BitWeaver: Read-Time Truncation in Memory

Garrett Gagnon
Samsung Semiconductor US
San Jose, California, USA
Rensselaer Polytechnic Institute
Troy, New York, USA
gagnog@rpi.edu

Srikanth Malla
Samsung Semiconductor US
San Jose, California, USA
srikanth.m@samsung.com

Yangwook Kang
Samsung Semiconductor US
San Jose, California, USA
yangwook.k@samsung.com

Liu Liu
Rensselaer Polytechnic Institute
Troy, New York, USA
liu.liu@rpi.edu

Abstract

Large language models (LLMs) have demonstrated remarkable capabilities in generating contextually relevant responses to prompts, but their inference performance is often constrained by a severe memory bottleneck in the self-attention stage. This bottleneck, which is inherently memory-bound, has led to extensive research into strategies for reducing the size of the Key-Value (KV) cache. Many existing approaches employ quantization to lower the data precision and reduce data volume. However, these methods are constrained by memory technology, which requires the data read from memory to match the data written into it. As a result, such strategies must apply reductions at write-time, limiting either model performance or achievable speedup.

We identify that enabling precision scaling at read-time – after data has been stored in memory – offers a unique opportunity to simultaneously reduce memory traffic and retain model accuracy. To this end, we develop a read-time precision-scaling mechanism and introduce BitWeaver, a hardware-enabled solution for in-memory truncation. BitWeaver dynamically reduces data precision during memory reads, achieving up to a 3× increase in memory throughput and execution speedups of up to 80% compared to baseline. Additionally, BitWeaver enhances sparse KV cache strategies by improving the efficiency of state-of-the-art sparsity techniques.

CCS Concepts

• **Computing methodologies** → **Representation of exact numbers**; Neural networks; *Natural language processing*; • **Hardware** → *Dynamic memory*.

Keywords

Large Language Models (LLMs), Precision Scaling, Multi-Precision Inference

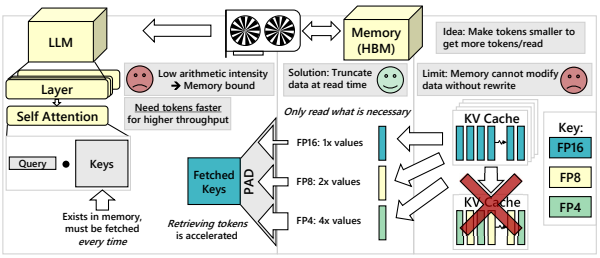

Figure 1: Overview of BitWeaver. Because self-attention is severely memory bound, we target accelerating token retrieval. With memory's limitations, we develop BitWeaver, a solution which uses truncation-in-memory to reduce memory read traffic.

ACM Reference Format:
Garrett Gagnon, Srikanth Malla, Yangwook Kang, and Liu Liu. 2025. BitWeaver: Read-Time Truncation in Memory. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3721145.3725779

1 Introduction

Transformer-based large language models (LLMs) have achieved remarkable performance across a wide range of natural language processing (NLP) tasks, including text generation, machine translation, and summarization [10, 18]. This success is largely attributed to the **self-attention** mechanism [45], which enables the model to capture intricate dependencies between tokens, embedding contextual influence during each forward pass. However, as the model size and sequence lengths increase to support complex downstream tasks, the memory and computational demands grow significantly.

To mitigate the computational burden during inference, modern LLMs employ **Key-Value (KV) caches** – precomputed representations that store attention-related vectors from previous steps. By reusing these cached values instead of recomputing them, KV caches reduce the redundant computation associated with processing long contexts, particularly for auto-regressive tasks. Despite their benefits, KV caches can introduce substantial memory overhead due to the sheer volume of data they retain, posing a critical challenge for latency-sensitive applications and resource-constrained deployments.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725779

To enhance the capabilities of LLMs, sequence lengths have been steadily increasing to accommodate broader contextual information beyond the explicit training data. For example, LLM inputs are often augmented with prior prompts, retrieved documents, or user-specific context to improve generation quality and relevance [19, 27]. This allows models to reference rich historical information during the generation of new tokens, thereby improving performance on tasks requiring extended memory over long dialogues or documents. The industry trend toward long-context models reflects the importance of this approach. Notably, Anthropic's Claude models exemplify this shift, with Claude 2 [4] offering a 100k-token context window and Claude 3 [5] extending it to 200k tokens. While future growth in sequence lengths may not follow the same exponential trend, these developments indicate that large context windows are here to stay.

However, the benefits of extended sequence lengths come at the cost of increased resource demands. For every additional token in the sequence, an extra Key and Value (KV) vector is generated for each attention head across all transformer layers. These KV vectors are stored in memory for access during each step of autoregressive inference, significantly inflating the memory footprint. Moreover, this growth in data volume exacerbates the memory bandwidth bottleneck during inference, as the KV caches must be read in full for each generated token.

The low arithmetic intensity of the self-attention operation further compounds this issue. Specifically, self-attention performs approximately 1 floating-point operation per byte of data when using FP16 precision, indicating that memory traffic dominates execution time rather than computation [49]. While increasing the sequence length does not inherently improve or worsen arithmetic intensity, it linearly increases the number of memory reads and operations required. As a result, extending sequence lengths amplifies the memory bandwidth burden, making it a critical bottleneck for efficient LLM inference. As highlighted by Figure 1, this means that token retrieval throughput is the limitation on computation throughput. In the absence of advances in physical bandwidth, optimizing memory traffic at the workload level is necessary to ensure scalability.

Existing strategies that mitigate memory traffic largely focus on techniques such as sparsification [1, 12, 26, 29, 36, 39, 48, 53, 54], quantization [9, 24, 30, 51], and dimensionality reduction [16, 24]. However, each of these approaches presents significant trade-offs. Sparsification can lead to a non-trivial loss in model accuracy, while quantization and dimensionality reduction, though effective in reducing data volume, can disrupt the memory access patterns, diminishing the overall efficiency of the KV cache. Furthermore, Figure 1 identifies how quantization and dimensional reduction must be applied some time before the next execution to accommodate the necessary write. These limitations highlight the need for new approaches that improve memory efficiency without compromising model performance or altering critical dataflows. Our work addresses this gap by proposing an in-memory truncation method for KV caches that strategically reduces memory traffic at read time. This approach aims to improve memory bandwidth utilization, enabling efficient LLM serving under long-sequence constraints.

To address the memory traffic bottleneck, we propose using truncation instead of conventional quantization for precision scaling. Shown in Figure 1, our approach performs truncation *in memory* to reduce the amount of data transferred during inference without duplicating data. By selecting the precision level for each token at read time, our method dynamically adjusts memory reads to balance accuracy and memory traffic. Additionally, we introduce a restorative adjustment term to mitigate truncation errors, ensuring minimal impact on model quality. This approach leverages the natural synergy between precision scaling and memory traffic reduction to optimize inference performance.

In this paper, we present **BitWeaver**, a low-cost hardware-based solution for enabling in-memory truncation at read time. Our key contributions are as follows:

(1) A hardware-aware in-memory truncation algorithm to dynamically adjust precision during inference.
(2) A specialized hardware implementation designed for compatibility with modern memory systems, such as HBM3.
(3) An accuracy retention mechanism that applies adjustment terms to minimize truncation errors.
(4) A detailed analysis of the trade-offs between memory traffic reduction and accuracy preservation.

2 Background

In this section, we discuss the role of memory in LLM serving, review the state of modern memory technology, and outline the need for pursuing memory techniques to address these challenges.

2.1 The Role of Memory in LLM Serving

Memory systems are critical to the performance of large language model (LLM) serving platforms, where managing and accessing vast amounts of data efficiently is paramount. Increasing sequence lengths in LLMs has significantly improved performance by enabling models to capture broader contexts during inference. However, this progress comes at the cost of an exponential increase in memory demands. For every additional token in a sequence, new Key and Value (KV) vectors are generated for every attention head across all layers, leading to dramatic growth in the memory footprint. These KV vectors are stored in memory and must be accessed repeatedly during token generation, resulting in substantial memory traffic.

