vector units, and hardware matrix engines interact to deliver compressed GeMM performance. Then, it proposes *DECA*, a new near-core ML-model decompression accelerator. DECA offloads tile de-sparsification and dequantization from the CPU, producing ready-to-use tiles for in-core GeMM engines. Finally, it introduces a new ISA extension that enables out-of-order invocation of the near-core accelerator. With this extension, accelerator and core computations can interleave and overlap with high performance. Our evaluation shows that, in a simulated 56-core Xeon 4 server with HBM, DECA accelerates the execution of compressed GeMMs by up to 4x over the use of optimized Intel software kernels. Further, DECA reduces the next-token generation time of Llama2-70B and OPT-66B by 1.6×—1.9×.

CCS Concepts

• **Computer systems organization → Multicore architectures**; **Special purpose systems**; **Neural networks**.

Keywords

Large Language Models, LLM, Inference, CPU, Hardware Accelerator, Quantization, Sparsity, Roofline Model

ACM Reference Format:
Gerasimos Gerogiannis, Stijn Eyerman, Evangelos Georganas, Wim Heirman, and Josep Torrellas. 2025. DECA: A Near-Core LLM Decompression Accelerator Grounded on a 3D Roofline Model. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3725843.3756073

1 Introduction

Large Language Models (LLMs) are an important Machine Learning (ML) workload, excelling at tasks such as chatbots, translation, text summarization, and content creation [44, 81, 84, 89]. LLMs use transformers [73] and mainly consist of multi-head attention and fully connected (FC) layers. The largest models contain trillions of parameters (weights) in the FC layers [1, 87]. During inference, the batch sizes are typically small and weights have low reuse—stressing the memory bandwidth of modern platforms [82].

GPUs are regarded as the standard platform for LLM inference because of their high compute and memory bandwidth. However, recent advances introduced by Intel Xeon 4 servers (codenamed Sapphire Rapids (SPR)) [5], make CPUs an attractive option for LLM inference. Such processors are equipped with an in-core generalized matrix multiplication (GeMM) engine called TMUL [37]. The TMUL serves the same purpose as the GPU Tensor Cores [53]. It is programmed with the AMX ISA extensions [37] to perform GeMMs on matrix tiles. The result is an order of magnitude increase in GeMM computational throughput compared to relying solely on vector SIMD units. Further, SPR servers can be equipped with High Bandwidth Memory (HBM), increasing the available memory bandwidth by 3-4× over their DDR-based counterparts.

In SPR CPUs, we observe that, similar to GPUs [82], LLM inference is memory-bandwidth bound. The large GeMMs in the FC layers account for more than 90% of the next token generation time for LLama2-70B [72]. Such GeMMs have low arithmetic intensity and require loading a large number of weights from main memory. To a large extent, accelerating LLM inference on CPUs means speeding-up these large GeMMs.

Compressing deep neural network (DNN) models [13, 47] with techniques such as low-bit weight quantization [21] and sparsification/pruning [34, 79, 90] can improve GeMM performance. The amount of data that needs to be loaded from memory is reduced, leading to significant speedups in memory-bound kernels. Sadly, like systolic arrays [41] and Tensor Cores [79], the TMUL cannot handle arbitrary quantization schemes or sparse patterns. It expects well-formed dense input tiles (i.e., zero values must be included), either in BF16 [43] or INT8 format.

To benefit from both model compression and TMUL GeMM throughput, Intel has recently introduced specialized kernels in the libxsmm framework [32]. Libxsmm uses a sequence of vector instructions (AVX) to read compressed tiles from memory, de-sparsify and/or dequantize them, and feed them to the TMUL AMX unit. This cooperative processing mode involves two different computational domains (vector and matrix), each with its own instructions (AVX and AMX), and functional units (SIMD units and TMUL).

We profiled the performance of the libxsmm kernels for different quantized and sparsified workloads. Our analysis shows that, although these kernels are very effective for moderately compressed GeMM data and with the relatively low-bandwidth DDR memory, their performance degrades with HBM. This degradation cannot be explained using a traditional two-dimensional (2D) roofline performance model [78] that only considers the memory bandwidth and the TMUL throughput as bounding factors.

To understand how to effectively accelerate kernels involving three different resources (memory, matrix, and vector resources), we introduce a novel three-dimensional (3D) performance model that we call *Roof-Surface*. This model constructs a roof-surface separating achievable from non-achievable performance. It offers greater performance explainability than its traditional 2D roofline counterpart, accurately attributing the libxsmm performance limitation to the AVX vector decompression sequence. The Roof-Surface model suggests that overcoming the decompression inefficiencies would require a prohibitive scaling of CPU core resources.

To address this problem, this paper proposes *DECA*, a new near-core *accelerator of ML model decompression*. DECA offloads tile de-sparsification and dequantization from the CPU, producing ready-to-use tiles for the TMUL. DECA can be programmed to handle quantized number formats with any number of bits between 1 and 8, supports any level of unstructured sparsity, and supports group quantization [21]. The DECA microarchitecture performs decompression by utilizing a pipeline with advanced vector operations. Importantly, we use the Roof-Surface model to: (1) make decisions about the vector pipeline microarchitecture and (2) perform design space exploration and derive a well-balanced DECA design.

We also observe that if CPU cores use regular memory-mapped load/store instructions to communicate with DECA, the communication latency gets exposed and hurts performance. Consequently, we introduce a new ISA extension to CPU cores that hides the CPU-DECA communication latency by invoking DECA out-of-order. We call this extension *Tile External Preprocess and Load* (TEPL).

Our evaluation for two different low-bit quantization formats (BF8 and MXFP4) and different unstructured sparsity levels shows that DECA is very effective. In a simulated 56-core SPR with HBM, DECA accelerates the execution of compressed GeMMs by up to 4x over the optimized Intel libxsmm software kernels. In addition, by speeding-up the FC layers, DECA reduces the next-token generation time of Llama2-70B and OPT-66B [86] by 1.6×—1.9× over the software-only solution, and by 2.5×—3.3× over the uncompressed baseline model.

This paper's contributions are:
• The *Roof-Surface* 3-D performance model that models the interaction between vector units, matrix units, and memory. Its use extends beyond CPUs and LLM decompression.
• The *DECA* near-core accelerator, designed to accelerate the de-sparsification and dequantization of compressed ML models.
• The *Tile External Preprocess & Load (TEPL)* extension that enables out-of-order invocation of near-core accelerators.
• A simulation-based evaluation of the performance of *DECA* for compressed GeMMs and LLM inference.

2 Background

2.1 LLM Inference

Large Language Models (LLMs) consist of different layers, such as Embedding layers, Fully-Connected (FC) layers, and Attention layers [73]. LLM inference has two phases [60]. The first one encodes the input tokens and generates the first token (prompt phase). The second one generates the next output tokens (generation phase). In this work, we focus on executing the low arithmetic-intensity generation phase efficiently since, for many practical use cases, it dominates the end-to-end LLM inference time [82].

GPUs are regarded as the standard platform for LLM inference [60, 71] because of their high compute and memory bandwidth. However, there has been increasing research and industrial interest in making CPUs better at machine learning (ML), by either incorporating extensions or small accelerators on the CPU die [23, 24, 39, 54, 57, 66]. In particular, recent advances such as HBM and in-core GeMM engines [5] make CPUs attractive for LLM inference. For this reason, in this paper, we focus on LLM inference on modern CPU servers. We mainly target batch sizes from 1 to 16, which are typical in local interactive inference. Larger batch sizes in CPUs increase the inference latency without proportional gains in throughput (Appendix 12.1).

2.2 Model Compression

For low arithmetic-intensity LLM FC layers, compressing the weight matrices reduces data movement and, therefore, can directly improve performance in both GPUs and CPUs. There are two main ways to compress an ML model [28, 29, 90]:
• **Quantization** involves storing weights in a lower-bit format, e.g., FP8 or FP4 instead of FP16. Multiple quantization schemes exist [45, 48, 75, 88]. Some of them additionally split weights in groups and introduce a per-group scaling factor (i.e., *group quantization*) to achieve higher accuracy. In our work, we evaluate two types of weight quantization: BF8 (8-bit brain floating point) and MXFP4 [63]. The latter uses a 4-bit floating point and group quantization with a shared scaling factor for every 32 weights. MXFP4 has been shown to not degrade LLM accuracy [63].
• **Sparsification** consists of eliminating (*i.e., pruning*) weights that do not contribute much to the model's accuracy [6, 34, 46]. *Unstructured sparsity* does not impose restrictions on which weights can be

removed, while *structured sparsity* requires the sparsity to follow a certain pattern. The former achieves higher accuracy for the same sparsity level [15, 49]. In this work, we assume unstructured sparsity, and use a bitmask-based sparse format to avoid storing zeros. To reconstruct the position of the nonzeros in the original weight matrix, the bitmask has as many bits as the number of elements in the matrix, where the '1' bits indicate the location of the nonzeros. The nonzeros are stored consecutively in a nonzero array. Recently proposed LLM weight pruning methods such as SparseGPT [15] have achieved unstructured sparsity levels of up to 60-70% without significant loss in accuracy. For traditional ML models like ResNet50 [30], sparsity of up to 95% is easy to achieve [62].

Models may be both sparse and quantized [29]. Let us define the density factor d as the fraction of weights that are nonzeros. Then, starting from a dense BF16 model, a Q bit quantized model with density factor of d reduces the model size by a factor of $16/(Q \times d + 1)$, where the '1' comes from the bitmask bit. We refer to this factor as the *compression factor*—as we assume that the footprint of activations is negligible. Later, to decompress a sparse and quantized model, one must read the compressed weights, the bitmasks, and the scaling factors (if group quantization is used). These three data structures are stored as separate arrays in memory.

The compression process is executed offline (e.g., after training). In this paper, we assume an already compressed model that we want to use online for inference.

2.3 Matrix Extensions

There are several hardware extensions to improve the efficiency of matrix multiplication on CPUs [4, 12, 35, 77]. In this work, we use Intel's Advanced Matrix Extensions (AMX) [35]. AMX extends a core's registers with 8 matrix registers, called *Tile Registers*. Each one can hold up to 16 rows, with 64 bytes of data per row that can be interpreted as 32 2-byte elements (BF16) or 64 1-byte elements (INT8). Each tile register can contain up to 1 KB of data.

Each core has the tile registers and a matrix multiplication (TMUL) unit. To load/store data to/from the tile registers, AMX includes tload/tstore instructions. The TMUL multiplies tiles in the tile registers. For example, assume that an LLM generation phase uses a batch size of $N \leq 16$ and BF16. A tile register can contain a weight tile W with $M = 16$ rows and $K = 32$ columns. Another tile register can contain an activation tile A with N rows and $K = 32$ columns. The TMUL can then perform the operation $A \times W^T$, to produce an NxM output tile. A TMUL operation can be launched every 16 cycles, regardless of the N value. It performs a total of $N \times K \times M = N \times 32 \times 16 = 512N$ fused multiply-adds (FMAs)—or equivalently $32N$ FMAs per cycle. For N>16, the TMUL throughput saturates at 512 FMAs per cycle, since the activation tile can hold no more than 16 rows. Any mention of FLOPs in this work refers to FMAs.

2.4 GeMM Decompression

The TMUL does not handle sparse data and, similar to other GeMM engines [41, 53], can only handle data in very specific data formats (i.e., BF16 and INT8). If a GeMM contains compressed weights, decompression is needed to produce tiles that conform to the TMUL requirements. Unlike compression, decompression is performed

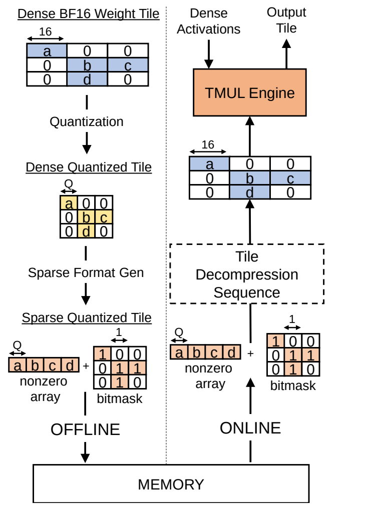


Figure 1: Weight offline compression and online decompression.

```
............
//Decompress Ti+1
for(r=0 to 16):
{
//Decompress
//row r of Ti+1
//using AVX VectorOps
...
}

//GeMM Ti
//using AMX MatrixOps
TLoad Ti
TComp Tout, Ti

//Decompress Ti+2
//GeMM Ti+1
............
```

Figure 2: Libxsmm compressed GeMM kernel pseudocode.

online. Thus, it can impact performance. Figure 1 shows the process of offline compression of weights and their online decompression.

