BASS: Batched Attention-optimized Speculative Sampling

Haifeng Qian[1,*] **Sujan Kumar Gonugondla**[1,*] **Sungsoo Ha**[2] **Mingyue Shang**[1]
Sanjay Krishna Gouda[1] **Ramesh Nallapati**[3] **Sudipta Sengupta**[1] **Xiaofei Ma**[1]
Anoop Deoras[1]

Abstract

Speculative decoding has emerged as a powerful method to improve latency and throughput in hosting large language models. However, most existing implementations focus on generating a single sequence. Real-world generative AI applications often require multiple responses and how to perform speculative decoding in a batched setting while preserving its latency benefits poses non-trivial challenges. This paper describes a system of batched speculative decoding that sets a new state of the art in multi-sequence generation latency and that demonstrates superior GPU utilization as well as quality of generations within a time budget. For example, for a 7.8B-size model on a single A100 GPU and with a batch size of 8, each sequence is generated at an average speed of 5.8ms per token, the overall throughput being 1.1K tokens per second. These results represent state-of-the-art latency and a 2.15× speed-up over optimized regular decoding. Within a time budget that regular decoding does not finish, our system is able to generate sequences with HumanEval Pass@First of 43% and Pass@All of 61%, far exceeding what's feasible with single-sequence speculative decoding. Our peak GPU utilization during decoding reaches as high as 15.8%, more than 3× the highest of that of regular decoding and around 10× of single-sequence speculative decoding.

1 Introduction

In recent years, generative large language models (LLMs) have rapidly gained popularity due to their ability to generalize across a wide variety of tasks. These models are increasingly deployed commercially for applications such as coding assistants, writing aids, conversational agents, search and summarization tools and more. The accuracy

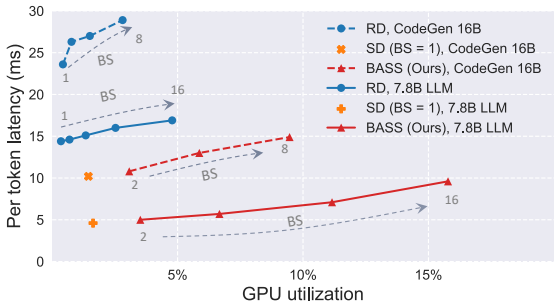

Figure 1: Comparing latency and GPU utilization of auto-regressive regular decoding (RD), single-sequence speculative decoding (SD) and our BASS method on two models. RD and BASS are measured with exponentially increasing batch sizes (BS).

performance of LLMs has been shown to scale with model size, with larger models demonstrating improved capabilities (Kaplan et al., 2020). However, this improvement comes at the cost of greater latency during inference and increased computational requirements.

Most popular LLMs are transformer-based decoder models. The inference speed of these models is often limited by memory bandwidth on typical hardware like GPUs. This is because GPUs tend to have much higher compute throughput relative to memory bandwidth. The auto-regressive decoding process of these models, where each output token is generated sequentially conditioned on previous tokens, means the entire model parameters need to be fetched from memory for each generated token. This sequential nature prevents parallelization during inference, resulting in under-utilization of available compute resources. For example, for both models in Figure 1, single-sequence regular decoding utilizes only 0.4% of GPU FLOPS.

To improve GPU utilization, batching multiple sequences is often employed to amortize the memory I/O costs across a batch and thereby utilize more FLOPS per memory I/O. However, large

[*]Equal contribution. [1]AWS AI Labs. [2]AWS NGDE. [3]Amazon AGI (work done at AWS). Correspondence to: Haifeng Qian <qianhf@amazon.com>, Sujan Kumar Gonugondla <gsujan@amazon.com>.

batch sizes are needed to effective utilize GPU compute, resulting in higher latency for individual sequences that are batched as well as bigger memory footprints. With larger model sizes, memory bottleneck becomes a challenge and limits allowable batch sizes. In Figure 1 for example, the highest GPU utilization by batched regular coding is only 4.8% before going out-of-memory.

Speculative decoding has emerged as an effective approach to improve latency of LLMs by increasing GPU utilization. The key idea is to draft a few tokens (typically by using a smaller LLM) and verify their correctness with the main LLM. By processing the draft tokens in parallel, speculative decoding amortizes the memory I/O of model parameters across the tokens. Despite its advantages, speculative decoding has limitations: It processes a single sequence at a time, restricting the parallelism to the number of draft tokens. This caps the potential GPU utilization improvements.