The increasing reliance on longer sequences exacerbates the memory bottleneck because self-attention, a core operation in LLMs, is inherently *memory-bound* on most modern hardware. As shown in Figure 2, a simple roofline analysis [49] underscores this challenge: self-attention's arithmetic intensity is several orders of magnitude below the level required to fully utilize compute resources, even on cutting-edge systems. As a result, the performance of self-attention is limited by memory bandwidth, leaving much of the hardware's computational capacity underutilized.

This growing disparity between computation and memory bandwidth creates a critical performance bottleneck for LLM inference. To bridge this gap, two primary approaches can be considered: increasing arithmetic intensity or accelerating memory bandwidth. However, while advancements in physical memory technologies, such as HBM3, provide incremental improvements, these solutions alone are insufficient to meet the demands of ever-growing LLM workloads. Consequently, optimizing memory traffic and access

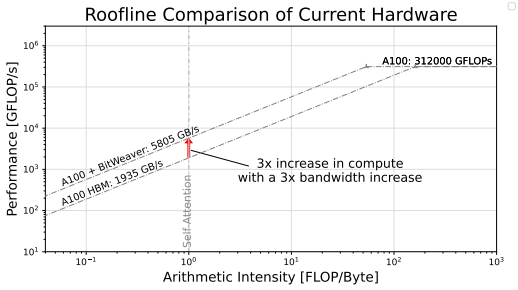

Figure 2: Roofline analysis of an A100 GPU with and without BitWeaver. With self attention exhibiting an arithmetic intensity of 1 FLOP/Byte, it is severely memory bound and benefits significantly from increases in effective bandwidth.

patterns at the workload level is essential to unlock the full potential of LLM serving platforms.

2.2 The State of Memory Technology

To meet the demands of modern AI workloads, high-bandwidth memory (HBM) has emerged as the premier memory solution for high-performance GPUs and accelerators [23]. Initially introduced in AMD's FURY line of GPUs [31], HBM has become the standard for flagship accelerators due to its unparalleled performance. The latest generation, HBM3 [7, 37], provides industry-leading bandwidth and capacity and is seeing commercial adoption. For example, NVIDIA's H100 GPU integrates 80 GB of HBM3, delivering an impressive 3.35 TB/s memory bandwidth [11]. This level of performance makes HBM3 indispensable for LLM serving platforms that process massive datasets with low latency requirements.

HBM achieves its exceptional performance through innovations in physical architecture and control mechanisms. By employing 3D stacking and increasing the number of memory channels, HBM significantly boosts bandwidth and reduces latency [23, 31]. Additionally, the introduction of pseudo-channels since HBM2 enables the first and second halves of a channel's bits to be accessed from separate rows [6]. This design facilitates non-uniform access patterns, improving memory efficiency for irregular workloads—a critical feature for AI and LLM serving.

2.3 The Need for Memory Acceleration

We propose to accelerate self-attention by increasing effective bandwidth, as the arithmetic intensity of this operation is inherently constant and inflexible. Unlike other layers in transformer models, which can increase arithmetic intensity through techniques like batching, self-attention does not benefit in the same way. For a given sequence, the operands – Queries, Keys, and Values – are unique to that sequence, preventing reuse or aggregation across batches. Consequently, the arithmetic intensity of self-attention remains fixed regardless of the batch size, making it a memory-bound operation. To accelerate self-attention, it is therefore necessary to focus on reducing memory traffic or enhancing memory bandwidth utilization.

Recent research highlights the importance of developing mechanisms to accelerate self-attention by addressing its memory demands. Two primary strategies have emerged: reducing memory traffic via sparsification and reducing the precision of stored KV data. The sparsification approach selectively reduces the size of the KV cache by predicting the importance of tokens and retaining only those deemed relevant to the current inference step. For instance, some methods evaluate past token performance to predict their relevance to future tokens [1, 12, 26, 53, 54]. While effective, this approach depends heavily on the quality of the prediction mechanism, introducing a risk of degraded accuracy if predictions are inaccurate or made too far in advance. On the other hand, reducing the precision of KV data is another promising avenue. Exemplar techniques [16, 24] leverage multiple precision schemes to minimize the KV cache footprint with minimal impact on model accuracy.

Despite their successes, these approaches suffer from two critical limitations: prediction distance and lack of synergy. Firstly, many sparsification methods require predictions to be made well in advance, tying accuracy directly to the quality of the predictor. This dependency increases the risk of errors and can lead to degraded model performance. Secondly, current precision reduction and sparsification techniques are often treated as mutually exclusive. Implementing both simultaneously typically requires significant data duplication or faces compatibility challenges. For example, sparsification decisions may conflict with precision adjustments, preventing efficient integration. Theoretically, precision scaling could complement sparsification by providing an intermediate option between dense and sparse representations, but this synergy remains unexplored.

3 Motivation

In this section, we outline the key design principles for effective memory acceleration in large language model (LLM) serving, emphasizing the need to reduce memory traffic, minimize decision latency, and ensure flexibility. Building on these principles, we provide the motivation for truncation-in-memory as a precision-scaling solution, highlighting its potential to address the memory bottlenecks inherent in self-attention and to complement existing optimization techniques.

3.1 Design Principles of Memory Acceleration

We aim for three design principles to overcome current limitations and improve the performance of LLM serving from effective memory acceleration. Firstly, the primary goal of memory acceleration is to **reduce traffic** in and out of memory. This reduction not only involves decreasing data volume but can also include more efficient memory access and data transfer mechanisms, as demonstrated by techniques like FlashAttention [14, 15]. By optimizing the management of memory traffic, unnecessary accesses are eliminated, mitigating the memory bottleneck and improving overall throughput. Without this fundamental reduction in traffic, no significant performance gains can be realized, making it the cornerstone of any effective memory design.

Secondly, we aim at **minimizing the decision distance** to ensure that decisions are based on the most up-to-date system state, enhancing their accuracy and relevance. Here, decision distance

refers to the temporal gap between when a decision (e.g., sparsification or precision adjustment) is made and when it is applied within the critical path of execution. While earlier decision-making often relies on incomplete information, deferring decisions until closer to their execution point guarantees access to the most comprehensive state data, enabling better-informed choices. This principle emphasizes the importance of dynamic, real-time adaptability in memory acceleration techniques.

Lastly, an ideal memory acceleration strategy should be flexible and seamlessly combined with other optimization techniques, enabling **synergistic implementations**. For example, sparsification and precision scaling are conceptually orthogonal; combining them allows for complementary benefits. Precision scaling can serve as an effective enhancement to sparsification, providing an intermediate solution for managing tokens that lie "on the fringe" of importance. This flexibility ensures that memory acceleration strategies can adapt to varying workloads and benefit from the strengths of multiple approaches simultaneously.

3.2 Motivation for Truncation-in-Memory

Traditional memory systems are inherently *precision-static*, meaning they only allow retrieval of data exactly as it was originally stored. While this is typically sufficient, it introduces inefficiencies in contexts where data importance diminishes over time. For such cases, the ability to retrieve a lower-precision representation of data would significantly reduce memory traffic, offering a path toward more efficient memory utilization.

Among precision-scaling methods, truncation and quantization are two common approaches, but they differ fundamentally in implementation and suitability. **Truncation** is a simple process that discards the least significant bits of a number, directly deriving lower-precision values from the original data. For instance, truncating the 4-bit binary representation of 15 (0b1111) to 2 bits results in 12 (0b1100). **Quantization**, on the other hand, adds a scaling operation to better match the data's dynamic range to the representable range. Using the same example, quantization of 15 within a range of 12-15 would yield a new value (e.g., 15).

Quantization, while effective in some scenarios, is less suited for multi-precision use because it generates new data rather than directly deriving from the original. This stores additional data/metadata for each precision variant or triggers rewrites, increasing memory overhead. In contrast, truncation leverages existing data and supports multi-precision use without data duplication, making it the natural choice for the core functionality of BitWeaver.

In addition, truncation is advantageous in systems using **floating-point** representations. The normalization inherent in floating-point numbers ensures that the most significant bits of the mantissa consistently encode valuable information, regardless of the magnitude. This reduces the risk of truncating to zero, a common issue in integer data where significant bits are sparsely utilized as numbers decrease in size. Floating-point truncation, therefore, enables precision reduction with minimal impact on data fidelity.