To partially hide the decompression overhead and attain high performance in compressed GeMM kernels, Intel recently introduced a software solution integrated in the Libxsmm framework [32]. As shown in Figure 2, the solution involves a decompression sequence handled using AVX vector operations, and a GeMM operation executed using AMX matrix operations. Libxsmm adopts a smart method to overlap the execution of the two: software allocates a double software buffer, and tries to keep it in the L1 cache. The output of the AVX decompression sequence for tile *Ti+1* is written in one of the two software buffers. At the same time, AMX instructions load data from the other software buffer that contains *Ti*, which has been previously decompressed by the AVX sequence. The execution of AMX and AVX instructions overlap in the pipeline, while dependencies are naturally honored.

The decompression sequence uses vector operations such as permutes for the decompression, and masked vector expands to insert zeros in the appropriate positions of the nonzero array. Although we omit the specifics, the first takeaway is that decompression is done using AVX, utilizing a different "domain" (i.e., separate instructions and functional units) than AMX. The second takeaway is that the AVX dynamic instructions vastly outnumber the AMX ones, since AMX uses tile-sized operands (1KB), while AVX operates on cache-line sized ones (64B, one tile row). Overall, dozens of dynamic AVX instructions are needed per AMX instruction. As a result, the decompression loop increases the total dynamic instructions of the kernel by more than an order of magnitude.

3 Motivation

3.1 GeMMs in FC Layers Dominate Inference

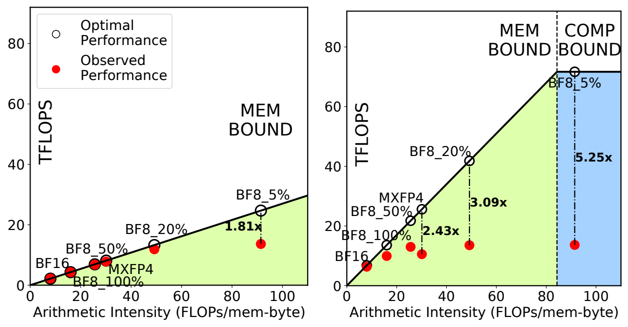
Table 1 shows the fraction of the "next-token" time spent in the GeMMs of the different Fully Connected (FC) layers of Llama2-70B [72] on an SPR server with either DDR5 or HBM. We show results for an uncompressed model with BF16 weights, with different numbers of input tokens and batch sizes (N). The rest of the time is spent on kernels such as attention, for which weight compression does not apply. We see that the time spent in such GeMMs is over 95% for DDR5 and 85–90% for HBM. Hence, accelerating these GeMMs can greatly improve the next-token time.

Table 1: Contribution of FC layer GeMMs to next-token time.

Memory	DDR (260GB/s)		HBM (850GB/s)	
Input Tokens	32	128	32	128
Batch size (N): 1	97.4%	97.5%	89.8%	89.5%
4	97.3%	97.1%	89.4%	88.9%
16	96.6%	95.5%	88.3%	85.9%

3.2 GeMMs in FC Layers are Bandwidth Bound

We show the roofline models for one of the large GeMMs of the FC layers in LLama2-70B for an SPR with either DDR5 (Figure 3a) or HBM (Figure 3b), and N=16. We use the TMUL FLOPS limit (Section 2.3) for the maximum achievable GeMM FLOPS in the compute-bound area. Further, when calculating the Arithmetic Intensity (AI) in FLOPs per memory byte, we only consider the footprint of the weight matrices. We neglect the footprint of the activation matrices because it is much smaller than the one of the weight matrices for small values of N. In both graphs, the leftmost circle labeled as 'BF16' is our baseline uncompressed execution. We see that this execution is memory-bandwidth bound in both cases due to a low AI. This motivates model compression, to reduce the amount of data that needs to be read from memory.



(a) 56-core SPR with DDR5 (b) 56-core SPR with HBM

Figure 3: Traditional rooflines for a GeMM with N=16.

3.3 Compressed GeMMs can be Inefficient

The other data points in Figure 3 represent compressed models with 4-bit quantization (MXFP4) or with 8-bit quantization (BF8) with density levels (i.e., the fraction of nonzeros) ranging from 5% to 100%. For example, BF8_20% is for BF8 and density 20%. As the compression factor increases, the amount of data fetched from memory decreases, and the AI goes up. As a result, the circles move to the right. For each design point, we show two circles: one at the *Observed* performance, and one on the roofline for the same AI. We call the latter *Optimal* performance.

We see that, as we increase the compression factor, the Observed and Optimal points increasingly diverge. In the DDR5 graph, the divergence appears at BF8_5%. However, in the HBM graph, all the compressed models are below their Optimal performance; at BF8_5%, the ratio between Optimal and Observed performance is 5.25x. This means that performance is limited by some inefficiency that is not captured by the roofline model. By manual profiling, we find that the root cause is the overhead of the AVX decompress instruction sequence. Effectively, the AVX SIMD processing units of the cores are unable to keep up with the memory bandwidth and/or the throughput of TMUL.

Given the importance of LLM workloads, some form of hardware support to minimize the decompression overhead could be justified. However, one has to be cautious when making changes in the resource-constrained CPU setting. The roofline model does not tell us the required vector throughput improvement needed for the kernels to shift from being bounded by vector processing to being bounded by memory or matrix computation. There is a danger of constructing hardware solutions that are either under-provisioned or overprovisioned. To avoid this danger, we need a new model that can guide the system design to efficiently eliminate the decompression overhead in compressed GeMMs.

4 The Roof-Surface Model

To guide the system design for kernels that involve matrix, vector, and memory operations, we develop a new performance model that captures their interaction. This model is called *Roof-Surface* and has a three-dimensional (3D) visualization.

4.1 The 3D Roof-Surface Performance Model

When multiple interacting factors can affect performance, the slowest factor ends up determining the performance. In our case, the factors are memory, vector, and matrix operations. To understand which one is the slowest, we should first express how fast (1) memory can provide compressed tiles (MEM), (2) vector hardware can process compressed tiles (VEC), and (3) matrix hardware can process decompressed tiles (MTX).

Memory can provide compressed tiles at a rate of $MBW/Bytes_{tile}$ tiles per second, where MBW is the memory bandwidth in bytes per second and $Bytes_{tile}$ is the number of bytes in a compressed tile. Since a compressed weight tile will be used for a single TMUL matrix operation, we refer to $1/Bytes_{tile}$ as the matriX-to-Memory arithmetic intensity or AI_{XM}. It expresses how many matrix operations can be executed per byte loaded from memory, and it is very similar to the traditional arithmetic intensity used in the rooflines of Figure 3. The main difference is that its units are matrix operations per byte and not FLOPs per byte. In our setting, compression schemes with higher compression factors (Section 2.2) have a higher AI_{XM}. Overall, the MEM rate in compressed tiles per second is $MBW * AI_{XM}$.

The Vector Hardware decompresses tiles at a rate of VOS/VO_{tile}, where VOS is the number of vector operations per second that can be executed by the architecture, and VO_{tile} is the number of vector operations needed per tile. VOS is the vector throughput and is an architecture-dependent parameter. For example, for our SPR system, it is given by the product of the processor frequency (f), the number of cores (c), and the number of SIMD units per core. VO_{tile} is a kernel-dependent parameter. Since only the weight matrix in a

GeMM needs to be decompressed, VO_{tile} effectively expresses how many vector operations are needed per matrix operation. We refer to $1/VO_{tile}$ as the matriX-to-Vector arithmetic intensity or AI_{XV}, since it expresses how many matrix operations can be executed per vector operation. Overall, the VEC rate in tiles per second is $VOS * AI_{XV}$.

The Matrix Hardware can perform *MOS* matrix operations per second. *MOS* depends on the architecture and not on the kernel. For example, in SPR systems, it is given by $f * c/16$, since each core has a TMUL that can perform a tile multiplication every 16 cycles. Overall, the MTX rate in tiles per second is simply *MOS*.

The Final Performance is determined by the lowest tile processing rate among the three rates considered. Specifically, the number of tiles per second (*TPS*) that the architecture can process is:

$$TPS = min\{MBW * AI_{XM}, VOS * AI_{XV}, MOS\} \quad (1)$$

We can easily get the rate of FLOPs per second (*FLOPS*) by recalling from Section 2.3 that a TMUL tile operation corresponds to $512{*}N$ FMAs. Thus:

$$FLOPS = 512 * N * min\{MBW * AI_{XM}, VOS * AI_{XV}, MOS\} \quad (2)$$

We call this equation the *Roof-Surface* equation. Any of the three terms inside the *min* clause can be the one limiting performance. For a given architecture (i.e., fixed *MBW*, *VOS*, and *MOS*), there are *two kernel-dependent variables* inside the *min* clause: AI_{XM} and AI_{XV}. These are the kernel's "signature"—if two kernels have the same signature, they have the same projected performance. In contrast, in the roofline model, the kernel signature is just one variable: the traditional FLOP-to-memory AI. Now, the illustration of the performance model can no longer be done in the two dimensions of Figure 3: FLOPS as a function of FLOP-to-memory AI. Instead, we need three dimensions: FLOPS (z dimension) as function of the AI_{XM} (x dimension) and AI_{XV} (y dimension).

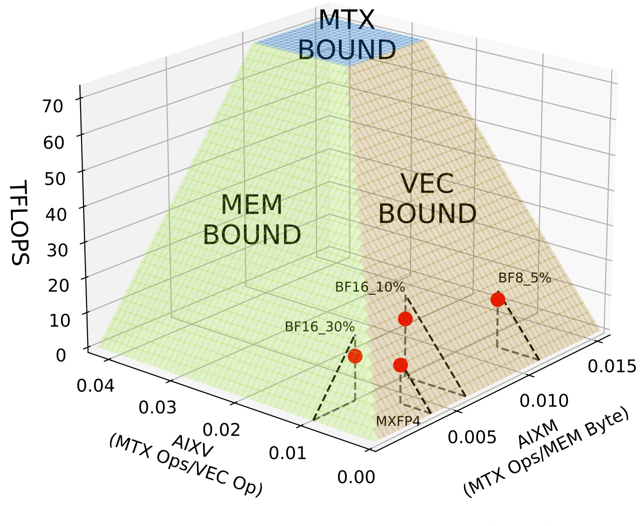


Figure 4: The 3D Roof-Surface model (N=16).

Figure 4 shows the result of plotting Equation 2 (for N=4,HBM) in three dimensions to form the *Roof-Surface* plot. A Roof-Surface plot has three regions, depicted in different colors. In each of the regions, a different term of the Roof-Surface Equation is the smallest one, and thus bounds performance. The operation points below the blue subsurface are bound by the MTX factor, the ones below the green subsurface are bound by the MEM factor, and the ones below the orange subsurface are bound by the VEC factor. Kernel performance is depicted by points in the 3D space. The achievable performance is bounded by the overall surface, rather than by a line like in the roofline model. For this reason, we call the model Roof-Surface. Points above the overall surface are not achievable.

Figure 4 shows four points that correspond to the observed performance of different compression schemes. We see that the MXFP4 and BF8_5% points are very near to the top of the corresponding tangent triangles (i.e., almost exactly on the orange subsurface). This visually reveals that they are bounded by vector operations. The other two points, namely BF16_30% and BF16_10%, are primarily bounded by memory bandwidth and vector operations, respectively. However, since these points are slightly below the roof surface, for these kernels, non-plotted factors such as memory latency or cache latency are also leaving some performance on the table.

Table 2: Performance limits predicted by the roofline (R-L) and Roof-Surface (R-S) models, and the real measured performance for kernels with different compressions in TFLOPS.