To address this, we present Batched Attention-optimized Speculative Sampling (BASS) – a parallel speculative decoder that handles multiple sequences simultaneously. BASS increases GPU utilization by parallelism across both the batch dimension and the draft-token dimension. We implement customized CUDA kernels to handle ragged tensors during attention calculation, which are a challenge posed by batched speculative decoding, and design a heuristic to dynamically adjust draft length for each step. As illustrated in Figure 1, BASS achieves latency and GPU utilization that are substantially improved from prior regular and speculative decoders. By comprehensive experiments on three different models including CodeGen and OPT, we study these trends as well as accuracy benefits of BASS. We study the impact of draft model design on overall system performance as well as that of algorithmic choices in attention kernels and draft lengths. BASS is applicable to both batch generation from a same prompt and batch generation from a set of different prompts.

2 Background

2.1 Inference with LLMs

This paper focuses on transformer-based (Vaswani et al., 2017) decoder-only generative LLMs. The standard inference of these LLMs can be divided into two phases: (a) *context encoding (prefill) phase* where the input prompt is processed in parallel to encode contextual information, and (b) *incremental decoding phase* where the model autoregressively generates output tokens one by one based on the encoded context (Figure 2(a)).

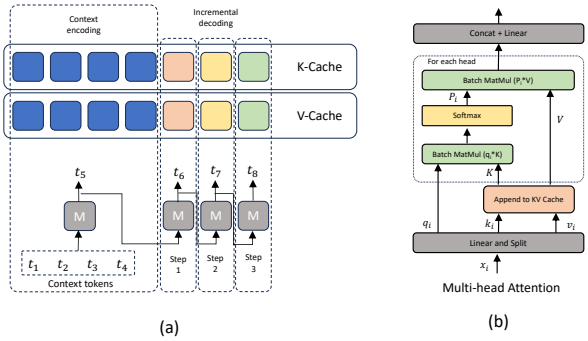

Figure 2: (a) Inference steps in regular decoding of an LLM. (b) Operations in multi-head attention.

Consider a decoder-only transformer (Radford et al., 2019) architecture with alternatively stacked feed-forward network layers and attention layers. In the attention mechanism (Figure 2(b)), key (k_i), value (v_i), and query (q_i) vectors are first computed by projecting the token embeddings for each position i. The queries are then used to calculate the relevance of the current token with past positions, by estimating their correlation with the past keys.

During the context encoding phase, all prompt tokens are processed in parallel. The attention keys and values for all context tokens (K and V tensors) are cached during this phase. This phase exhibits high GPU utilization.

The incremental decoding phase is bottlenecked by memory I/O from repeated fetching of model parameters and the KV cache as each output token is decoded. This phase is typically the dominant portion of inference time and exhibits low GPU utilization. It is the focus of our optimizations.

2.2 Speculative decoding

Speculative decoding (Stern et al., 2018; Xia et al., 2022; Leviathan et al., 2023; Chen et al., 2023) is a popular technique to reduce latency of LLM inference. As illustrated in Figure 3, the idea is to use a small draft model to generate k draft tokens. They are then processed by the main model as incremental context encoding to decide which draft tokens to accept.

The number of draft tokens to accept is decided based on the probabilities of generating these tokens according to the main and draft models. In the case of rejection, a corrected token is sampled from the outputs of the main model. Overall, this

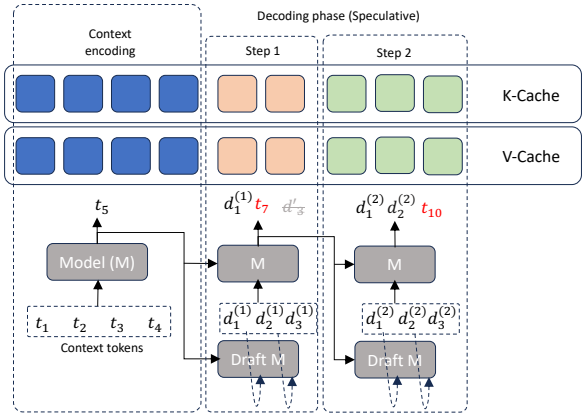

Figure 3: Standard speculative decoding. The draft model (Draft M) generates k draft tokens auto-regressively, which are then processed by the main model (M) in parallel to verify correctness.

decision process is stochastic and is designed to be equivalent to sampling from the main model's distribution (Leviathan et al., 2023; Chen et al., 2023). For example, we may accept the first five tokens, correct the sixth token, and then generate new draft tokens from the seventh position.