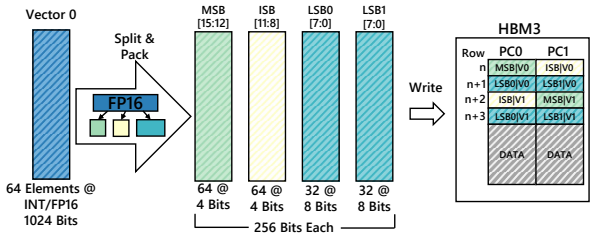

Figure 3: The packing of data within HBM3. Vectors of 64 elements are stored in 4 4-byte 8-burst groups. The packing strategy is then repeated/mirrored for subsequent vectors. This strategy maximizes throughput while minimizing latency.

4 Truncation-in-Memory

We first define our fundamental approach for performing truncation-in-memory. This section outlines our reconstruction-based strategy for truncation-in-memory. Additionally, we discuss its flexibility/adaptability to other realizations.

4.1 Effective Truncation

Write-time fragmentation is the critical mechanism that enables read-time precision scaling. As discussed previously, memory is inherently **precision-static**, meaning that it is not possible to read what was not previously stored. With just truncation, the data would still need to be truncated before being placed in memory.

However, we can circumvent this limitation by fragmenting instead of directly truncating. By fragmenting the data before it is written in, the data can be restored as necessary during reads. In this way, we are effectively truncating in memory. This does, though, create a need to reconstruct during the write process. Due to the fundamental IO limits of memory, this process cannot be completed within the memory itself. Therefore, this would not be an entirely in-memory process. Instead, data would need to be reconstructed somewhere nearer the processor.

We design a 3-tier multiprecision system for FP16 data. Shown in Figure 3, the fragmentation process is fairly simple for this system, with every number split into pieces of progressive degrees of precision. The splits are based the target precisions. In our design, we consider scaling FP16 into target sizes of 4, 8, and 16 bits. This creates 3 slices of 4, 4, and 8 bits. We then group elements based on the smallest supported read. In HBM3, that is 256 bits of data (@ 32 bits/pc x 8 reads/bursts). With this sizing, we group 64 values together (256 bits / 4 bits/element). We then make 4 memory blocks to hold all the data.

HBM's pseudo-channels presents an opportunity to reduce the latency impact of fetching from two separate rows. BitWeaver leverages this extensively by tiling across the pseudo-channel boundary. This prioritizes pairs of reads such that either can read simultaneously. This reduces the latency of stitching and buffering requirements.

In the typical 16-bit 1x, 2x, 4x throughput configuration, we use the packing strategy shown in Figure 3. This strategy places the MSB & ISB packs as well as the two LSB packs in the same

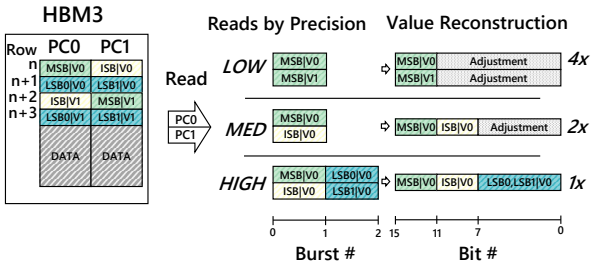

Figure 4: The process of reading values from memory. Because data has already been fragmented at the burst granularity, the data required for 4-, 8-, and 16-bit precision can be accessed independently. Vectors of FP16 values can then be reconstructed with only the data retrieved. Compared to high precision, medium and low precision achieve 2x and 4x throughput increase in values/second respectively

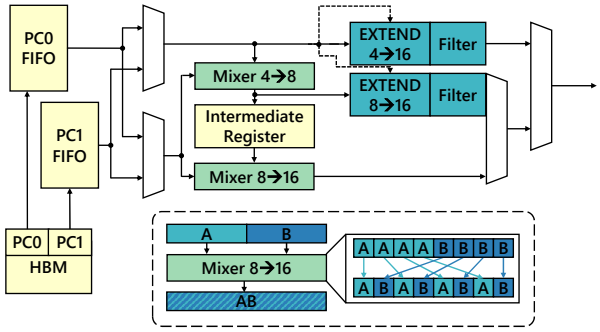

Figure 5: Diagram of the combiner unit. The combiner unit performs the reconstructs elements with minimal hardware. Data is received by buffers (already provisioned in the memory controller) and is stitched together by Mixer units. Mixers use simple wired logic, requiring no actual gates. Similarly, padding is directly concatenated to the lower precision data.

row across the PC boundary. This forms a two-row, two-pc mapping which is then repeated and mirrored for subsequent blocks of elements.

With data importance spread across reads, we can skip low-importance reads to reduce traffic. Figure 4 highlights how reading can be done for each of the three supported precisions. By skipping bursts, we reduce traffic and scale the precision down accordingly. Further, because we do not modify the memory architecture, we do not risk reducing bandwidth.

As shown in Figure 4, the 3 possible reconstructions are HIGH ({[15:0]}), MEDIUM ({[15:8], Padding}, and LOW ({[15:12], Padding}). In both LOW and MEDIUM precisions, padding is necessary to align the reduced bits for computation with HIGH precision values. In traditional truncation, this would be uniformly padded (0).

4.2 Adaptability

In order to scale precisions, we need a selection mechanism. In our system, we choose to use a proven strategy [54] for determining token importance. This strategy was used for sparsifying KV caches at the token level based on the importance of individual tokens for future predictions. In our case, we use it to create an additional tier of tokens which are of lesser importance but not sparsified.

This strategy may not be optimal as it selects at the token level. In the models being evaluated, the hidden dimension is thousands of elements long, much larger than the 64-element requirement minimum. This suggests that some additional memory savings may be attainable if sub-token ranges were scaled in precision. This is based on [28] which applied activation-aware quantization to weights. A strategy which identifies regions within the query vector of low & high importance and maps this to tokens being fetched could potentially recover greater accuracies at lower memory volumes.

The proposed implementation balances hardware complexity, latency, and granularity with the size and variety of precisions. It is, of course, not the only way to compose the system. Finer granularities and lower full-precision latency are possible when going between 8- & 16-bit precisions. This would, of course, come at the cost of a larger low-precision, which caps the maximum attainable speedup. More precision variety is also possible, but would come at the cost of additional reconstruction stages.

5 Hardware Implementation

Now that we have discussed the high-level approach of BitWeaver, we present the hardware needed to support it. As previously outlined, while the truncation effectively occurs in the memory through read skipping, the critical reconstruction occurs outside of the memory. In this section, we detail a specific deployment of this required hardware which exists between the processor and the memory. The remainder of this is organized as follows: we first discuss the specific hardware needed to reconstruct values in Section 5.1, then detail the operation of this hardware in Section 5.2, and finally explain the location and interface of the hardware in Section 5.3.

5.1 Reconstruction Hardware

The combiner unit is the hardware component which enables the reconstruction of fragmented data. As shown by Figure 5, it is composed of 3 stages: The input stage, reconstruction stage, and output stage. The exact operation of these stages are covered in Section 5.2.

The input stage is composed of two asynchronous FIFOs dedicated to receiving the data from their respective HBM3 PCs. These FIFOs must be large enough to accommodate the PC-to-PC latency present in HBM3, but not so large as to dramatically increase the cost of the hardware. In our case, we already have buffers provisioned in the memory controller. This allows us to remove these buffers in this design.

The recombination stage is relatively simple. It consists of two multiplexers to select primary and secondary sources, two mixer units, and an intermediate register. The mixer units are simple, only scattering the bits of the datalines directly through wire mapping. The intermediate register exists to hold an intermediate result and free the heads of the two FIFOs.

The output stage rectifies all values to the correct bit width (16-bits). This stage consists of adjusters, subnormal filters, and output

Algorithm 1 Reconstruction in combiner

```
repeat
  Select primary and secondary
  Read data into FIFOs.
  if LOW then
    Expand primary 4 to 16
    Filter value
    Send to processor
  end if
  Mix primary and secondary
  if MEDIUM then
    Expand mixer out 8 to 16
    Filter value
    Send to processor
  end if
  if HIGH then
    Store to intermediate
    Select secondary
    Mix intermediate and secondary
    Send to processor
  end if
until Empty
```

multiplexers. Discussed in Section 6, the adjusters and filters apply simple manipulations to the data before sending it to the processor. Without these units being active, the data is simply padded to 16 bits.