	MXFP4	BF8	BF8_50%	BF8_30%	BF8_20%	BF8_10%
R-L	25.2	13.3	21.2	31.2	40.8	59.2
R-S	11.5	13.3	16.1	16.1	16.1	16.1
Real	10.6	10.0	13.0	13.5	13.6	13.5

	BF8_5%	BF16_50%	BF16_30%	BF16_20%	BF16_10%	BF16_5%
R-L	70	11.8	18.4	25.2	40.8	59.2
R-S	16.1	11.8	18.4	23.0	23.0	23.0
Real	13.7	9.2	12.3	15.4	16.3	16.75

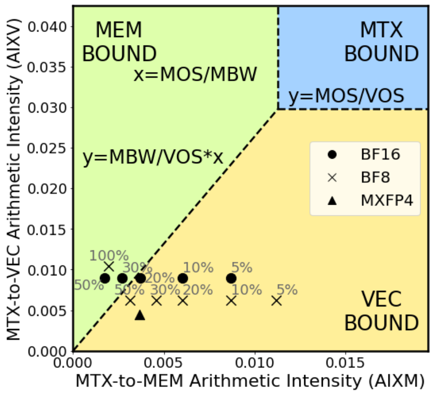
To gain further insight, Table 2 shows, for kernels with different compression schemes, the performance limits predicted by the roofline model (R-L) and the Roof-Surface model (R-S), and the real measured performance. We see that, for almost all kernels, the Roof-Surface produces relatively accurate performance bounds, while the roofline is often way off. For kernels BF8, BF16_50%, and BF16_30%, the performance bounds predicted by R-L and R-S are the same. The reason is that these kernels are classified as MEM-bound by both models.

4.2 The 2D Bounding Region Diagram

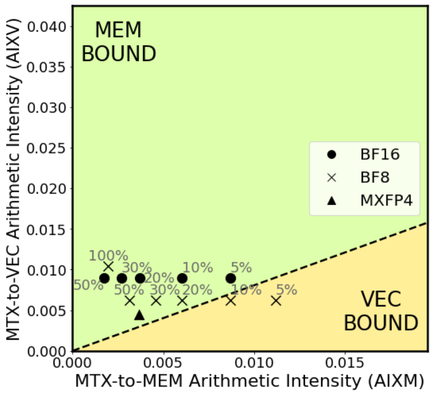
We now introduce a 2D representation of the Roof-Surface plot that is easier to visualize. We call it the Bounding Region Diagram (BORD). BORD is the projection of the roof surface on the xy plane. A BORD does not depict FLOPS information, but identifies which one of the plotted factors bounds the performance of a given kernel.

Figures 5a, 5b, and 5c show the BORD for some variants of the SPR architecture. In the general case, a BORD divides the plane in three regions, bound by memory, vector, or memory operations. As shown in Figure 5a, the equations of the lines separating the three regions are: $y = (MBW/VOS) * x$, $x = MOS/MBW$, and $y = MOS/VOS$.

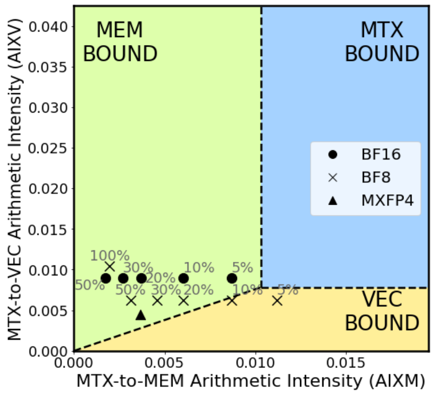
The BORD in Figure 5a corresponds to SPR with HBM. The figure shows the positions of different compressed GeMM kernels that use BF16 or BF8 with different density levels, or MXFP4. Some of the BF8 and MXFP4 points were shown in Figure 3b. We observe that the vast majority of kernels are VEC-bound. To reach the performance



(a) BORD for HBM SPR



(b) BORD for DDR SPR



(c) BORD for HBM SPR with 4x VOS

Figure 5: Bounding Region Diagrams (BORD). The batch size is N=16.

of the roofline in Figure 3b, these points must be pushed away from the VEC-bound region.

Figure 5b shows the BORD for SPR with DDR, which has a smaller *MBW* value. Now, the area of the MEM-bound region increases. The MTX-bound region is no longer visible for the AI_{XM} and AI_{XV} value ranges plotted in the BORD. The BORD also shows that all of our kernels except BF8_10% and BF8_5% are in the MEM-bound area. This explains why most of the kernels in Figure 3a reach the roofline.

Finally, Figure 5c shows the BORD when we take the HBM SPR variant and increase the vector throughput (VOS) by 4x, in an attempt to eliminate the vector bottleneck. When compared to Figure 5a, we see that the area of the VEC-bound region decreases and the MEM-bound region covers more kernels. However, even a 4x VOS increase is not enough to make all kernels not VEC-bound.

We find that, in the HBM SPR variant of Figure 5a, over 95% of the dynamic instructions of the cores typically perform tile decompression, and that cores are already using 40–80% of their commit slots. Hence, increasing the VOS by 4x would require not only a 4x increase in the number of SIMD AVX units, but also a prohibitive increase in the core's superscalar width [58]. Alternatively, increasing the SIMD AVX vector width by at least 4x is also undesirable, as it requires significant pipeline changes (e.g., redesigning wider versions of all the vector instructions, adding new register files, etc.). Further, feeding the core with vectors so large would require, at a minimum, increasing the number of ports in the L1 cache. This would in turn hurt the L1 access latency and the core's cycle time, affecting the core's performance for general-purpose workloads. We further evaluate the limitations of traditional CPU scaling in Section 8.

Appendix 12.2 shows the effect of the batch size on the Roof-Surface, while Section 9 discusses how the Roof-Surface generalizes to other use-cases or architectures.

5 DECA Overview and Out-of-Order Invocation

The previous analysis reveals that, to hide the decompression overheads with a conventional solution, one would need a very expensive scaling of the resources of a general-purpose core. This motivates us to propose *DECA*, a *near-core decompression accelerator for ML models*. DECA offloads vector processing for decompression from the cores. In this section, we describe DECA's integration and then introduce a new mechanism and ISA extensions for efficiently overlapping the operation of CPU cores and DECA accelerators.

5.1 DECA Placement & System Integration

We envision each CPU core to have an associated DECA accelerator as shown in Figure 6. At a high level, DECA has a processing element (PE), control registers, and tile output (*TOut*) registers. It has a memory-mapped interface that allows the core to write commands and read data. The core uses privileged stores to the control registers to configure the PE to perform decompression of tiles with a given quantization scheme and with or without sparsity. These stores also fill some look-up tables (LUTs) that DECA employs for efficient dequantization (Section 6).

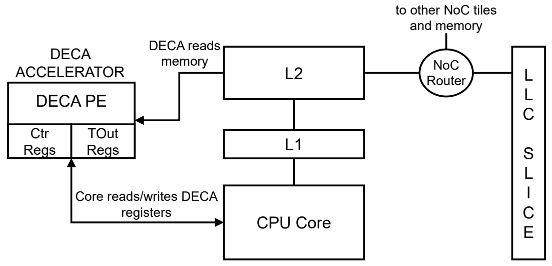


Figure 6: DECA placement next to a core.

The DECA PE reads a compressed tile from memory, processes it, and then writes the decompressed tile to the TOut registers. Then, the CPU core reads the TOut registers and uses the data to execute the GeMM using AMX instructions. The PE accesses memory through the L2, issuing both regular loads (but never stores) and prefetch requests, generated by a prefetcher integrated in the PE. DECA shares the L2 TLB with the core like prior work [19, 24, 66] and, therefore, uses the virtual space of the CPU core.

CPU writes to the L1 propagate to DECA with the mechanism used for a 2-core cluster with a shared L2 [2]. CPU writes to L1 change the cache line's state at the inclusive L2, so if DECA tries to read a line written by the CPU, the updated copy will be moved from the CPU's L1, to the L2, and eventually to DECA.

DECA can potentially be used by multiple processes. One way to handle this would be to save and restore the DECA state on context switches. Alternatively, we propose that DECA retains its state across context switches and only when a new process attempts to use DECA, a trap to the OS saves the state and reconfigures DECA. Further, in the case of SMT, the OS ensures that only one thread per core has DECA access, and traps otherwise. Linear algebra kernels are typically able to saturate a core with a single hyperthread, and thus Intel advices against using SMT in combination with the TMUL [36].

5.2 DECA-Core Cooperative Tile Processing

To execute compressed GeMMs with high performance, we introduce a mechanism that overlaps vector operations in DECA with AMX operations in a CPU core using hardware double buffering. The design is shown in Figure 7. DECA has two Loader modules and two TOut registers. A Loader reads a compressed tile from the memory system, which includes three data structures: the data, a bitmask, and scaling factors. A Loader can also issue prefetches to load a tile in advance.

The journey of a tile involves DECA loading it into a Loader (D1 in Figure 7), decompressing it in the DECA vector pipeline (D2), and storing it in a TOut register (D3). Then, the core reads it (C1), uses it to perform the AMX operation (C2), and prompts a Loader (C3) to initiate the fetching of the next tile by passing the starting address and the length of the three data structures of the tile. As shown in the figure, the double buffers enable overlapping of the operations on two tiles. While the core is reading and processing Tile *i-1*, DECA reads, processes, and writes out Tile *i*. After the core finishes *i-1*, it triggers the fetching of Tile *i+1*.

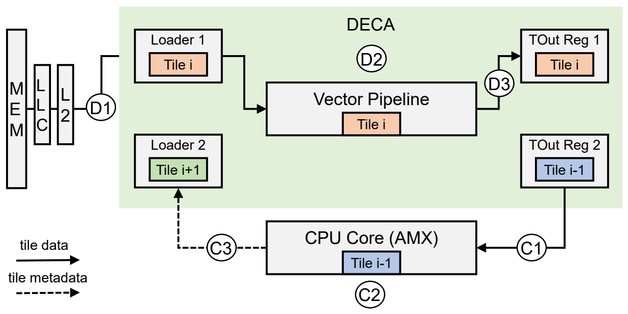


Figure 7: DECA-CPU core cooperative tile processing.

We explore two options for a CPU core to communicate with DECA. The first one uses regular stores to the memory-mapped DECA interface; the second uses ISA extensions that we describe in Section 5.3. Using the first approach, Figure 8 shows the pseudocode of the core as it processes tiles as shown in Figure 7. The key instructions are those in Lines 4-6. The core uses *TLoad* (an AMX instruction) to load tile T_{i-1} from a DECA TOut register into a tile register TReg$_1$ (Line 4). It then uses this tile in a *TComp* instruction (an AMX instruction that performs a GeMM), saving the output in a tile register TReg$_2$ (Line 5). Finally, it writes the metadata for tile T_{i+1} (shown as M_{i+1}) to a memory-mapped register in DECA's Loader2 using a plain store. The write prompts Loader2 to initiate the fetch of tile T_{i+1}. In parallel with Lines 4-6, DECA is decompressing T_i.

```
1   ............
2   DECA_{ldr1}  ←  ST $M_i$
3   Fence
4   TReg_1  ←  TLoad $T_{i-1}$
5   TReg_2  ←  TComp TReg_1
6   DECA_{ldr2}  ←  ST $M_{i+1}$
7   Fence
8   TReg_1  ←  TLoad $T_i$
9   TReg_2  ←  TComp TReg_1
10  ............
```

Figure 8: CPU core pseudocode for store-based DECA invocation.

```
1   ............
2   TReg_1  ←  TEPL $M_{i-1}$
3   TReg_2  ←  TComp TReg_1
4   TReg_1  ←  TEPL $M_i$
5   TReg_2  ←  TComp TReg_1
6   TReg_1  ←  TEPL $M_{i+1}$
7   TReg_2  ←  TComp TReg_1
8   ............
```

Figure 9: CPU pseudocode for TEPL-based DECA invocation. The architectural tile registers TReg$_1$ and TReg$_2$ get renamed to different physical tile registers in each iteration.

Figure 8 also shows a piece of the previous iteration (Line 2) and of the subsequent iteration (Lines 8-9). To prevent incorrect memory operation reordering, we add a memory fence per iteration. Specifically, the load of tile T_i (Line 8) should not execute before the metadata for T_i is written to the control register in DECA's Loader1 (Line 2), which resets TOut Register 1 and initiates the tile fetch from memory. Since these two instructions do not depend on each other, we place a fence in Line 3. There is a fence in each iteration.