Latency benefit comes from the higher GPU utilization associated with incremental context encoding rather than auto-regressive decoding. More than 2× latency reduction has been reported in literature (Leviathan et al., 2023; Chen et al., 2023).

2.2.1 Limitations of speculative decoding

A major limitation of speculative decoding is that batch size of the main model's generation is preferably just 1, which is the setting in most existing works. It is straightforward to see why. In a naive implementation with batch size more than 1, we stop accepting tokens at the first reject position in the batch and hence lose some latency benefit. For illustration, let's make a simplistic assumption that each token generated by the draft model has an independent chance p of getting accepted, then the number of output tokens per draft has a geometric distribution with an expected value of $1/(1-p)$. For example, if $p = 80\%$, then on average the decoding process moves forward by 5 tokens per draft. With a batch size of b, the probability of acceptance per position becomes p^b. For a batch size of five as an example, the probability of acceptance per position becomes 33% and on average the decoding process moves forward by merely 1.5 tokens per draft, and we have lost most, if not all, of the latency benefit.

To retain the latency benefit with $b > 1$, we need to accept variable numbers of draft tokens across a batch. This poses challenges that no prior work has addressed efficiently and the existing systems remain single-sequence inference.

3 Batched Attention-optimized Speculative Sampling

Batched Attention-optimized Speculative Sampling (BASS) extends speculative decoding by enabling batch processing across multiple sequences. While speculative sampling improves GPU utilization for a single sequence, parallelism is limited to the small number of draft tokens per sequence (typically 5-10). Batching sequences with speculative decoding can further maximize GPU usage. To fully realize these benefits, specialized tensor manipulations and CUDA kernels are required.

3.1 Challenges with batching

One challenge with batched speculative decoding stems from the uncertainty in the numbers of acceptable draft tokens for each sequence, which vary across the batch. LLM inference kernels are designed to handle regular shaped tensors, primarily driven by CUDA restrictions. If we enforce a uniform sequence length across the batch, that would result in less accepted draft tokens and in diminishing benefits as discussed in Section 2.2.1.

In order to maintain the performance gains of speculative sampling, we need to be able to accept variable numbers of tokens across the batch. This will result in variable sequence lengths across the batch and consequently ragged-shape K and V tensors. In particular, this will affect the computations in the attention layer where we may not be able to batch the operations into a single kernel. Handling this effectively needs careful design considerations.

Another challenge is the choice of draft lengths. Although prior experimental systems tend to use a fixed draft length, real-life deployment requires adaptive draft lengths because different prompts may lead to different degrees of alignment between draft and main models during generation and hence different optimal draft lengths. For efficient incremental context encoding by the main model, we need to choose a uniform draft length across the batch at each step, and this decision needs to balance the needs of the multiple sequences.

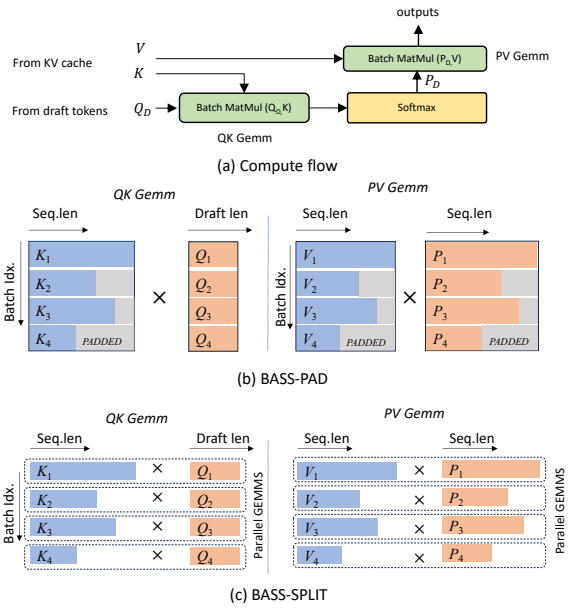

Figure 4: Attention calculation in BASS: (a) Attention compute flow, (b) BASS-PAD launches one kernel for QK GEMM and one kernel for PV GEMM by padding the K, V and P tensors to the maximum sequence length across the batch, and (c) BASS-SPLIT launches one kernel per sequence and thereby accommodates variable sequence lengths.