5.2 Combiner Operation

Because the combiner is a high-throughput device, execution must be kept fairly simple and parallel. With that, reconstruction can best be considered as repeated precision extensions until the required precision is achieved, at which point it is dispatched to the processor.

Algorithm 1 describes the process of reconstructing data within the combiner unit. Broadly, the process can be broken into reconstruction and padding. In reconstruction, blocks of data from memory are combined through simple wiring in *mixer units*. These rearrange the bits of data so that they are contiguous by element. In the case of going from 8-bit to 16-bit precision, we provision an intermediate register to catch the 8-Bit intermediate values which can then free the heads of the FIFOs. The intermediate values are then combined with the blocks of LSBs which now reside in the heads of the queues.

For padding, we must expand 4- & 8-bit values to 16-bits. In order to do this, we pad the remaining bits with the value in the corresponding adjustment register. This yields a complete 16-bit value. This result is then filtered with the *subnormal filter* before it is outputted. The subnormal filter nullifies any value which has leading 0's, which suggests that the value *could* be subnormal/zero. The motivation behind non-zero padding and filtering is further explained in Section 6.

When using the combiner unit, the adjustment term is set prior to reading as no information about the adjustment term is conveyed by the instruction. This does, however, limit how frequently

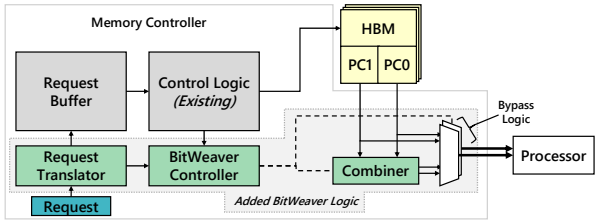

Figure 6: Simplified view of the memory controller integration. Requests are screened for BitWeaver control information, which is then managed by the BitWeaver controller. Bypass logic makes the interface transparent when unused.

adjustment regions can be changed. This is a reasonable tradeoff to reduce hardware cost and interface complexity.

5.3 Hardware Interface

The use case of this hardware is for accelerating memory-bound, saturated workloads. This places a significant burden on optimizing the throughput of this device. Because the interface is critical to maximizing throughput, we place the combiner unit in the memory controller, directly in sequence with memory IO, and add dedicated commands. This maximizes our throughput through the combiner and avoids data movement-related latency and power overhead.

Despite being a controller extension, BitWeaver remains minimally invasive as shown by Figure 6. It does this by reusing much of the existing hardware and providing a bypass mechanism. When standard requests are issued to the controller and executed, the output multiplexers directly pass the results from the output queue to the memory bus/processor, bypassing the combiner.

When requests arrive to reconstruct through the combiner unit, the request translator translates the request into the corresponding standard read requests. It also intercepts any overlapped reads. These occur when full precision data is requested for a block already being read out and only serve to signal the expected volume of data. The BitWeaver controller maintains a queue of reads to occur, which is used when the controller scheduler executes the corresponding reads from memory. When the correct data is at the heads of the FIFOs, the BitWeaver controller changes the multiplexers to the combiner output and begins streaming reconstructed data.

Because the adjustment term is not a static, universally-defined value, it is necessary to include a process for periodic updates. This functionality enables the parameterization of the adjustment term, which is essential to maximize accuracy.

For updating the adjustment term, we prioritize minimizing additional hardware & overhead. In our deployment of the BitWeaver algorithm, we consider a single value per model. In most LLM deployments, we expect the frequency of model changes to be low. This puts a low emphasis on fast adjustment updates. With the tight power and area constraints of memory controllers, we reason that this is an opportunity to minimize the cost. Thus, we design this interface to use as little new hardware as possible.

To minimize controller modifications, updating the adjustment term reuses most of the reading functionality. At a high level, to update, a read is performed to load data into one of the FIFOs.

Then specific values from the read data (bits 31:24 and bits 23:12 of burst 0) are set as the new adjustment values. This simplifies the hardware down to some additional request decoding and a short connection to extender inputs. Figure 5 highlights the connections of the primary elements to the mixers.

6 Truncation Error Mitigation

Although truncation alone provides reasonable accuracy with floating-point data, we identify a means to recover additional accuracy at a very low cost. This section details the process for improving the accuracy of approximated values. We first motivate its necessity, then describe the two key innovations & their hardware, and finally describe the optimization of truncation adjustment.

6.1 Truncated Value Adjustment

Truncation introduces error into a value. This error, though, is not entirely unpredictable. In fact, it is always *reductive to the magnitude*, which means $|x| \geq |\text{trunc}(x)|$ for all x. Thus, for every x, we have some expected error that must be negative ($E[|\text{trunc}(x)| - |x|] < 0$). Intuitively, we would expect that if we add the expected error, the accuracy would improve. Arithmetically implementations of this would be too expensive to be practical; performing addition or multiplication on all fetched values would massively increase the operations within self-attention.

However, non-zero padding presents an opportunity to recover some of the accuracy lost by direct truncation. First, because the fundamental error arises from replacing the lowest bits with 0's, the correct value is always achieved from within these bits. Put another way, the range of potential values which includes the original, correct value is entirely contained within the truncated bits. Second, because the bits are zero, direct bit masking can be applied to finely adjust the fetched values. This means that with a near-zero computational burden, we can mildly scale the fetched values.

When it comes to identifying a term to set as the lowest bits, a naive choice might be to apply a term equivalent to the expected value of the lost bits. With floating point numbers, because of the normalized mantissa, we observe that the distribution of bits on nonzero values is uniform. In this case, the expected value is half the dynamic range of the truncated bits (about 0x7F for 8 bits).

We can demonstrate that 0x7F minimizes the mean-squared error on a per-element basis with Figure 7. For values in the range $[1, 2]$, the percent of error in truncation is shown both with and without adjustment. Because all mantissas fall within this range, this is representative of all exponents. With adjustment, we see that it "splits" the error, ranging from around +10% to −10% compared to the unadjusted with errors ranging from 0% to −20%. This alone motivates the adjustment term as a per-element error minimizer.

6.2 Subnormal Filter

Applying an adjustment term is advantageous for most numbers but presents a slight risk; when numbers are zero or near-zero, applying an adjustment term will massively increase the magnitude of a stored value. Zeroes would become non-zero under a naive policy, which could increase computational load and disrupt accuracy. As well, zeroes are fairly common in keys and values, which forces it to be addressed.

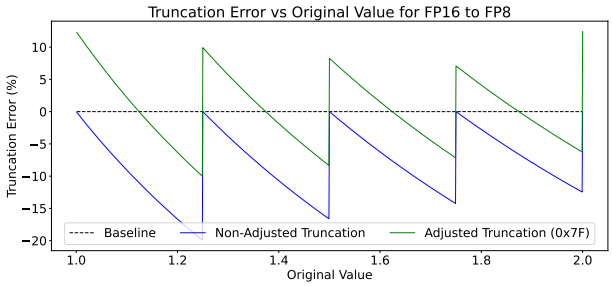

Figure 7: The effect of adjustment on truncation accuracy. Using an adjustment of 0x7F, we see that it "Splits the difference" and reduces the worst case truncation error and MSE.

If we could selectively identify and sparsify these subnormal values, we could retain the benefits of truncation adjustment without the potential performance degradation of these subnormal values. Fortunately, identification of subnormal values is simple, as they occur with an exponent of all 0's. Because the exponent is retained, the combiner hardware could precisely filter subnormal values based on this exponent. For this we design and implement *subnormal filters* which enables the combiner to achieve simple per-element sparsification.

The subnormal filters are a simple but effective hardware addition that prevents erroneous amplification of a given value. They work by detecting whether the bits read out suggest that the full value **could be** subnormal. When it could be, the entire value is nullified. The reasoning behind this is that dropping a value entirely was shown to be preferable to applying an adjustment term to a value which otherwise would have been zero. The implementation is a simple AND array on the output to the multiplexer.

6.3 Adjustment Term Optimization

As indicated earlier, a naive choice might be to apply a term equivalent to half the dynamic range of the truncated bits (about 0x7F for 8 bits). However, this does not consider the workload's characteristics.