Unfortunately, this approach is likely to deliver limited performance for two reasons. First, each iteration has a fence that prevents cross-iteration overlap. Second, within an iteration, no instruction overlaps: the instructions in Lines 4 and 5 have a true dependence, and the store in Line 6 can only perform the update when it is at the head of the reorder buffer (ROB). The execution of all instructions is serialized, as if the core was in-order. As a result, in every iteration, the latency of the communication between core and DECA (both the load and the store) is fully exposed.

5.3 ISA Support for Out-of-Order Invocation

To reinstate out-of-order execution and hide the core–DECA communication, we propose a different approach that relies on an extension to the CPU AMX ISA. We call the extension *Tile External Preprocess and Load* (TEPL). The main idea is to eliminate the per-iteration fence in Figure 8 by combining, in hardware, the instructions in Lines 2 and 8 into a single instruction. This instruction updates the control register of a loader with metadata, triggering a tile fetch, and only returns to the core when DECA has decompressed the tile and stored it in a core tile register (e.g., TReg$_1$).

A TEPL instruction takes as arguments a core source register with the metadata for a tile, and a core destination tile register. During the execution of the instruction, the metadata in the core source register is transferred to DECA to initiate decompression and the instruction completes when the decompressed tile is received by the core destination tile register. In this design, the maximum number of TEPL instructions that can execute concurrently is equal to the total number of DECA Loaders (i.e., two). A structural hazard prevents more TEPLs from executing, since each DECA loader is able to handle only one tile at a time.

With this design, the code in Figure 8 is rewritten as Figure 9. Fences are removed and an iteration has only two instructions (Lines 4, 5). There are no register dependencies between the iterations because TReg$_1$ and TReg$_2$ are renamed. However, a structural hazard causes the TEPL in Line 6 to stall until one of the two previous TEPLs completes.

A context switch can only occur in between two instructions. Hence, the DECA state that needs to be saved and restored when a new process attempts to use DECA is only the DECA control registers and LUTs, and not any tile data.

To support these instructions, the core has a *TEPL Queue*, and two *TEPL execution ports*, each leading to a DECA loader. A TEPL instruction is deposited into both the ROB and the TEPL Queue. When the TEPL source register is available and there is a free TEPL execution port, the TEPL is issued to DECA.

To attain high performance, TEPLs are issued to DECA as soon as possible—they do not wait until they reach the ROB head. Hence, like a load instruction, they execute speculatively and out-of-order. Invoking DECA speculatively is always safe, as DECA does not update memory state. If the core needs to flush the pipeline (e.g., on a branch misprediction or exception) while a TEPL instruction is outstanding, the core sends a squash signal to DECA. At that point, DECA aborts any tile operation in progress, no matter the state it is in. The core may safely reissue the same TEPL. For a TEPL to commit, it needs to both (1) finish its execution on DECA and write the output to the CPU core's physical tile register and (2) be at the head of the CPU core's ROB.

If DECA reads a cache line and then a core updates the line, DECA does not receive an invalidation from the L2. This is not a concern because DECA uses read-only weights, which are not updated by CPU cores during LLM inference. In the uncommon case that CPU cores update data that can potentially be accessed by a TEPL, we place a fence before the TEPL, to ensure that the TEPL reads the updated data.

Overall, this design hides the communication between the core and DECA. The core executes without fences and overlaps the operation on multiple tiles.

6 DECA Microarchitecture

We now describe the microarchitecture that enables DECA to sustain high decompression performance and, at the same time, support a rich set of compression schemes. For simplicity, in the rest of the paper, we assume that DECA's output tile is in BF16 format. DECA can be trivially configured to produce INT8 output tiles.

6.1 DECA Microarchitecture

Figure 10 displays the DECA PE microarchitecture. To understand it, we describe its multiple components.

1. Accessing Memory. DECA has two Loaders, each composed of a *Load Unit* (LDU) and a prefetcher. The LDU accesses memory to read a tile's compressed weights, bitmask, and scaling factors. The base memory addresses and lengths of these structures are part of the metadata provided by the CPU on DECA invocation. When a requested cache line arrives from memory, depending on which of the three types of data it contains, it is placed in the *Sparse Quantized Queue* (SQQ), *Bitmask Queue*, or *Scale Factor Queue*.

2. Prefetcher. The prefetcher (PF) observes the address bases and lengths used for a tile, and predicts the ones for future tiles. The PF then generates prefetch requests that will bring this data to the L2 cache. Since DECA operates on virtual addresses, it uses the L2 TLB for address translation for prefetches, similar to normal

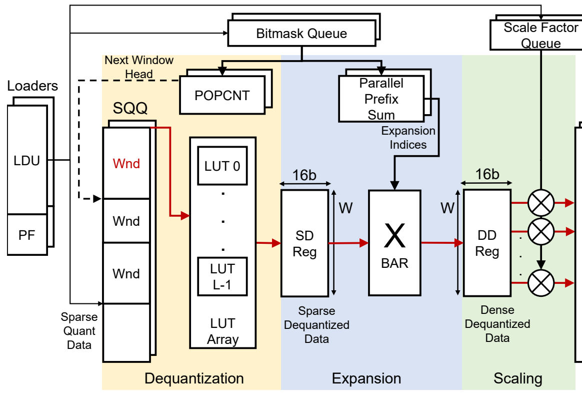


Figure 10: DECA PE microarchitecture.

DECA loads. The DECA PE does not have any prefetch buffers, since the prefetched data stays at the L2 and only moves to DECA through demand loads. Instead, the DECA PE reuses the L2 prefetch buffers/MSHRs. The existing L2 prefetcher only issues prefetches on behalf of the CPU core. The PF aggressiveness (i.e., how many tiles ahead are prefetched) is configurable.

3. Pipeline Stages. The pipeline is split into three stages, responsible for dequantization, expansion (i.e., de-sparsification) and scaling. To enable pipelining, each stage has its own output register, namely the SD, DD, and TOut registers in the figure. The Dequantization stage reads values from the SQQ, dequantizes them using an array of L Lookup Tables (*LUT Array*), and writes dequantized BF16 values to the *Sparse Dequantized* (SD) register. These values are potentially sparse—stored contiguously with zero values skipped.

The Expansion stage de-sparsifies data by inserting zeros in the positions indicated by the bitmask. This operation is done using a crossbar (*XBAR*) that is controlled using expansion indices. The latter are generated from the bitmask using the *Parallel Prefix Sum* circuitry. The result is written to the *Dense Dequantized* (DD) register, which has dense (i.e., with explicit zeros) dequantized data.

Finally, if group quantization is used, the Scaling stage applies appropriate scaling to the BF16 values by multiplying them with the scaling factors. It then writes the final values to the *TOut* register. The critical path is shown with red arrows in the figure.

4. Duplicated Modules. A DECA PE contains two Loaders and two TOut registers to enable the overlapping of DECA and CPU operation. Hence, as shown in Figure 10, the PE replicates LDU, PF, the input queues (SQQ, Bitmask queue, and Scale Factor queue), and TOut. One Loader can be loading data while the pipeline is processing data that was provided by the other Loader. The POPCNT and Parallel Prefix Sum circuitry process the bitmask by mainly performing additions of 1-bit data; they are also duplicated so their latency can be hidden. The rest of the pipeline is not duplicated and is used by one Loader-TOut pair at a time.

5. Vector Operations (vOps). It takes multiple cycles to generate a decompressed tile, which always contains 512 BF16 elements. This is because the pipeline generates output *chunks* of W elements (e.g., 32 in our design) at a time, each using one DECA Vector Operation (vOp). In the absence of pipeline bubbles, a new chunk is generated every cycle. A vOp reads data from the SQQ, executes in the pipeline stages, and finally writes the chunk of W elements to a TOut. vOps exploit pipelining: when a vOp enters the Expansion stage, the next vOp can enter the Dequantization stage. The vOps are processed in order, and can enter the pipeline as long as (1) their input has arrived from memory and (2) the first pipeline stage is free.

Without sparsity, a vOp reads W elements from the SQQ. With sparsity, less than W elements are needed, since the SQQ does not contain zero values. We refer to the elements that a given vOp needs to read from the SQQ as the vOp's window (*Wnd*). To determine the size of a Wnd, the POPCNT circuitry counts the number of "1s" in the bitmask, and determines the end of the current Wnd and the start of the next Wnd. The latter is the next SQQ position from which data will be read into the pipeline.

6. LUT Array Organization. The DECA dequantization stage supports quantized numbers of 8 bits or less, which can represent a maximum of 256 different values. For this reason, each of the L LUTs in the LUT array stores 256 BF16 values. Dequantizing an 8-bit value corresponds to a lookup using the 8-bit value as the LUT address to read a BF16 dequantized value. DECA contains L LUTs to allow for parallel dequantization of multiple values.

In addition, each LUT is internally divided into 4 smaller sub-LUTs, each one with a read port and 64 entries. If the quantized data bitwidth is 6 bits or less, the 4 "small" LUTs can be used in parallel to enable 4 reads from one 256-entry "big" LUT. For less than 6-bit quantization, some of the LUT entries are redundant and are not used at runtime.

7. Bubbles and the Roof-Surface. We set the number of "big" LUTs to $L < W$ to limit DECA's area. If the Wnd of a vOp is larger than L elements, the vOp occupies the Dequantization stage for more than one cycle. This injects one or more *bubbles* in the pipeline, which reduce the vOp throughput. For example, the Wnd of a dense 8-bit quantization scheme is W and, therefore, a vOp will always require W/L cycles for dequantization. Although setting $L < W$ limits the DECA throughput for dense quantization schemes, this is not a major concern. The reason is that dense schemes like BF8_100% and MXFP4 require less vector throughput (i.e., VOS) in order to escape the vector (VEC) region. This can seen in the BORDs of Figure 5.

On the other hand, sparser schemes require a higher VOS to escape the VEC-bound region. Luckily, this is naturally achieved by the DECA pipeline: the probability that the Wnd of a vOp is larger than L decreases with sparsity. Thus, fewer bubbles are introduced for sparse schemes, naturally achieving higher throughput than their dense counterparts for the same L. The same behavior is achieved for lower bitwidth schemes because they can perform more than L reads in parallel from the LUT array.

9. Generality and Performance. DECA supports quantization formats of 8 bits and lower, group quantization, and unstructured sparsity, which cover most current and likely future model compression schemes. DECA's design is flexible, since by changing the values in its LUT array and/or using different scale factors, it enables the support for a rich set of formats without redesigning the hardware. Additionally, individual stages can be skipped if they are unneeded (e.g., quantization without sparsity). In terms of performance, the main benefit of DECA is that it replaces multiple vector (AVX) instructions by a single vOp that performs the whole decompression—dequantization, expansion, and scaling—for W elements. The decreased vOp count *increases the* AI_{XV} (Section 4), moving the points away from the VEC region.

6.2 Quantitative Microarchitecture Design

We now show how DECA's W and L relate to the Roof-Surface model quantitatively. Consider Equation 2. We should express how the parameters in the equation depend on W and L. Of all the parameters, only AI_{XV} depends on W and L. Indeed, VOS is fixed to $c * 1 * f$, since each of the c CPU cores has one DECA PE that completes at most one vOp per cycle and operates at the core frequency f. On the other hand, the AI_{XV} of different kernels depends on DECA's W and L parameters. To calculate AI_{XV}, we need to add-up the number of vOps that are needed per tile plus the number of bubbles that are generated per tile.[1]

The number of vOps per tile is $\#vOps = 512/W$, since each tile has 512 elements and a single vOp produces W elements. We express the number of bubbles per tile as $\#bbl = \#vOps * bpv$, where bpv is the number of bubbles per vOp. Since bubbles can only be generated due to insufficient resources in the Dequantization stage, we use L_q to denote the maximum number of elements that DECA can dequantize in a cycle. L_q is equal to L for 8-bit quantization schemes, $2 * L$ for 7-bit, and $4 * L$ for 6-bit and below. Without sparsity, $bpv = ceil(W/L_q) - 1$. With sparsity, the bubble generation is not deterministic, as it depends on the number of nonzeros in a compressed tile. For a matrix of density d, if we assume that nonzeros are uniformly distributed, then the number of nonzeros in W consecutive matrix elements is a binomial distribution with parameters W, d. We compute the expected number of bubbles as:

$$\begin{aligned} bpv &= \sum_{k=0}^{\frac{W}{L_q}-1} k \cdot P(kL_q < NNZ \leq (k+1)L_q) \\ &= \sum_{k=0}^{\frac{W}{L_q}-1} k \cdot [F((k+1)L_q; W, d) - F(kL_q; W, d)] \end{aligned}$$

where $F(i; W, d)$ is the binomial cumulative distribution function. Finally, the AI_{XV} is given by $1/[\#vOps * (1 + bpv)]$.