3.2 Proposed method

The attention computation flow, illustrated in Figure 4(a), involves a GEMM[1] operation between the query tensor Q of the current tokens and the key tensor K of all tokens processed so far, a softmax operation that outputs $P = \text{softmax}(Q^T K/c)$, and another GEMM operation between P and value tensor V of all tokens processed so far. There are three ragged tensors, K, V and P. We propose two variants: BASS-PAD and BASS-SPLIT as illustrated in Figure 4(b) and Figure 4(c) respectively. They implement the two GEMMs differently and share the same softmax operation: we simply launch separate softmax kernels, one for each sequence.

In the BASS-PAD approach, we pad the K, V and P tensors along the sequence length dimension to match the longest sequence of the batch and use the padded tensors for computations. We assign zero probabilities for the padded tokens in P. BASS-PAD does not incur additional cost of launching kernels, but we waste some compute to perform dummy computations with the padded elements.

[1]General matrix multiplication (GEMM).

The BASS-SPLIT approach is derived from the insight that attention operation is not associated with any model parameters and therefore applying batching to attention has no benefit of reducing the amount of GPU memory reads. In this approach, we break up the KQ GEMM and the PV GEMM into smaller per-sequence kernels, so that we can handle the variable sequence-length dimension in K, V and P tensors, as shown in Figure 4(c). Note that these separate kernels can be launched and executed in parallel. Since there is no sharing of memory reads among these separate kernels, the only extra cost that we pay is the cost of launching them. BASS-SPLIT does not waste any compute.

BASS-PAD and BASS-SPLIT apply to both the main and the draft models and apply to both token generation and incremental context encoding. With either approach, we can now let each sequence proceed at its own pace according to its own reject points and let sequences in the batch have different lengths.

Note that other steps, including the feed-forward network, the KQV projection layer, and the projection layer at end of attention, all remain the same and remain regularly batched to share memory reads of model parameters.

The comparison between BASS-PAD and BASS-SPLIT depends on the application scenario. We find BASS-PAD more attractive when the sequence lengths are closer to each other and BASS-SPLIT more favorable when the sequence lengths across batch vary by a large margin. Most results in Section 4 are based on BASS-PAD and their comparison results are in Section 4.6.

Algorithm 1 A heuristic to adjust draft length

$l_{\text{draft}} \leftarrow l_0$
$s \leftarrow 0$
for each speculative decoding step **do**
  $x_1, \cdots, x_b \leftarrow$ numbers of accepted tokens
  if $\max(x_1, \cdots, x_b) = l_{\text{draft}}$ **then**
    $l_{\text{draft}} \leftarrow \min(l_{\text{draft}} + l_{\text{incre}}, l_{\text{limit}})$
    $s \leftarrow 0$
  else
    $l_{\text{draft}} \leftarrow l_{\text{draft}} - \lceil l_{\text{draft}}/l_{\text{mod}} \rceil - s$
    $l_{\text{draft}} \leftarrow \max(1, x_1, \cdots, x_b, l_{\text{draft}})$
    $s \leftarrow 1$
  end if
end for

Algorithm 1 describes the heuristic that we use to dynamically adjust draft lengths. The rationale is

to increase draft length when at least one sequence has accepted all draft tokens in the last speculative decoding step and to decrease it otherwise. The speed of decrease is larger when the current draft length is larger and when it is a consecutive step of decrease. However, the next draft length is never decreased to be less than the max number of accepted tokens in the batch. We empirically chose the parameters of $l_0 = 7$, $l_{\text{incre}} = 2$, $l_{\text{mod}} = 10$ and $l_{\text{limit}} = 32$. Comparisons against constant draft lengths are in Section 4.6.

The degree of alignment between draft and main models varies across prompts, across different sequences from the same prompt, and also within the same sequence. When generating commonly used sentences or boilerplate code for example, the alignment tends to be strong and the optimal draft length is long. When generating uncommon sentences or novel code segments, the alignment tends to be weak and the optimal draft length is short. The effect of Algorithm 1 versus a fixed draft length is to dynamically get closer to the optimal draft length in both scenarios, so that the system generates longer drafts where possible yet does not waste compute to generate throw-away tokens.

4 Experiments

In this section we demonstrate the benefits of BASS over batched auto-regressive regular decoding (RD) and single-sequence speculative decoding.