We use a simple feedback-driven approach. In our testing, while statistical approaches provided a reasonable choice for an adjustment term, they did not always predict the optimal term. To better identify high-accuracy terms, we performed a sweep of adjustment values on a suite of lighter tasks to predict a generalized adjustment term for the model. Although individual tasks exhibited tendencies towards separate adjustment terms, what was more important was identifying any terms that presented with strong consensus.

It also appeared to be possible to optimize the adjustment term through gradient-based strategies. Both the derivative of the error of elements with respect to the adjustment term and the gradient of the error of dot products with respect to the per-element error are continuous. However, this was beyond the scope of this work.

7 Evaluation

To validate our algorithm and hardware design, we use a wide range of experiments. The organization of this section is as follows: We

Table 1: Accuracy comparison of several tasks using BitWeaver. All BitWeaver trials apply the FP16 to FP8 conversion to all tokens. Trials marked "+Truncation" apply a padding of 0 to all elements in the keys and values. Trials marked "+Adjustment" apply a model-specific padding specified in Table 3. GeoMean is taken on the relative error rate, with lower being better.

Model (Accuracy %)	COPA	WinoGrande	RTE	PiQA	MathQA	OpenbookQA	Hellaswag	Lambada	GeoMean
GPT-J 6B	86	64.3	54.9	75.4	26.6	29.0	49.5	68.2	-
+Truncation	85	63.3	50.5	74.6	25.4	27.8	47.8	67.8	103.83
+Adjustment	84	64.4	54.2	75.1	25.8	28.2	49.3	68.5	102.21
Mistral 7B	92	73.9	68.2	80.3	35.4	33.0	60.9	75.4	-
+Truncation	92	73.0	70.8	80.2	34.8	34.2	60.2	74.6	99.93
+Adjustment	92	73.7	70.8	80.3	35.2	33.0	60.6	75.0	99.36
LLaMa3 8B	89	72.6	69.7	79.7	40.6	34.8	60.2	76.0	-
+Truncation	88	74.0	70.0	79.8	38.5	34.4	59.3	77.2	100.40
+Adjustment	90	73.9	70.8	79.5	39.3	34.4	59.8	74.8	98.95

Table 2: Comparison of scores on summarization tasks. In all cases, adjustment significantly outperforms truncation alone.

Model (Rouge)	XSUM	CNN/DailyMail
GPT-J 6B	.0627	.154
+Truncation	.0520	.128
+Adjustment	.0618	.150
Mistral 7B	.0501	.096
+Truncation	.0472	.076
+Adjustment	.0495	.090
LLaMa3 8B	.0608	.124
+Truncation	.0024	.008
+Adjustment	.0587	.024

first explain our testing setup in Section 7.1, then present our results for our algorithm-based evaluations in Section 7.2, and finally present our results for our hardware evaluations in Section 7.3.

7.1 Methodology

For LLM evaluations, we used three different models, Mistral [22], LLaMa3-8B [17], and GPT-J [47]. These models were chosen based on their size, relatively strong performance, and different characteristics. We have four main evaluation suites that we perform for each model in all tasks. The first is the uniform performance evaluation, allowing for direct comparisons of characteristic individual runs. The second is the adjustment term sweeps, which applies BitWeaver uniformly to FP16 values and tests different values for the adjustment term. Third is the equivalent-density sparsity sweep, which compares different ratios of full:half:sparse tokens at equivalent memory footprints. Fourth is the full-representation density sweep, which gradually decreases the total memory density while maintaining full token representation with FP8 and FP4.

To evaluate these we use the open-source LM Evaluation Harness. We make minor modifications in the framework to inject attention modifications. We evaluate all suites on the following tasks: WinoGrande [25], COPA [40], RTE [46], Lambada [35], MathQA [3], OpenbookQA [32], Hellaswag [52], and PiQA [8]. Additionally, the summarization tasks CNN/Daily Mail [20] and XSUM [33] were used for uniform evaluations. These tasks were not used for sweep-based evaluation to show the generalizability of the adjustment term from simpler tasks. Additionally, no changes were made to the default evaluation configurations as set by the maintainers, as these are already standard protocols.

For all instances requiring token selection, we use the prediction method outlined in [54]. This was for two major reasons. First is that it is already an accepted method for ranking token importance, making it a compelling choice for selection. Second is that it limits the scope of this work. Although BitWeaver could enable different prediction strategies, it was beyond our scope for this work.

For hardware-based evaluations, we used several tools. A modified version DRAMSim3, which used the pseudo-channels HBM2E and HBM3, was used to get speedup & power characteristics as precision was scaled to different levels. To generate traces, we first generated the individual mapping of clusters of elements. We then randomly marked clusters as high or medium precision at sub-token granularity. This was done to demonstrate worst-case performance which is not model-specific.

To evaluate the hardware design, we first validated RTL functionality and then synthesized it. Having written the RTL for a proof-of-concept design, we validated it operated as expected through thorough testing with cocoTB on Verilator. With confirmed functionality, we synthesized the design with Synopsys Design Compiler using the FreePDK45 [44] 45nm pdk.

To estimate execution speedup, we used VIDUR [2]. This was because it was open source and thoroughly validated against hardware performance. However, it is based on *profiled performance* and therefore cannot directly evaluate a more theoretical hardware enhancement. However, the profiled data are granular enough so that we can specially modify the self-attention decoding time to reflect a reasonable improvement under BitWeaver.

We determine a reasonable improvement as follows. Using the analysis in Section 2, we estimate that because the self-attention stage is significantly memory-bound, memory speedups will proportionally speed up the attention computation itself. To prevent a complete oversimplification, we assume an 85% gain; that is, we assume that a 2x memory speedup will produce a 1.85x decoding speedup. We then create two new sets of pseudo-profiled data of

Table 3: FP8 adjustment terms. The selected terms are based on task consensus for the three tested models.

Model	Decimal Value	Hex Value
GPT-J 6B	112	0x70
Mistral 7B	80	0x50
LLaMa3 8B	192	0xC0

A100s for use in VIDUR: 2x & 2.9x effective bandwidth amplification under truncation in decoding. These were then used in place of the original data when simulating performance.

7.2 Algorithmic Evaluation

We first evaluate the accuracy impact of BitWeaver to confirm viable operating parameters. This section discusses the results of the four major algorithmic evaluations as introduced by Subsection 7.1 and highlights the ability for BitWeaver to decrease the theoretical memory traffic while preserving accuracy.

7.2.1 Uniform Application. The first set of evaluations is the application of BitWeaver uniformly to all Key & Value tokens. In Tables 1 & 2, we compare the baseline, BitWeaver with no adjustment, and BitWeaver with the model's best adjustment term for three models. It is important to note that although individual tasks had better adjustment terms, we only show the results from the same adjustment term for a given model. This is because we are interested in the transferability of adjustment terms at a model level. These selected adjustment terms are shown in Table 3.

In Table 1, we see that when uniformly applied, the adjustment term consistently results in higher accuracies. Occasionally, the accuracy even improved relative to the baseline performance. We feel this highlights the importance of applying this adjustment to truncated values, as many times it recovers several percentage points of accuracy over the truncated-only values.

Despite the promising performance, it should be noted that occasionally, the accuracy decreased relative to the non-adjusted case. While the occurrences are minimal, it highlights an imperfection in the strategy. We believe it suggests that there is more to understand about the adjustment term and its role in maintaining accuracy.

Moving towards generalization, Table 2 shows the scores of two summarization tasks across models. In these evaluations, the adjustment term was only selected based on the performance of the simpler tasks in Table 1. Critically, we see that despite not using these tasks for feedback, we see significant score improvement of the adjusted terms over the baseline truncated values. This highlights both the transferability of the adjustment term at the model level as well as the applicability of the adjustment term to more than just token-prediction tasks.