Intuitively, given a DECA architecture with a given W and L, a certain kernel introduces an average number of bubbles per decompressed tile equal to $\#vOps * bpv$. As one tunes the architecture by modifying W or L, the number of bubbles may increase or decrease, pushing the kernel in the Roof-Surface plot down or up, respectively, and moving it closer to or farther away from the VEC-bound region, respectively. Consequently, we can use the Roof-Surface model to perform an analytical Design Space Exploration (DSE). We can plot the BORDs of different (W, L) pairs and pick the design that pushes all the kernels out of the VEC-bound area at the minimum DECA hardware cost. In Section 8.2, we perform such a DSE.

7 Methodology

1. Simulation and System Parameters. To evaluate our work, we simulate a 56-core server with SPR-like parameters using an in-house simulator based on Sniper [9]. We evaluate a DDR5-based and an HBM-based memory with about 260GB/s and 850GB/s achievable memory bandwidth, respectively. We extend the simulator so each core has: (1) a DECA PE, and (2) an 8-entry TEPL Queue and ports to support TEPLs in the core pipeline. Both cores and DECA PEs run at 2.5GHz. Our baseline DECA PE is dimensioned with W=32 and L=8, but we also evaluate other options in Section 8.2.

[1]Note that to calculate the AI_{XV} for decompression with a CPU core instead of with DECA, we also have to take into account CPU core bubbles (e.g., due to dependencies between AVX operations and operation latencies).

2. Software and DECA Control Code Generation. As our software baseline, we use the Intel libxsmm kernels that overlap weight decompression and GeMM execution (Section 2.4). To invoke DECA, we modify the libxsmm JIT compiler by replacing the AVX decompression sequence with TEPL instructions.

To evaluate the effectiveness of DECA for compressed GeMMs in isolation, we implement a large cascade of Fully Connected (FC) layers (without other types of layers) and use Parlooper [17] for loop parallelization. The weight matrices in these layers have ≈250M parameters, similar to the large FC layers of Llama-2-70B. Libxsmm and Parlooper are integrated in the Intel Tensor Processing Primitives (TPP) Framework [16], which supports end-to-end Llama-2 and OPT inference on CPUs. Hence, we use TPP as is for software-only LLM inference, and by invoking the TEPL-augmented libxsmm kernels for inference with DECA. We test batch sizes (N) of 1–64.

3. Compression Schemes. We evaluate the BF16, BF8, and MXFP4 compression schemes, which we refer to as Q16, Q8, and Q4, respectively. We limit the compression schemes to these because these are the ones supported by libxsmm. We also evaluate unstructured sparsity with weight density ranging from 50% to 5% for Q16 (only sparsity) and Q8 (quantization plus sparsity). The Q4 sparse kernels are not currently supported by our libxsmm-Parlooper-TPP stack, so we cannot directly evaluate them. For end-to-end Llama-2-70B and OPT-66B inference, the uncompressed dense Q16 baseline, Q16 with 50% density (Q16_50%), and dense Q8 do not fit in the 64GB of HBM. Hence, we simulate a larger HBM capacity for those schemes.

Table 3: Key DECA structure types, sizes, and port counts.

Structure	Type	Total Size	Entry Size	# Rd Ports	# Wr Ports	# per PE
SQQ	SRAM	2KB	64B	1	1	2
TOut Reg	SRAM	2KB	64B	1	1	2
Bitmask Queue	Regs	64B	4B	1	1	2
Scale Factor Queue	Regs	64B	2B	1	1	2
LDU	CAM	256B	8B	1 (+1 search)	1	2
Big LUT	SRAM	512B	2B	4	1	L=8
SD/DD Reg	Regs	64B	64B	1	1	1

4. Area and Power. We estimate the area and power of our DECA design with W=32 and L=8. Table 3 shows the key DECA structures. The key for the LDU CAM enters through the search port. The design also includes 64 32-bit parallel prefix adders for Parallel Prefix Sum / POPCNT, 32 BF16 multipliers for scaling, and a triangular crossbar with 32 I/O ports, where each port is 2B wide.

To estimate the area of the SRAM/CAM structures (e.g., LDU and SQQ) and registers, we use CACTI [3]. For the crossbar, BF16 multipliers, and adders, we use numbers from [8], [83], and [64], respectively. We then use [69] to scale down the numbers to 7nm. Overall, we estimate the total area for 56 DECA PEs to be ≈2.51 mm^2. Of this area, the LDUs, SQQs, Bitmask queues, Scale Factor queues, and TOut registers consume about 55%; the LUT array consumes 22%; and the rest consumes 23%. Given that the total die area of a 56-core SPR is around 1600 mm^2 [76], the DECA area overhead is less than 0.2%.

To estimate the power of the DECA PEs, we use CACTI and information from the literature. We estimate the combined dynamic power (at maximum activity) and static power to be 3.1W, or about 1% of the SPR TDP of 350W. Queues and registers contribute with about 50% of DECA's power, the LUTs with 21%, while the rest with 29%. When idle, the DECA PEs consume less than 0.1% of the TDP. We use CACTI to validate that all the memory structures meet timing, and [8, 64, 83] to validate that the rest of the components also do.

5. Systems Evaluated. In our evaluation, we compare several SPR systems with TMUL hardware that execute GeMMs and LLM workloads. *Baseline* is a system that loads uncompressed BF16 weights from memory. *Software-only* is a state-of-the-art system that loads compressed weights from memory and decompresses them using AVX instructions; it uses the libxsmm library. *DECA* is our system, which loads compressed weights from memory and decompresses them using DECA. Sometimes, we also use *Optimal*, an ideal system that delivers the performance predicted by the roofline model and, therefore, assumes that all VEC overheads are hidden.

8 Evaluation

8.1 DECA for Compressed GeMMs

Figures 11 and 12 show, for different compression schemes, the speedups of the Software-only, DECA, and Optimal systems over the Baseline, running the GeMMs in our FC layer cascade for N=1. The compression schemes appear in increasing compression factor.

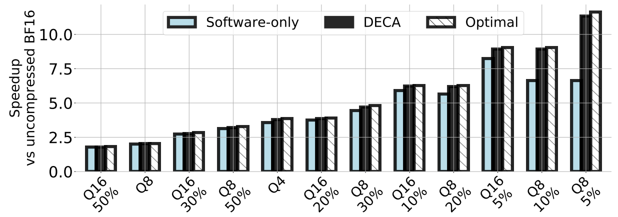


Figure 11: Compressed GeMM speedups for DDR and N=1.

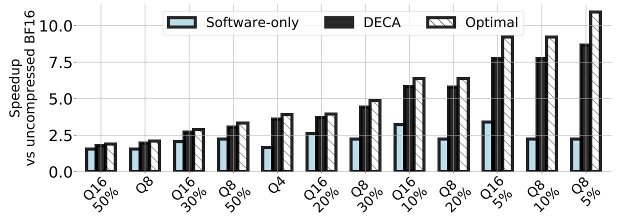


Figure 12: Compressed GeMM speedups for HBM and N=1.

For the DDR setting (Figure 11), DECA offers speedups over Software-only only for high compression factors. This is expected since, according to the BORD in Figure 5b, only high compression factors are VEC-bound. The speedup of DECA over Software-only reaches 1.7 for Q8_5%. For the HBM setting (Figure 12), DECA offers speedups over Software-only for almost all the compression schemes. This is because, as shown by the BORD in Figure 5a, almost all schemes are VEC-bound. The speedups reach 4.0× for Q8_5%. In addition, in both DDR and HBM, the performance of DECA is close to the Optimal one, revealing that DECA successfully hides the VEC overheads. We repeated this analysis for batch sizes of up to N=16 and observed similar results.

DECA-augmented cores are much more capable at vector processing than conventional cores. Figure 13 compares the performance of the DECA and Software-only systems for different numbers of cores. The figure shows data averaged across all the compression schemes for the DDR setting with N=4. We see that 16 DECA-augmented cores achieve higher performance than 56 conventional cores. With DECA, the extra cores can run other workloads that consume little memory bandwidth, or be power-gated.

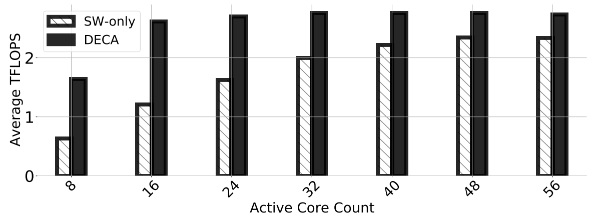


Figure 13: TFLOPS across all compressions for DDR and N=4.

To provide further insights into the performance of the Software-only and DECA systems, Table 4 displays the percent utilization of the memory bandwidth, of the TMUL, and of either the CPU's AVX units or DECA for different density factors of the weight data. Since performance is proportional to the utilization of the TMUL, the table shows that the DECA system has much higher performance than Software-only.

Table 4: Component utilization for Q8, N=1, and HBM.

Density Factor	Software-only			DECA		
	MEM	TMUL	AVX	MEM	TMUL	DECA
100%	**74%**	14%	50%	**93%**	18%	75%
50%	66%	20%	**88%**	**92%**	28%	71%
20%	35%	20%	**89%**	**91%**	53%	63%
5%	19%	20%	**89%**	73%	79%	**87%**

Since the operations of the three components overlap, the one with the highest utilization ends up being the bottleneck. In the Software-only system, for almost all of the densities, the bottleneck is the AVX vector units. This observation validates the Roof-Surface prediction. With DECA, the memory bandwidth is much better utilized, leading to direct performance improvements. As sparsity increases, the kernels take less time to execute, as shown by the higher TMUL utilization. Hence, one would expect a large increase in DECA utilization. In reality, this is not seen. The reason is that, as explained in Section 6, as sparsity increases, DECA naturally suffers fewer pipeline bubbles and, therefore, increases its throughput.

Finally, we compare DECA with the alternative of scaling a CPU core's vector resources as a method to alleviate the decompression overhead. We model cores with: (1) 4× more vector AVX units (*More AVX Units*) or (2) 4× wider AVX units (*Wider AVX Units*). These AVX2048 units are optimistically modeled by removing the dynamic instructions from 3 out of 4 iterations of the decompression loop. Since we do not modify the system cache line, each AVX2048 memory operation is executed as 4 cache-line sized operations. For these non-DECA systems, we do not scale the superscalar width of the core or the number of L1 ports since, as explained in Section 4, such changes are prohibitive. Figure 14 compares the speedups of DECA, *More AVX Units*, and *Wider AVX Units* over Baseline for different compression schemes. We see that the performance of conventional vector scaling methods is far below DECA's.