4.1 Setup

Inference setup and CUDA kernels: All experiments throughout this paper are conducted on a single A100 GPU with 40GB memory. All inference runs, except for the "vLLM" rows in Tables 1 and 2, use a modified version of DeepSpeed[2] (DS), including both regular decoding and BASS runs. As can be seen in the tables, the regular decoding latencies are at the state of the art and comparable to those reported in (Aminabadi et al., 2022; Yao et al., 2022) for both FP16 and INT8. The vLLM runs use the latest vLLM version[3] (v0.3.0) and all sequences start immediately and hence result in the best possible latencies. Our modifications to DeepSpeed include:

- Kernels for quantizing both weights and activations to INT8 for all linear layers. We use CUTLASS[4] INT8→INT32 kernels for GEMM calls and modify them and other layers to fuse quantization and de-quantization operators.
- Kernels for attention calculations to address the ragged-tensor challenge in batched speculative decoding without sacrificing latency, as discussed in Section 3.

Models and tasks: We report experimental results on three main models with their respective draft models and tasks[5]:

- OPT 13B as the main model and OPT 125M or 350M as the draft model (Zhang et al., 2022). Following the same experimental setting as (Chen et al., 2023), we use the XSum task with 11,334 test examples (Narayan et al., 2018), use 1-shot prompting, generate 128 tokens per sequence, and use ROUGE-2 (Lin, 2004) as the metric to verify accuracy.
- CodeGen-Mono 16B as the main model and CodeGen-Mono 350M as the draft model. We use the HumanEval task with 164 examples and use the Pass@K accuracy metric (Chen et al., 2021), and we generate 256 tokens for each sequence.
- A 7.8B model trained on text and code as the main model and one of three draft models with sizes 310M, 510M and 1B. We again use the HumanEval task.

Latency metrics: We use the metric of per-token latency for each generated sequence [6]. For regular decoding, this value is the same across a batch. For speculative decoding, it varies within a batch and therefore we report three metrics: per-token latency of the first finished sequence in a batch, per-token latency of the last finished sequence in a batch, and per-token latency averaged across a batch, where each of the three is then averaged across examples in a task dataset.

[2] https://github.com/microsoft/DeepSpeed
[3] https://github.com/vllm-project/vllm
[4] https://github.com/NVIDIA/cutlass
[5] While the three tasks are the scenario of batch generation from a same prompt, please note that BASS is also applicable to batch generation from a set of different prompts.
[6] It is important to note that we do not divide latency by batch size, which was done in some papers, e.g., (Su et al., 2023). Fundamentally, our definition is a latency metric while the definition in (Su et al., 2023) is a throughput metric. With our definition, per token latency increases as the batch size increases because the amount of FLOPS during the per token latency is multiplied by batch size, and this applies to both regular decoding and speculative decoding.

Prec.	Batch	Method	ROUGE-2	Mean per-token latency & Speedup					
				First		Last		All	
FP16	1	RD (DS)	0.086	23.4 ms	1×	23.4 ms	1×	23.4 ms	1×
		RD (vLLM)	0.083	24.0 ms	0.98×	24.0 ms	0.98×	24.0 ms	0.98×
		BASS (ours)	0.084	10.8 ms	2.16×	10.8 ms	2.16×	10.8 ms	2.16×
	2	RD (DS)	0.085	25.9 ms	1×	25.9 ms	1×	25.9 ms	1×
		RD (vLLM)	0.084	23.9 ms	1.08×	23.9 ms	1.08×	23.9 ms	1.08×
		BASS (ours)	0.084	9.4 ms	2.74×	12.6 ms	2.05×	11.0 ms	2.34×
	4	RD (DS)	0.085	27.0 ms	1×	27.0 ms	1×	27.0 ms	1×
		RD (vLLM)	0.084	24.3 ms	1.11×	24.3 ms	1.11×	24.3 ms	1.11×
		BASS (ours)	0.084	9.6 ms	2.81×	16.6 ms	1.62×	12.7 ms	2.13×
INT8	1	RD (DS)	0.085	17.4 ms	1×	17.4 ms	1×	17.4 ms	1×
		BASS (ours)	0.087	8.5 ms	2.05×	8.5 ms	2.05×	8.5 ms	2.05×
	2	RD (DS)	0.086	20.1 ms	1×	20.1 ms	1×	20.1 ms	1×
		BASS (ours)	0.087	7.8 ms	2.57×	10.7 ms	1.87×	9.3 ms	2.16×
	4	RD (DS)	0.086	21.1 ms	1×	21.1 ms	1×	21.1 ms	1×
		BASS (ours)	0.087	8.2 ms	2.58×	14.8 ms	1.43×	11.2 ms	1.88×
	8	RD (DS)	0.086	23.5 ms	1×	23.5 ms	1×	23.5 ms	1×
		BASS (ours)	0.087	9.6 ms	2.44×	21.7 ms	1.08×	14.5 ms	1.62×