7.2.2 Adjustment Optimization. Identifying adjustment terms is a major aspect of our implementation. Our second evaluation are sweeps adjustment terms for uniformly applied BitWeaver on simple tasks. Put another way, we apply truncation uniformly to all keys & values, but vary the adjustment to see its effect on accuracy. We plot the results for several models in Figure 8. This allows us

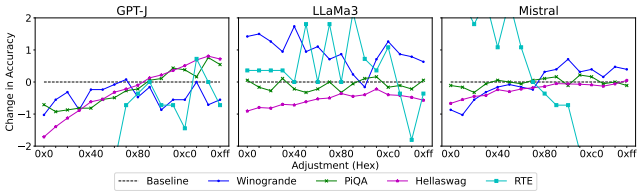

Figure 8: The effect of the adjustment term on accuracy across models. Within this subset of tasks, we see that despite tasks having vastly different trends, they tend to have consensus at some value.

to identify points of consensus where the model is most accurate across a range of tasks.

This evaluation uncovers interesting behavior at the task level. First, some tasks, such as Hellaswag [52], exhibit strong correlation to the adjustment term. We see that across models, Hellaswag generally improves in accuracy as the adjustment term increases in magnitude. Similarly, PiQA [8] remains relatively tolerant to changes in the adjustment. Finally, some tasks exhibit entirely different behavior dependent on the model, like RTE [46].

This task-level behavior was somewhat unexpected, but highlights the interaction between model, task, and adjustment term. From this, though, we see that the task-transferability of adjustment terms is low, with no general trends arising across tasks. Despite this, points of consensus appeared that showed agreement between tasks at specific values. Furthermore, these points were unique to specific models. This leads us to determine that the model is a major factor in selecting the adjustment term.

To use Figure 8 to identify the adjustment terms per model, we searched for the regions most accurate of the least accurate tasks and identified a region that maximizes the accuracy of multiple tasks. Effectively, we search within the least accurate task to approximately maximize all tasks' accuracies. The resultant values which consider all tested tasks are shown in Table 3.

7.2.3 FP16 & FP8. Next we evaluate the accuracy as a function of memory traffic for different percentages of FP16 and FP8 tokens. Effectively, it progressively sparsifies the tokens being used for generation while representing the accuracy for different percentages of data fetched (Analogous to sparsity). We aim to validate that utilizing adjusted FP8 is a stronger tool than sparsification alone. To do this, we employ a proven sparsification strategy to isolate the effect of our precision-scaling strategy. The sparsity scheme we choose to use is H2O [54].

In Figure 9, we show that utilizing adjusted FP8 consistently achieves higher accuracy values compared to the equivalent sparsity of FP16 tokens. Further, when comparing mixtures of full- and half-precision tokens (FP16 & FP8), we see that it is consistently more advantageous to accept lower precision tokens than to sparsify more. This demonstrates that when considering FP16 and adjusted FP8, greater degrees of token representation is more advantageous than higher precision.

7.2.4 FP8 & FP4. Seeing as greater degrees of representation was advantageous in the FP16-FP8 evaluation, it was natural to evaluate

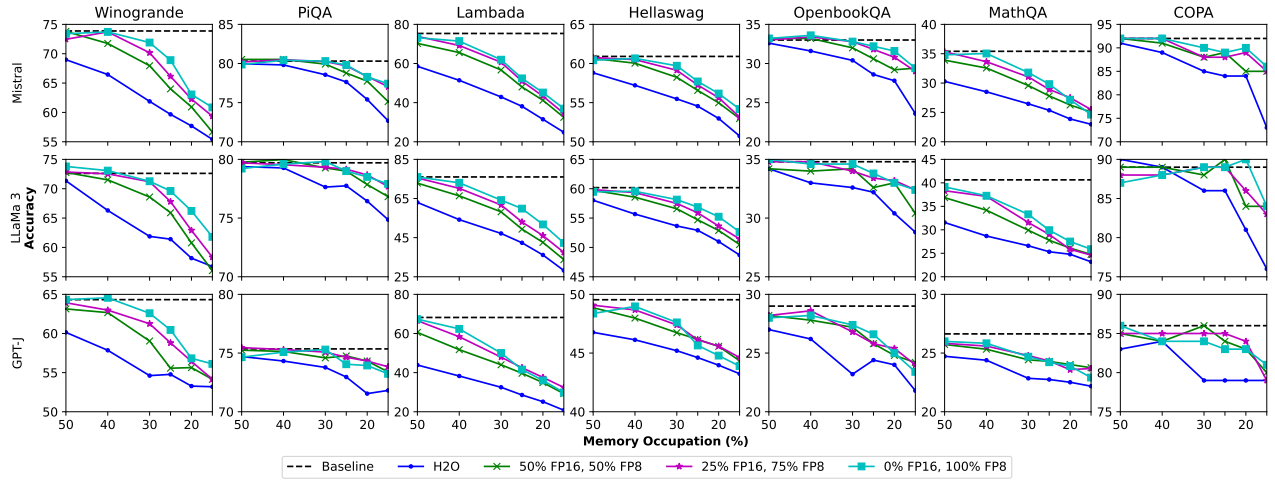

Figure 9: Accuracy impact of various memory footprints for several mixtures of FP16 and adjusted FP8 tokens. Results are compared across memory footprints and highlights that increased percentages of FP8 tokens preserve greater accuracy than just sparse FP16. H2O is representative of the H2O algorithm [54] with exclusively FP16 tokens. All others implement an adjusted policy which preserves a given percentage of FP8 tokens relative to the FP16 tokens.

Table 4: Synthesized results for a single combiner in 45nm. Dynamic power is based on peak switching activity for all transistors.

Parameter	Value	Unit
Area	20950	μm^2
Dynamic Power	4.48	mw
Leakage	63.29	μw

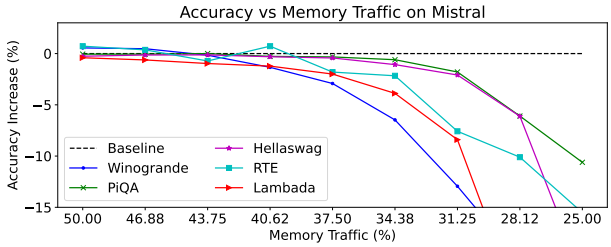

Figure 10: The effect of different ratios of FP8 & FP4 tokens reported as a function of memory traffic on Mistral-7B. 50% represents entirely FP8 while 25% represents entirely FP4. We see that unlike the case of balancing FP8 & FP16, there is a limit to token representation in lower precision. However, we see that a significant amount of tokens can be represented by FP4 without degrading accuracy. This motivates a dynamic multi-precision system.

if this trend continued between FP8 & FP4. In this evaluation, rather than consider sparsity with various ratios of tokens, we instead consider no sparsity with and vary the ratio as a function of memory traffic percentage. Therefore, the memory traffic varies between 50% (all FP8) and 25% (all FP4). In this way, we can compare trends across several tasks together.

The target of this evaluation is to validate that a mixture of precisions retains substantial accuracy while reducing the overall memory traffic. We see from Figure 10 that this is confirmed; The accuracy in many tasks can be maintained while injecting large amounts of FP4 tokens. Simpler tasks like PiQA and Hellaswag maintain near-baseline accuracy at 34% traffic while other tasks can only tolerate FP4 tokens down to around 40% memory traffic. This is a positive result, as it highlights that the lossy FP4 of BitWeaver can be used to preserve model accuracy while further reducing the memory accesses.

7.3 Hardware Evaluation

In this subsection, we show that BitWeaver can provide theoretical speedups of up to 80% through simulated hardware performance. We first verify the memory access pattern regularity is maintained via DRAMSim3 and then show that the accelerated bandwidth speeds up LLM Inference systems.

7.3.1 Hardware Synthesis. Our synthesis results confirm our design to be reasonable for implementation. The results are presented in Table 4. As outlined previously, we synthesized our design in 45nm, which is orders of magnitude less power and area efficient than modern processes. Despite this, our synthesized design is reasonably sized, at less than $0.021mm^2$. Furthermore, the power, which assumes maximum switching activity, represents less than a 1% increase in channel power.

7.3.2 Memory Access Evaluation. We must verify that the sparse memory access pattern is regular and hardware-friendly. This is critical as an irregular access pattern could waste cycles/activations. Therefore, despite a theoretical speedup from reduced reads, a poor-quality pattern could fail to improve read times. We thus need to confirm read saturation at a low level. In addition, a consistent

Table 5: Results for single-channel BitWeaver performance for progressively finer quantities of elements. We observe that even at fine granularities, bandwidth is saturated without any substantial power increase. FP16, FP8, and FP4 refer to reading the entire sequence of values at that precision.