8.2 Design Space Exploration with Roof-Surface

The DECA W and L parameters determine how fast DECA can decompress, but values that are too large may increase area without real benefit. Here, we use the *Roof-Surface* to examine the performance for different {W,L} pairs. To dimension DECA, we want to pick the smallest {W,L} pair for which the predicted performance

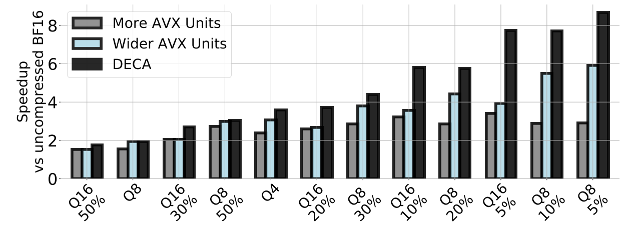
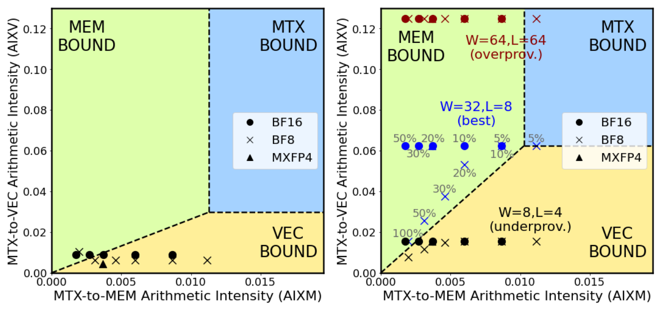


Figure 14: DECA vs traditional vector scaling for HBM & N=1.

saturates (i.e., all the kernels are predicted *not* to be VEC-bound anymore). Figure 15 shows the BORDs for the HBM SPR system without DECA (a) and with DECA (b) with three different {W,L} sizes: {W=8,L=4} (which we find it is underprovisioned), {W=32,L=8} (which we find it is the best one), and {W=64,L=64} (which we find it is overprovisioned).



(a) No DECA **(b) DECA with different {W,L} pairs.**

Figure 15: HBM BORDs with different-sized DECAs.

We observe that, compared to a CPU without DECA, DECA has fewer vector operations per second (VOS), since its VEC-bound region is larger. However, DECA decreases the number of vector operations needed per matrix operation (i.e., it increases AI_{XV}) as discussed in Section 6. The underprovisioned DECA with {W=8,L=4} is unable to push the kernels out of the VEC-bound region. The overprovisioned one with {W=64,L=64} pushes them out, but more than needed. The best design ({W=32,L=8}) pushes them just enough so that they are out of the VEC-bound region.

We simulate the performance of these designs to validate the model's accuracy. We find that the DECA-best system is 2× faster than the DECA-underprovisioned one. The DECA-overprovisioned system is less than 3% faster than the DECA-best one. At the same time, DECA-best is much cheaper than DECA-overprovisioned: it has 8× fewer LUTs and half the W. Overall, the Roof-Surface model accurately captures the dynamics of the matrix-vector-memory interaction and can guide microarchitectural decisions.

8.3 Analysis of DECA Integration and TEPLs

We now evaluate different decisions we made regarding DECA's integration with a core. We start with a base configuration where DECA reads compressed tiles from the LLC (bypassing L2), writes decompressed tiles into the L2 for the core to read, and is invoked using loads, stores, and fences. Then, we progressively enhance it to: (1) allow the accelerator to read compressed tiles from the L2 and use the L2 prefetcher (*+Reads L2*), (2) use its own prefetcher instead of the L2 prefetcher (*+DECA prefetcher*), (3) write to the TOut Regs instead of to the L2 (*+TOut Regs*), and (4) use TEPL instructions as in Figure 9 instead of loads, stores, and fences (*+TEPL (DECA)*).

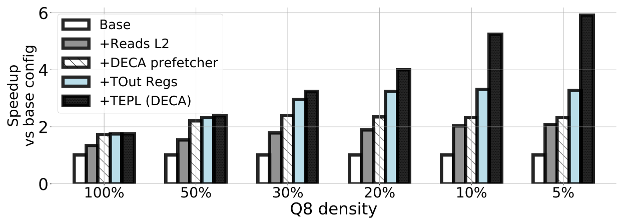


Figure 16: Speedups due to different DECA integration features for HBM and N=4.

Figure 16 shows the speedup of each configuration over the base configuration for Q8 with different densities. We see that *+Reads L2* improves performance for all densities. The benefit comes from the L2 hardware prefetcher already available in the system, which fetches future tiles, hiding the memory/LLC access latencies. *+DECA prefetcher* further improves performance by using the DECA prefetcher rather than the default L2 one. *+TOut Regs* and *+TEPL (DECA)* reduce or hide the DECA-core communication latency, and are needed to execute instructions speculatively and out-of-order. Specifically, *+TOut Regs* enables the core to directly fetch data from DECA, instead of taking the longer path through the L2. Further, *+TEPL (DECA)* overlaps communication with computation, hiding the former.

The effectiveness of *+TOut Regs* and *+TEPL (DECA)* increases as the density decreases. This is because DECA takes less time to process a lower density tile, while the overhead of communication with the core remains constant. Thus, for lower densities, the communication cost gets more exposed. TEPLs are very effective for low-density models: for 5% density, they double the performance.

Table 5: Prefetching sensitivity analysis (MXFP4,N=4,HBM).

Prefetching Scheme	L2 Miss (Million)	Weight PFs (M)	Other PFs (M)	Perf. (TFLOPS)
No Weight Prefetching	90	0	6	2.9
Default L2 Prefetcher	12	78	6	3.5
DECA Pref. Degree 1	12	79	6	3.4
DECA Pref. Degree 5	1	89	6	6.0
DECA Pref. Degree 15	2	90	5	5.4

In Table 5, we perform a sensitivity study of the DECA prefetch degree (i.e., how many tiles ahead to prefetch) for a GeMM compressed with MXFP4. We also compare to no weight prefetching, and to prefetching weights using the default L2 prefetcher. In all configurations, the L2 prefetcher also prefetches non-weight data (e.g., activations) triggered by CPU loads (*Other PFs*). When the DECA prefetcher is active, the L2 one does not prefetch weights.

We see that the DECA prefetcher with degree of 5 (which is the one our evaluation uses) offers great gains over both the default L2 prefetcher, and the DECA prefetcher with degree 1. On the other hand, too deep prefetches (e.g., degree 15) start to hurt performance. Also, we found that one of the key reasons for the performance gains enabled by the DECA prefetcher is that it generates prefetches with virtual addresses, which work seamlessly across physical page boundaries—in contrast to the physically-addressed prefetches issued by the default L2 prefetcher that don't cross pages. Finally, the number of weight prefetches does not increase drastically with the degree. The reason is that the DECA prefetcher does not issue duplicate prefetches; it filters out tiles for which prefetches have already been issued.

8.4 DECA for LLM Inference

Lastly, we show the performance benefit of DECA for LLM next token generation (i.e., including the non-GeMM stages). Table 6 shows the next token latencies of the Llama2-70B and OPT-66B models on SPR with HBM, for 128 input tokens, 128 output tokens, batch sizes of 1, 16, and 64, and different compression schemes. We compare Software-only (*SW-only*) with DECA. In the table, the Q16 entries do not have a DECA entry because DECA is not used with uncompressed data. Also, as explained in Section 7, we simulate the Q16 model in *SW-only* assuming a larger HBM size that can fit it. Indeed, an additional benefit of DECA is that it enables efficient execution of large models that only fit in the HBM compressed.

Table 6: Llama2-70B & OPT-66B next-token latency (ms).

System	N=1			N=16			N=64		
	Q16	**Q4**	**Q8 30%**	**Q16**	**Q4**	**Q8 30%**	**Q16**	**Q4**	**Q8 30%**
				Llama2-70B					
SW-only	192.3	124.6	98.2	211.2	139.1	116.6	452.5	441.6	423.4
DECA	-	68.3	59.6	-	82.7	75.7	-	277.1	259.8
				OPT-66B					
SW-only	178.5	117.0	91.3	203.9	132.3	111.7	470.0	436.4	407.3
DECA	-	60.8	53.9	-	81.8	75.5	-	236.5	230.6

We see that DECA substantially reduces the next-token latency compared to *SW-only* for both models, and for all schemes and batch sizes. Using N=64 increases the next-token latency to high values, and is generally less meaningful for inference with CPUs (Appendix 12.1). Since N=64 is more compute-bound, compressed models for this batch size are less effective in comparison with N=1 and N=16. In fact, without DECA, MXFP4 is almost ineffective. Overall, across models, batch sizes, and compression schemes, DECA reduces the next-token time by 1.6×–1.9× over *SW-only*. If we only account for the more meaningful batch sizes (i.e., N=1 and 16), this translates into a 2.5×–3.3× reduction over the uncompressed base model (or a 1.6×–3.3× reduction if we include N=64). We observed similar results for shorter and longer token sequences.

Table 7: Comparing CPU, CPU plus DECA, and GPU platforms for W4A16 compression, Llama2-70B, and N=1.

Metric	SPR HBM	SPR HBM + DECA	RTX 6000 ADA	A100 80GB PCIe	H100 80GB PCIe
BW (GB/s)	850	850	960	1935	2000
Max compute (TFLOPS)	72	72	364	312	989
Next-token latency (ms)	125	68	58	40	38

We consider how close the performance of a CPU with DECA gets in comparison with GPUs. Table 7 considers three GPUs (RTX 6000 ADA, A100 80GB, and H100 80GB), and lists their bandwidth, maximum compute capability, and next-token latency from the repository in [11]. The results are for 4-bit quantized weights and 16-bit activations (W4A16), Llama2-70B, and N=1. We compare these numbers to results from simulations of our CPU system with and without DECA. We see that, with DECA, the SPR HBM system attains a next-token latency that is very close to RTX6000 ADA's. In addition, it has only a 1.8x higher next-token latency than an H100 GPU, despite the latter having 2.4x higher bandwidth and 14x higher maximum computational throughput.

Table 8: Comparison of DECA with related in/near-core accelerators.

Accelerator	Supports Different Quantizations	Supports Structured Sparsity	Supports Unstructured Sparsity	GeMM Execution Units	High GeMM Throughput	(I)n or (N)ear Core	Fine-grained Interleaving with the Core	HW Changes Required to the Core
TMUL [37]	Limited	✗	✗	Matrix	✓	I	✓	N/A
TensorCore [56]	Limited	2:4	✗	Matrix	✓	I	✓	N/A
RASA [40]	✗	✗	✗	Matrix	✓	I	✓	Many
VEGETA [39]	✗	2:4,1:4	✗	Matrix	✓	I	✓	Many
SAVE [23]	✗	✓	✓	Vector	✗	I	✓	Many
SPADE [19]	✗	✓	✓	Vector	✗	N	✗	None
DECA	✓	✓	✓	Matrix	✓	N	✓	Few, reusable

9 Discussion

In this section, we briefly discuss two additional topics.

1. Utility of a DECA-inspired engine for other architectures. DECA is not limited to Intel CPU architectures. It can also be used in other x86 processors such as AMD's, as well as in processors with different ISAs such as those of ARM and RISC-V. In both the ARM and RISC-V memory models, fences are also needed to enforce st → ld ordering when using store-based DECA invocation. Fences are needed to ensure that the accelerator is first invoked before trying to read from it. Hence, TEPL-like instructions would also be useful, and we expect the design to remain largely the same.

Regarding GPUs, similar to the TMUL, GPU Tensor Cores support only limited quantization formats and do not support unstructured sparsity. Table 7 reveals that the performance of kernels with 4-bit quantized weights largely scales with the bandwidth of the GPU. This suggests that, for these kernels, GPUs are largely bottlenecked by memory and not by decompression—thanks to the high vector FLOPS of GPUs compared to CPUs.

Currently, there are multiple software systems to decompress weights for GPUs. For example, AWQ [48] includes a dequantization kernel, and Flash-LLM [79] de-sparsifies data with unstructured sparsity and feeds it to the Tensor Cores. Although effective, Flash-LLM ends-up substantially increasing the utilization of the L1 and shared memory of the SMs, preventing full Tensor Core and HBM utilization. A DECA-inspired decompression engine could reduce the L1 and shared memory pressure, and thus potentially prove useful for GPUs for kernels involving sparsity or combination of sparsity and quantization. NVIDIA recently introduced the TMA accelerator [52] for supplying data from memory to Tensor Cores. Augmenting TMA with DECA-inspired decompression capabilities is an interesting future direction.

2. Roof-Surface Generality. The Roof-Surface is applicable beyond Intel chips, CPUs, and decompression. It applies in any scenario involving chains of matrix, vector, and memory operations. For example, beyond Intel, it can be used to model the performance of a kernel involving Scalable Vector Extensions (SVE)[68] and Scalable Matrix Extensions (SME) [77] in an ARM architecture. It can be used for non-CPU architectures that involve separate matrix/vector units such as the AWS Trainium [7, 14], the TPU [55], GPUs with their Tensor Cores and SIMT Cores, and the Tandem processor [20].