	Power (mW)			Bandwidth (GB/s)		
# Vals	FP16	FP8	FP4	FP16	FP8	FP4
16k	454.3	454.2	455.9	29.8	29.6	29.5
8k	454.3	454.0	442.8	29.7	29.6	29.5
4k	454.0	440.8	442.4	29.6	29.6	29.5
2k	441.0	440.4	442.0	29.7	29.6	29.5

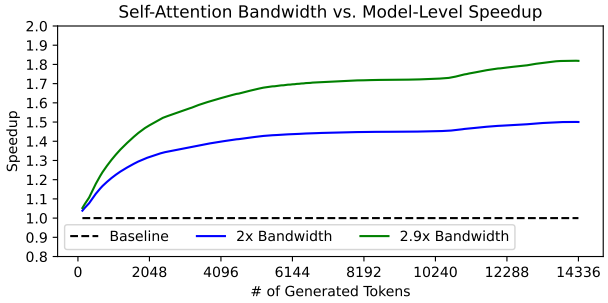

Figure 11: Plot of execution time for requests of varying decoding lengths. All have a prefill context of 512 tokens. We see that at this context, the speedup remains relatively consistent.

memory speedup factor would allow for full system evaluation with existing tools.

Table 5 highlights that BitWeaver's memory access pattern is hardware-friendly. We generate traces of fetching a given number elements at each of the possible precisions and measure the achieved bandwidth, power use, and actual numbers of reads. Table 5 shows the achieved "real" bandwidth of HBM2 at several access patterns. We see that even with small data volumes, the bandwidth saturates to the hardware limit regardless of the pattern. Additionally, power does not unduly rise due to inefficient memory accesses.

7.3.3 System-Level Performance. With the regularity shown for the memory access pattern, we evaluate the performance impact of decoding acceleration. Although we can show an expected improvement in decoding speed, it is unclear what the full execution improvement would be. This is primarily due to the linear layers of LLM's having a significant performance impact while not being influenced by BitWeaver in this implementation.

Figure 11 shows that bandwidth amplification via BitWeaver produces measurable execution speedups. Using a simple BitWeaver pattern that produces a 2x memory bandwidth amplification results in an execution speedup of over 1.4x when generating about 4,000 tokens or more. With smaller generated sequences, the performance improvement was still substantial, with 1.2-1.4x speedups typical in the sub-3000 token regime.

BitWeaver for 2.9x bandwidth amplification showed even greater performance gains. When generating about 3,000 tokens or more, the speedup achieved was more than 1.6x, even exceeding 1.8x when generating large quantities of tokens.

8 Related Work

Attention sparsity is sparsification of the underlying attention calculation, not inherently reducing the size of any stored tokens. It still reduces the memory traffic and could be coupled with BitWeaver. SpAtten [48] progressively removes tokens from atternion calculations based on their importance in earlier layers. DOTA [38] uses low-rank linear transformations of queries and keys to predict token importance and only calculates attention for significant outputs. Similarly, SparQ [39] uses an approximation of attention to predict and select tokens of high importance. InfiniGen [26] enables offloading the KV cache to CPU memory by using an effective prediction mechanism to identify strong attention scores and prefetch them in time for computation. Token-picker [36] develops a strategy which predicts the score for tokens prior to calculation in order to prune out unnecessary calculations. All these methods could be readily combined with BitWeaver, as they provide an token ranking mechanism.

Sparsification of KV caches is another form of attention sparsity primarily targeting reducing the KV cache size. Instead of choosing to retrieve fewer tokens at compute time, the tokens have been evicted from the cache. Similar to general attention sparsification, KV cache sparsification algorithms are highly applicable as they create a ranking methodology which can be applied as tiers for precision. One such algorithm, H2O [54], is a work which proposes using past performance of tokens to identify "Heavy Hitters" and sparsifies other tokens. In this work, we directly extend H2O to create precision tiers. KeyFormer [1] takes a similar approach and develops a score mechanism which factors in the past performance of tokens and retains high scored tokens. Q-Hitter [53] improves H2O by prioritizing maintaining tokens which quantize well, thereby allowing the cache itself to be more quantizable. CORM [12] uses the previous queries and their attention to predict which tokens must remain in the cache based on current queries. This could potentially be adapted as a ranking scheme for BitWeaver. Scissorhands [29] progressively drops tokens from the KV cache based on the "importance" of that token to past attention calculations.

Multiprecision KV caches is another approach to reducing their footprint. GEAR [24] proposes using multiple precisions to fully represent the KV cache by having each successive tier minimize the error from all previous ones. We, conversely, avoid data duplication and computation by reusing slices of the original data for multiple precisions. MiKV [51] uses an ultra-low precision to avoid outright sparsifying stored values. LESS [16] applies the H2O [54] algorithm to sparsify tokens, but then uses a low rank approximation of the sparsified elements to recover significant accuracy. KIVI [30] identifies that keys and values have separate quantization preferences and develops a low-loss 2-bit quantization scheme around it. Another work [9] identifies that activations are difficult to quantize due to outliers arising from attention attempting to prevent updates to the state and effectively mitigates this and quantizes activations down to INT8.

Quantization methods are widely applicable across domains.For post-training quantization (PTQ), which generally more applicable to activation quantization, VS-Quant [13] proposes quantization at the vector level (16-64 elements at a time). This method maintains hardware-friendliness through low-bitwidth scale factors. For values generated at execution time, though, calculating and maintaining the metadata can be costly. We skip identifying per-vector scale factors and just read a segment of individual elements to achieve a lower precision without arithmetic. MX [41] is a proposed method to share "microexponents" between groups of floating-point numbers. This could potentially be integrated with BitWeaver as we focus primarily on the mantissa and not the exponent.

Truncation is an alternative to quantization which we use in this work. One work which aims to reduce IO traffic [42] makes use of truncation by truncating and compressing floating-point values before writing to memory and decompressing upon retrieval. SmartQuant [50] explores slicing elements at each bit to offer a wider range of available precisions. Additionally, it provisions a more advanced reconstruction hardware and uses CXL to enable greater transparency to the host device. We conversely implement our design on host memory by splitting data into only 3 chunks to reduce granularity and reconstruction cost. Furthermore, we make use of HBM's pseudo-channels to improve hardware friendliness. SDP [21] quantizes values down to INT8, and then selects chunks of INT4 data to compute with to meet computational budgets. Its priority is the compute budget and focuses on CNN models. We prioritize memory bandwidth within the self-attention stage of transformer-based LLMs and apply no quantization to values to maintain the original, full-precision value.

Attention acceleration also is explored beyond the previous two categories. FlashAttention 1 [15] & 2 [14] reformulate and optimize the execution flow of self-attention to reduce memory traffic and better utilize compute resources. FlashAttention-3 [43] uses some optimizations for tensor cores, but also implements lower-precision execution. Sparse FlashAttention [34] extends the original FlashAttention [15] algorithm to accommodate dynamic sparsity in long sequences. On the other hand, BitWeaver does not change anything within the compute pattern of self-attention or any other applied workload. Thus, it is compatible with these

9 Conclusion

We present BitWeaver, a low-cost hardware-based approach to enable truncation-in-memory at read-time. We apply this utility to reduce data volume moving from memory during self-attention to realize 3x improvements in effective bandwidth, translating to an 80% increase in full model inference throughput. Further, we show this to be an adaptable strategy with the potential for numerous other applications.

Acknowledgments

The authors would like to the the members of Samsung MSL for their consistent and impactful feedback throughout the course of this research. The authors would also like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This work was supported in part by the Samsung Global Research Outreach Program and the National Science Foundation under Grant No. 2442271.

References

[1] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems* 6 (2024), 114–127.

[2] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A Large-Scale Simulation Framework For LLM Inference. *Proceedings of Machine Learning and Systems* 6 (2024), 351–366.

[3] Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. MathQA: Towards Interpretable Math Word Problem Solving with Operation-Based Formalisms. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 2357–2367. https://doi.org/10.18653/v1/N19-1245

[4] Anthropic. 2023. Claude 2. https://www.anthropic.com/news/claude-2. [Accessed 15-01-2025].