In addition, it is straightforward to extend the Roof-Surface for the general case of n pipelined abstract cooperative compute domains and memory. A general n-dimensional analytical model of the interaction of the domains can be derived as follows. First, one of the domains is chosen as the *target* (T). The final performance equation will be a limit for the operations of that domain. In this paper, we chose the MTX as the T domain. Then, we should express the AI of the T domain with respect to (1) memory (AI_{TM}), and (2) every other domain D_i (AI_{TD_i}). AI_{TM} is $\frac{1}{Bytes_{Top}}$, where $Bytes_{Top}$ is how many bytes need to be loaded from memory to execute one operation of the T domain. The AI_{TD_i} is given by $\frac{\#Tops}{\#D_i ops}$, where $\#Tops$ and $\#D_i ops$ are the total number of operations in the target kernel from domains T and D_i, respectively. Then the performance model equation in predicted $Tops/second$ or $\hat{TOS}$ is given by:

$$\hat{TOS} = min\{MBW * AI_{TM}, D_1OS * AI_{TD_1}, ..., D_nOS * AI_{TD_n}\} \quad (3)$$

where D_iOS is the rate at which the architecture can execute operations from domain D_i. For more than two domains plus memory, visualization requires more than 3 dimensions, but the model can still be used to produce performance limits or identify limiting factors analytically. If the resulting model is 3-D, then the visualization techniques in this paper (Roof-Surface and BORDs) apply.

10 Related Work

1. Decoupled Accelerators. A variety of stand-alone decoupled accelerators that target sparsity in ML and scientific applications have been proposed [10, 18, 22, 27, 31, 51, 59, 67, 85]. Other decoupled accelerators rely on quantization [38, 65, 91]. Recently, accelerators for attention are also becoming popular [25, 26, 42, 50, 74]. Decoupled accelerators come with large area and power budgets [39], and suffer from data movement overheads [19].

2. In/near-core Accelerators. Due to the previous reasons, CPU-integrated accelerators that reside in or near the CPU cores have been proposed [19, 23, 24, 39, 40, 54]. DECA falls in this line of work. Table 8 shows a taxonomy of the in/near-core accelerators that are most related to DECA. The table classifies the accelerators based on whether they support (1) different quantization schemes, (2) sparsity (and what type), (3) high GeMM throughput, and (4) fine-grained interleaving with the core; whether they use matrix or vector units to execute GeMMs; whether they are in- or near-core accelerators; and whether they require core hardware changes.

3. In-core accelerators using matrix operations. Traditional matrix units such as TMUL [37] and RASA [40] cannot deal with compressed tiles. To avoid the need for tile decompression, some in-core accelerator designs [39, 56, 61] including VEGETA [39] augment matrix units with support for specific structured sparsity patterns. Such an approach increases hardware complexity in the core (e.g., a larger matrix unit, more architectural registers, or changes in register renaming). Further, although this approach can increase the matrix throughput (*MOS*) by skipping some computations with zeros, our *Roof-Surface* analysis of Section 4 reveals that such an increase is unneeded for our kernels: most of them become memory-bound after leaving the vector-bound region. Note that VEGETA can be paired with software that conservatively transforms unstructured to structured sparsity, as the hardware does not support unstructured sparsity natively.

Other designs augment the matrix units with native support for more efficient lower bit quantization formats [38, 56]. However, such designs require extra hardware in the matrix unit for each one of the supported formats. Further, the hardware needs to be redesigned if a new, previously unseen, quantization format emerges. In contrast, DECA can support a rich set of quantization formats without requiring extra hardware for each one: it can change the values in its LUT array and/or use different scale factors.

4. In/near-core accelerators using vector operations. SPADE [19] and SAVE [23] are accelerators for sparse applications designed to be integrated with CPUs. Instead of relying on matrix units, they use vector units to execute the actual GeMM. While this approach might work for highly sparse matrices, using the high-throughput matrix units is necessary for the moderately sparse matrices found in ML models [80].

Overall, as shown in Table 8, DECA offers a unique combination of characteristics. It supports a rich set of quantization schemes combined with structured or unstructured sparsity. It enables high GeMM throughput by cooperating with the TMUL matrix units. Through speculative invocation, it is the first near-core accelerator design that enables fine-grained interleaving with the core. Finally, it introduces only a few changes to a core's pipeline, which could potentially be reused for other near-core accelerators.

5. Performance Models. Gables [33] is a related performance model for multiple compute domains and memory with a 2-D visualization. Gables models heterogeneous compute engines accessing memory independently and *in parallel*. In contrast, Roof-Surface models a *serial* (pipelined) combination of multiple engines, where one engine accesses memory and produces data for a second engine. As a result, Gables only uses Arithmetic Intensity (AI) with respect to memory for each engine, while the Roof-Surface defines AI from one engine to the other (e.g., AIXV). Overall, the two models target different settings and are orthogonal. Combining them into a single model is an interesting future direction.

11 Conclusion

To improve LLM inference in advanced CPU platforms with in-core GeMM engines and HBM, this paper made three contributions: the *Roof-Surface* performance model, the *DECA* near-core accelerator for ML-model decompression, and the TEPL ISA extension for out-of-order accelerator invocation to hide communication latency. Our evaluation showed that DECA is very effective. In a simulated 56-core Xeon 4 server with HBM, DECA accelerated the execution of compressed GeMMs by up to 4x over the use of optimized Intel software kernels. Further, DECA reduced the next-token generation time of Llama2-70B and OPT-66B by 1.6×—1.9×.

Acknowledgments

This work was supported in part by Intel Corporation and ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The work was done while the first author was an intern at Intel.

12 Appendix

12.1 Batch Sizes in LLM Inference with CPUs

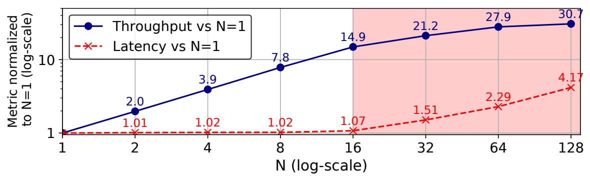


Figure 17: Llama2-70B next-token time (latency) and tokens-per-second (throughput) scaling for different batch sizes N.

This section examines what batch sizes are meaningful for LLM inference using modern CPU servers like SPR equipped with AMX extensions. Figure 17 shows how the next-token time (latency) and tokens-per-second (throughput) of the (uncompressed) Llama2-70B model scale with the batch size N. We see that as N scales from 1 to 16, the throughput increases linearly, while the latency remains constant. For N>16, which is the TMUL throughput saturation point (Section 2.3), the next-token time increases without linear gains in throughput. Although hyperscalers commonly use batch sizes of 64 in LLM serving with (multiple) GPUs [70], a single CPU has utility for smaller batch sizes (e.g., for low-latency single-user scenarios). For this reason, in this work we mainly focus on sizes from 1 to 16.

12.2 Effect of Batch Size on the Roof-Surface

We now discuss how the Roof-Surface and BORDs change as the batch size N changes from 1 to 16. Note from Equation 2 that N appears outside of the min-clause. Further, the AI_{XM} is not affected: A weight tile is only used in one matrix operation, since batch sizes from 1 to 16 can all be handled with a single matrix operation (Section 2.3). Although the operation has different FLOPS depending on the batch size, AI_{XM} is not affected. Finally, AI_{XV} also remains unchanged. Thus, as the batch size changes from 1 to 16, the min-clause does not change and the BORDs look the same. For example, if a kernel is VEC-bound for N=16, it is also so for N=1 and vice-versa. The 3D roof-surface itself just scales in height as N increases.

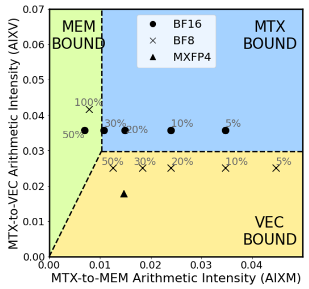


Figure 18: HBM BORD N=64.

As the batch size increases beyond 16, things change. Take N=64. First, the TMUL throughput saturates, and thus the height of the 3D roof-surface remains equal to the one for N=16. Second, each weight tile can now be reused in 4 matrix operations, increasing the AI_{XM} by 4. The AI_{XV} can also increase by 4 by decompressing only once per weight tile, and re-using the result for the other 3 matrix operations. The resulting BORD is shown in Figure 18. Note that the axes in the figure display a larger range of arithmetic intensities than in Figure 5a. We see that points are shifted up and right. However, many kernels still remain VEC-bound, even for this batch size that is less meaningful for low-latency CPU inference.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, et al. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Paul Alcorn. 2023. *Intel Details Sierra Forest and Granite Rapids Architectures, Xeon Roadmap.* https://www.tomshardware.com/news/intel-details-sierra-forest-and-granite-rapids-architecture-xeon-roadmap Accessed: 2025-09-04.

[3] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[4] Puneeth Bhat, José Moreira, and Satish Kumar Sadasivam. 2021. *Matrix-multiply Assist Best Practices Guide.* Technical Report. IBM, Tech. Rep., 2021.[Online]. Available: htt ps://www.redbooks.ibm.com.

[5] Arijit Biswas and Sailesh Kottapalli. 2021. Next-Gen Intel Xeon CPU-Sapphire Rapids. In *Hot Chips*, Vol. 33.

[6] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. 2020. What is the state of neural network pruning? *Proceedings of machine learning and systems* 2 (2020), 129–146.

[7] Nafea Bshara. 2024. AWS Trainium: The Journey for Designing and Optimization Full Stack ML Hardware. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 4. https://doi.org/10.1145/3620666.3655592

[8] Cagla Cakir, Ron Ho, Jon Lexau, and Ken Mai. 2015. Modeling and design of high-radix on-chip crossbar switches. In *Proceedings of the 9th International Symposium on Networks-on-Chip.* 1–8.

[9] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 3, Article 28 (Aug. 2014), 25 pages.

[10] Xinyu Chen, Yao Chen, Feng Cheng, Hongshi Tan, Bingsheng He, and Weng-Fai Wong. 2022. ReGraph: Scaling graph processing on HBM-enabled FPGAs with heterogeneous pipelines. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 1342–1358.

[11] Xiongjie Dai. 2025. GPU-Benchmarks-on-LLM-Inference. https://github.com/XiongjieDai/GPU-Benchmarks-on-LLM-Inference

[12] João PL de Carvalho, José E Moreira, and José Nelson Amaral. 2022. Compiling for the IBM matrix engine for enterprise workloads. *IEEE Micro* 42, 5 (2022), 34–40.

[13] Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. 2020. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proc. IEEE* 108, 4 (2020), 485–532.

[14] Haozheng Fan, Hao Zhou, Guangtai Huang, Parameswaran Raman, Xinwei Fu, Gaurav Gupta, Dhananjay Ram, Yida Wang, and Jun Huan. 2024. HLAT: High-quality Large Language Model Pre-trained on AWS Trainium. *arXiv preprint arXiv:2404.10630* (2024).

[15] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-shot. In *International Conference on Machine Learning.* PMLR, 10323–10337.

[16] Evangelos Georganas, Dhiraj Kalamkar, Sasikanth Avancha, Menachem Adelman, Cristina Anderson, Alexander Breuer, Jeremy Bruestle, Narendra Chaudhary, Abhisek Kundu, Denise Kutnick, Frank Laub, Vasimuddin Md, Sanchit Misra, Ramanarayan Mohanty, Hans Pabst, Barukh Ziv, and Alexander Heinecke. 2021. Tensor processing primitives: A programming abstraction for efficiency and portability in deep learning workloads. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–14.

[17] Evangelos Georganas, Dhiraj Kalamkar, Kirill Voronin, Abhisek Kundu, Antonio Noack, Hans Pabst, Alexander Breuer, and Alexander Heinecke. 2023. Harnessing Deep Learning and HPC Kernels via High-Level Loop and Tensor Abstractions on CPU Architectures. *arXiv preprint arXiv:2304.12576* (2023).

[18] Gerasimos Gerogiannis, Sriram Aananthakrishnan, Josep Torrellas, and Ibrahim Hur. 2024. HotTiles: Accelerating SpMM with Heterogeneous Accelerator Architectures. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE, 1012–1028.

[19] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. 2023. SPADE: A Flexible and Scalable Accelerator for SpMM and SDDMM. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 19, 15 pages. https://doi.org/10.1145/3579371.3589054

[20] Soroush Ghodrati, Sean Kinzer, Hanyang Xu, Rohan Mahapatra, Yoonsung Kim, Byung Hoon Ahn, Dong Kai Wang, Lavanya Karthikeyan, Amir Yazdanbakhsh, Jongse Park, Nam Sung Kim, and Hadi Esmaeilzadeh. 2024. Tandem processor: Grappling with emerging operators in neural networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2.* 1165–1182.

[21] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2022. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision.* Chapman and Hall/CRC, 291–326.

[22] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. 2019. SparTen: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 151–165.

[23] Zhangxiaowen Gong, Houxiang Ji, Christopher W Fletcher, Christopher J Hughes, Sara Baghsorkhi, and Josep Torrellas. 2020. SAVE: Sparsity-aware vector engine for accelerating DNN training and inference on CPUs. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 796–810.

[24] Zhangxiaowen Gong, Houxiang Ji, Yao Yao, Christopher W Fletcher, Christopher J Hughes, and Josep Torrellas. 2022. Graphite: optimizing graph neural networks on CPUs through cooperative software-hardware techniques. In *Proceedings of the 49th Annual International Symposium on Computer Architecture.* 916–931.

[25] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H. Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W. Lee, and Deog-Kyoon Jeong. 2020. A^3: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 328–341.

[26] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W Lee. 2021. ELSA: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 692–705.

[27] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[28] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[29] Simla Burcu Harma, Ayan Chakraborty, Elizaveta Kostenok, Danila Mishin, Dongho Ha, Babak Falsafi, Martin Jaggi, Ming Liu, Yunho Oh, Suvinay Subramanian, and Amir Yazdanbakhsh. 2024. Effective Interplay between Sparsity and Quantization: From Theory to Practice. *arXiv preprint arXiv:2405.20935* (2024).

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 770–778.

[31] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275

[32] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 981–991.

[33] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A roofline model for mobile SOCs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 317–330.

[34] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research* 22, 241 (2021), 1–124.

[35] Intel. 2022. Accelerate Artificial Intelligence (AI) Workloads with Intel Advanced Matrix Extensions (Intel AMX). https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-12/accelerate-ai-with-amx-sb.pdf

[36] Intel 2023. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Intel. Chapter 20: Intel AMX, Section 20.17.2 – Intel® Hyper-Threading Technology. Available at https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[37] Intel. 2024. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.*

[38] Jaeyong Jang, Yulhwa Kim, Juheun Lee, and Jae-Joon Kim. 2024. FIGNA: Integer Unit-Based Accelerator Design for FP-INT GEMM Preserving Numerical Accuracy. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE, 760–773.

[39] Geonhwa Jeong, Sana Damani, Abhimanyu Rajeshkumar Bambhaniya, Eric Qin, Christopher J Hughes, Sreenivas Subramoney, Hyesoon Kim, and Tushar Krishna. 2023. VEGETA: Vertically-integrated extensions for sparse/dense gemm tile acceleration on cpus. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE, 259–272.

[40] Geonhwa Jeong, Eric Qin, Ananda Samajdar, Christopher J Hughes, Sreenivas Subramoney, Hyesoon Kim, and Tushar Krishna. 2021. RASA: Efficient register-aware systolic array matrix engine for cpu. In *2021 58th ACM/IEEE Design Automation*

Conference (DAC). IEEE, 253–258.

[41] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–14.

[42] Christoforos Kachris. 2024. A Survey on Hardware Accelerators for Large Language Models. *arXiv preprint arXiv:2401.09890* (2024).

[43] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. 2019. A study of BFLOAT16 for deep learning training. *arXiv preprint arXiv:1905.12322* (2019).

[44] Dinesh Kalla, Nathan Smith, Fnu Samaah, and Sivaraju Kuraku. 2023. Study and analysis of chat GPT and its impact on different fields of study. *International journal of innovative science and research technology* 8, 3 (2023).

[45] Jeonghoon Kim, Jung Hyun Lee, Sungdong Kim, Joonsuk Park, Kang Min Yoo, Se Jung Kwon, and Dongsoo Lee. 2024. Memory-efficient fine-tuning of compressed large language models via sub-4-bit integer quantization. *Advances in Neural Information Processing Systems* 36 (2024).

[46] Yann LeCun, John Denker, and Sara Solla. 1989. Optimal brain damage. *Advances in neural information processing systems* 2 (1989).

[47] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. 2021. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing* 461 (2021), 370–403.

[48] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2023. AWQ: Activation-aware weight quantization for LLM compression and acceleration. *arXiv preprint arXiv:2306.00978* (2023).

[49] Liyang Liu, Shilong Zhang, Zhanghui Kuang, Aojun Zhou, Jing-Hao Xue, Xinjiang Wang, Yimin Chen, Wenming Yang, Qingmin Liao, and Wayne Zhang. 2021. Group Fisher pruning for practical network compression. In *International Conference on Machine Learning*. PMLR, 7021–7032.

[50] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 977–991.

[51] Liqiang Lu, Jiaming Xie, Ruirui Huang, Jiansong Zhang, Wei Lin, and Yun Liang. 2019. An efficient hardware accelerator for sparse convolutional neural networks on FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 17–25.

[52] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. Benchmarking and dissecting the NVIDIA Hopper GPU architecture. *arXiv preprint arXiv:2402.13499* (2024).

[53] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. NVIDIA tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 522–531.

[54] Nevine Nassif, Ashley O. Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexandra M. Kern, Bill Bowhill, David R. Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad M. Haq, Roopali Sharma, and Kevin Duda. 2022. Sapphire Rapids: The next-generation Intel Xeon scalable processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. IEEE, 44–46.

[55] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. 2021. The Design Process for Google's Training Chips: TPUv2 and TPUv3. *IEEE Micro* 41, 2 (2021), 56–63. https://doi.org/10.1109/MM.2021.3058217

[56] NVIDIA. 2024. NVIDIA Blackwell Architecture Technical Brief. Retrieved 2024 from https://resources.nvidia.com/en-us-blackwell-architecture

[57] Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L Aragón, David Wentzlaff, and Margaret Martonosi. 2022. Tiny but mighty: designing and realizing scalable latency tolerance for manycore SOCs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 817–830.

[58] Subbarao Palacharla, Norman P Jouppi, and James E Smith. 1997. Complexity-effective superscalar processors. In *Proceedings of the 24th annual international symposium on Computer architecture*. 206–218.

[59] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH computer architecture news* 45, 2 (2017), 27–40.

[60] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2023. Splitwise: Efficient generative LLM inference using phase splitting. *arXiv preprint arXiv:2311.18677* (2023).

[61] Christodoulos Peltekis, Vasileios Titopoulos, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. 2024. DeMM: A Decoupled Matrix Multiplication Engine Supporting Relaxed Structured Sparsity. *IEEE Computer Architecture Letters* (2024).

[62] Alexandra Peste, Eugenia Iofinova, Adrian Vladu, and Dan Alistarh. 2021. AC/DC: Alternating compressed/decompressed training of deep neural networks. *Advances in neural information processing systems* 34 (2021), 8557–8570.

[63] Bita Darvish Rouhani, Nitin Garegrat, Tom Savell, Ankit More, Kyung-Nam Han, et al. 2023. OCP Microscaling Formats (MX) Specification. https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf

[64] Rajarshi Roy, Jonathan Raiman, Neel Kant, Ilyas Elkin, Robert Kirby, Michael Siu, Stuart Oberman, Saad Godil, and Bryan Catanzaro. 2022. *PrefixRL: Optimization of Parallel Prefix Circuits Using Deep Reinforcement Learning*. IEEE Press, 853–858. https://doi.org/10.1109/DAC18074.2021.9586094

[65] Sungju Ryu, Hyungjun Kim, Wooseok Yi, Eunhwan Kim, Yulhwa Kim, Taesu Kim, and Jae-Joon Kim. 2022. BitBlade: Energy-efficient variable bit-precision hardware accelerator for quantized neural networks. *IEEE Journal of Solid-State Circuits* 57, 6 (2022), 1924–1935.

[66] Marco Siracusa, Víctor Soria-Pardos, Francesco Sgherzi, Joshua Randall, Douglas J Joseph, Miquel Moretó Planas, and Adrià Armejach. 2023. A Tensor Marshaling Unit for Sparse Tensor Algebra on General-Purpose Processors. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1332–1346.

[67] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 689–702. https://doi.org/10.1109/HPCA47549.2020.00062

[68] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. 2017. The ARM Scalable Vector Extension. *IEEE micro* 37, 2 (2017), 26–39.

[69] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration, the VLSI Journal* 58 (2017), 74–81. http://vcl.ece.ucdavis.edu/pubs/2017.02.VLSIintegration.TechScale/.

[70] Jovan Stojkovic, Esha Choukse, Chaojie Zhang, Inigo Goiri, and Josep Torrellas. 2024. Towards Greener LLMs: Bringing Energy-efficiency to the Forefront of LLM Inference. *arXiv preprint arXiv:2403.20306* (2024).

[71] Qidong Su, Christina Giannoula, and Gennady Pekhimenko. 2023. The synergy of speculative decoding and batching in serving large language models. *arXiv preprint arXiv:2310.18813* (2023).

[72] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[73] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[74] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 97–110.

[75] Xiuying Wei, Yunchen Zhang, Yuhang Li, Xiangguo Zhang, Ruihao Gong, Jinyang Guo, and Xianglong Liu. 2023. Outlier suppression+: Accurate quantization of large language models by equivalent and optimal shifting and scaling. *arXiv preprint arXiv:2304.09145* (2023).

[76] Wikipedia. 2024. Sapphire Rapids Die Configurations. https://en.wikipedia.org/wiki/Sapphire_Rapids#Die_configurations.

[77] Finn Wilkinson and Simon McIntosh-Smith. 2022. An Initial Evaluation of Arm's Scalable Matrix Extension. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 135–140.

[78] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[79] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. 2023. Flash-LLM: Enabling Cost-Effective and Highly-Efficient Large Generative Model Inference with Unstructured Sparsity. *Proceedings of the VLDB Endowment* 17, 2 (2023), 211–224.

[80] Yifan Yang, Joel S Emer, and Daniel Sanchez. 2024. Trapezoid: A Versatile Accelerator for Dense and Sparse Matrix Multiplications. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 931–945.

[81] Binwei Yao, Ming Jiang, Diyi Yang, and Junjie Hu. 2023. Empowering LLM-based machine translation with cultural awareness. *arXiv preprint arXiv:2305.14328* (2023).

[82] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, Yan Yan, Beidi Chen, Guangyu Sun, and Kurt Keutzer. 2024. LLM Inference Unveiled: Survey and Roofline Model

Insights. *arXiv preprint arXiv:2402.16363* (2024).

[83] Hao Zhang, Dongdong Chen, and Seok-Bum Ko. 2019. New flexible multiple-precision multiply-accumulate unit for deep neural network training and inference. *IEEE Trans. Comput.* 69, 1 (2019), 26–38.

[84] Haopeng Zhang, Xiao Liu, and Jiawei Zhang. 2023. SummIt: Iterative text summarization via chatGPT. *arXiv preprint arXiv:2305.14835* (2023).

[85] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[86] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[87] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).

[88] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2024. Atom: Low-bit quantization for efficient and accurate LLM serving. *Proceedings of Machine Learning and Systems* 6 (2024), 196–209.

[89] Pengyuan Zhou, Lin Wang, Zhi Liu, Yanbin Hao, Pan Hui, Sasu Tarkoma, and Jussi Kangasharju. 2024. A survey on generative AI and LLM for video generation, understanding, and streaming. *arXiv preprint arXiv:2404.16038* (2024).

[90] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. 2023. A survey on model compression for large language models. *arXiv preprint arXiv:2308.07633* (2023).

[91] Zeyu Zhu, Fanrong Li, Gang Li, Zejian Liu, Zitao Mo, Qinghao Hu, Xiaoyao Liang, and Jian Cheng. 2024. MEGA: A Memory-Efficient GNN Accelerator Exploiting Degree-Aware Mixed-Precision Quantization. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 124–138.