[5] Anthropic. 2024. Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family. [Accessed 15-01-2025].

[6] JEDEC Solid State Technology Association. 2021. *High Bandwidth Memory DRAM (HBM1, HBM2).* Technical Report. JEDEC.

[7] JEDEC Solid State Technology Association. 2023. *High Bandwidth Memory DRAM (HBM3).* Technical Report. JEDEC.

[8] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. 2020. PIQA: Reasoning about Physical Commonsense in Natural Language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*.

[9] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. 2023. Quantizable transformers: Removing outliers by helping attention heads do nothing. *Advances in Neural Information Processing Systems* 36 (2023), 75067–75096.

[10] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. 2024. Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference. arXiv:2403.04132 [cs.AI]

[11] Jack Choquette. 2023. Nvidia hopper h100 gpu: Scaling performance. *IEEE Micro* 43, 3 (2023), 9–17.

[12] Jincheng Dai, Zhuowei Huang, Haiyun Jiang, Chen Chen, Deng Cai, Wei Bi, and Shuming Shi. 2024. Sequence can Secretly Tell You What to Discard. *arXiv preprint arXiv:2404.15949* (2024).

[13] Steve Dai, Rangha Venkatesan, Mark Ren, Brian Zimmer, William Dally, and Brucek Khailany. 2021. Vs-quant: Per-vector scaled quantization for accurate low-precision neural network inference. *Proceedings of Machine Learning and Systems* 3 (2021), 873–884.

[14] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691* (2023).

[15] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[16] Harry Dong, Xinyu Yang, Zhenyu Zhang, Zhangyang Wang, Yuejie Chi, and Beidi Chen. 2024. Get More with LESS: Synthesizing Recurrence with KV Cache Compression for Efficient LLM Inference. *arXiv preprint arXiv:2402.09398* (2024).

[17] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[18] Clémentine Fourrier, Nathan Habib, Alina Lozovskaya, Konrad Szafer, and Thomas Wolf. 2024. Open LLM Leaderboard v2. https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard.

[19] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval Augmented Language Model Pre-Training. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 3929–3938. https://proceedings.mlr.press/v119/guu20a.html

[20] Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching machines to read and comprehend. *Advances in neural information processing systems* 28 (2015).

[21] Kai Huang, Bowen Li, Dongliang Xiong, Haitian Jiang, Xiaowen Jiang, Xiaolang Yan, Luc Claesen, Dehong Liu, Junjian Chen, and Zhili Liu. 2023. Structured dynamic precision for deep neural networks quantization. *ACM Transactions on Design Automation of Electronic Systems* 28, 1 (2023), 1–24.

[22] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[23] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*. IEEE, 1–4.

[24] Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. 2024. Gear: An efficient kv cache compression recipefor near-lossless generative inference of llm. *arXiv preprint arXiv:2403.05527* (2024).

[25] Sakaguchi Keisuke, Le Bras Ronan, Bhagavatula Chandra, and Choi Yejin. 2019. WinoGrande: An Adversarial Winograd Schema Challenge at Scale.

[26] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 155–172.

[27] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[28] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. *Proceedings of Machine Learning and Systems* 6 (2024), 87–100.

[29] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2024. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems* 36 (2024).

[30] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750* (2024).

[31] Joe Macri. 2015. AMD's next generation GPU and high bandwidth memory architecture: FURY. In *2015 IEEE hot chips 27 symposium (HCS)*. IEEE, 1–26.

[32] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a Suit of Armor Conduct Electricity? A New Dataset for Open Book Question Answering. In *EMNLP*.

[33] Shashi Narayan, Shay B. Cohen, and Mirella Lapata. 2018. Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization. *ArXiv* abs/1808.08745 (2018).

[34] Matteo Pagliardini, Daniele Paliotta, Martin Jaggi, and François Fleuret. 2023. Fast attention over long sequences with dynamic sparse flash attention. *Advances in Neural Information Processing Systems* 36 (2023), 59808–59831.

[35] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. 2016. The LAMBADA dataset. https://doi.org/10.5281/zenodo.2630551

[36] Junyoung Park, Myeonggu Kang, Yunki Han, Yang-Gon Kim, Jaekang Shin, and Lee-Sup Kim. 2024. Token-Picker: Accelerating Attention in Text Generation with Minimized Memory Transfer via Probability Estimation. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[37] Myeong-Jae Park, Jinhyung Lee, Kyungjun Cho, Jihwan Park, Junil Moon, Sung-Hak Lee, Tae-Kyun Kim, Sanghoon Oh, Seokwoo Choi, Yongsuk Choi, Ho Sung Cho, Taesik Yun, Young Jun Koo, Jae-Seung Lee, Byung-Kuk Yoon, Young-Jun Park, Sangmuk Oh, Chang Kwon Lee, Seong-Hee Lee, Hyun-Woo Kim, Yucheon Ju, Seung-Kyun Lim, Kyo Yun Lee, Sang-Hoon Lee, Woo Sung We, Seungchan Kim, Seung Min Yang, Keonho Lee, In-Keun Kim, Younghyun Jeon, Jae-Hyung Park, Jong Chan Yun, Seonyeol Kim, Dong-Yeol Lee, Su-Hyun Oh, Jung-Hyun Shin, Yeonho Lee, Jieun Jang, and Joohwan Cho. 2022. A 192-Gb 12-high 896-GB/s HBM3 DRAM with a TSV auto-calibration scheme and machine-learning-based layout optimization. *IEEE Journal of Solid-State Circuits* 58, 1 (2022), 256–269.

[38] Zheng Qu, Liu Liu, Fengbin Tu, Zhaodong Chen, Yufei Ding, and Yuan Xie. 2022. Dota: detect and omit weak attentions for scalable transformer acceleration. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 14–26.

[39] Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. 2023. Sparq attention: Bandwidth-efficient llm inference. *arXiv preprint arXiv:2312.04985* (2023).

[40] Melissa Roemmele, Cosmin Adrian Bejan, and Andrew S Gordon. 2011. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *2011 AAAI spring symposium series*.

[41] Bita Rouhani, Ritchie Zhao, Venmugil Elango, Rasoul Shafipour, Mathew Hall, Maral Mesmakhosroshahi, Ankit More, Levi Melnick, Maximilian Golub, Girish Varatkar, Lei Shao, Gaurav Kolhe, Dimitry Melts, Jasmine Klar, Renee L'Heureux, Matt Perry, Doug Burger, Eric Chung, Zhaoxia (Summer) Deng, Sam Naghshineh, Jongsoo Park, and Maxim Naumov. 2023. With shared microexponents, a little shifting goes a long way. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[42] Vijay Sathish, Michael J Schulte, and Nam Sung Kim. 2012. Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. 325–334.

[43] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608* (2024).

[44] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. 2007. FreePDK: An Open-Source Variation-Aware Design Kit. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education (MSE '07)*. IEEE Computer Society, USA, 173–174. https://doi.org/10.1109/MSE.2007.44

[45] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[46] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In the Proceedings of ICLR..

[47] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

[48] 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.

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

[50] Rui Xie, Asad Ul Haq, Linsen Ma, Krystal Sun, Sanchari Sen, Swagath Venkataramani, Liu Liu, and Tong Zhang. 2024. SmartQuant: CXL-Based AI Model Store in Support of Runtime Configurable Weight Quantization. *IEEE Computer Architecture Letters* (2024).

[51] June Yong Yang, Byeongwook Kim, Jeongin Bae, Beomseok Kwon, Gunho Park, Eunho Yang, Se Jung Kwon, and Dongsoo Lee. 2024. No token left behind: Reliable kv cache compression via importance-aware mixed precision quantization. *arXiv preprint arXiv:2402.18096* (2024).

[52] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a Machine Really Finish Your Sentence?. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

[53] Zhenyu Zhang, Shiwei Liu, Runjin Chen, Bhavya Kailkhura, Beidi Chen, and Atlas Wang. 2024. Q-Hitter: A Better Token Oracle for Efficient LLM Inference via Sparse-Quantized KV Cache. *Proceedings of Machine Learning and Systems* 6 (2024), 381–394.

[54] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Zhangyang Wang, and Clark Barrett. 2024. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems* 36 (2024).