Table 1: OPT 13B accuracy and latency on XSum with auto-regressive regular decoding (RD) with DeepSpeed (DS) and vLLM, and BASS. Temperature is 0.2, nucleus top p is 0.95, and draft model is OPT 125M.

4.2 Performance on summarization

Table 1 shows the accuracy and latency results of the OPT 13B model on the summarization task of the XSum dataset, with OPT 125M as the draft model. As expected, the results suggest neutral accuracy between regular decoding and speculative decoding, while speculative decoding provides up to 2.81× speed up for finishing the first sequence and up to 2.34× speed up on average for all sequences. In a real-life application, and particularly in the scenario of generating multiple sequences for the same prompt, we can respond to the user as soon as the first sequence finishes while the other additional recommendations continue to generate. Therefore, speeding up the first sequence by 2.05×–2.81× implies a significant improvement in user-perceived latency.

The latency divergence between the first and last finished sequences increases with batch size. However, the last sequence latency is not as important in a real-life application because, when generating multiple sequences for the same prompt, we can simply choose a cut-off latency limit and return, e.g., five finished sequences out of a batch of eight.

4.3 Performance on code generation

Table 2 shows the accuracy and latency results of the CodeGen-Mono 16B model on the HumanEval task, with CodeGen-Mono 350M as the draft model. The trends are similar to Table 1: accuracy is neutral between regular decoding and speculative decoding; the first finished sequence is sped up by up to 2.65×, representing a significant improvement in user-perceived latency; the average latency of all sequences is reduced by up to 2.43×; the latency divergence between the first and last finished sequences increases with batch size.

Unlike Table 1, the accuracy metric in Table 2 increases with batch size: it is the percentage of examples where at least one correct generation exists in the batch. It represents an accuracy benefit of batched over single-sequence speculative decoding and more results will be presented in Section 4.5.

Overall the speed-up ratios in Table 2 are less than those in Table 1, and we hypothesize that the main factor is the larger size of the draft model.

Table 3 shows the accuracy and latency results of a custom 7.8B model, which was trained on text and code, on the HumanEval task, with a 310M-size draft model which is the first in Table 4. The overall trends are similar to those in Table 2 except that the speed-up ratios are substantially higher: the first finished sequence is sped up by up to 3.23×, and the average latency of all sequences is reduced by up to 2.94×. We hypothesize that the draft model architecture choice is the main reason and we will look at impact of draft model designs next.

4.4 Impact of draft model choices

Table 4 compares three draft models, all GPT2-like models with different architecture parameters as listed in the first three rows. They are trained with the same data and for the same amount of tokens. According to the fifth row, i.e., their stand-alone accuracy performance on HumanEval, the second draft model is more performant, likely due to its greater depth. This is also supported by the

Prec.	Batch	Method	Pass@Batch	Mean per-token latency & Speedup					
				First		Last		All	
FP16	1	RD (DS)	30.5%	23.6 ms	1×	23.6 ms	1×	23.6 ms	1×
		RD (vLLM)	31.0%	26.7 ms	0.88×	26.7 ms	0.88×	26.7 ms	0.88×
		BASS (ours)	30.5%	10.2 ms	2.31×	10.2 ms	2.31×	10.2 ms	2.31×
	2	RD (DS)	36.6%	26.3 ms	1×	26.3 ms	1×	26.3 ms	1×
		RD (vLLM)	35.9%	28.2 ms	0.93×	28.2 ms	0.93×	28.2 ms	0.93×
		BASS (ours)	36.0%	9.9 ms	2.65×	11.7 ms	2.25×	10.8 ms	2.43×
	4	RD (DS)	39.0%	27.0 ms	1×	27.0 ms	1×	27.0 ms	1×
		RD (vLLM)	40.4%	28.9 ms	0.93×	28.9 ms	0.93×	28.9 ms	0.93×
		BASS (ours)	40.2%	10.8 ms	2.50×	15.6 ms	1.73×	13.0 ms	2.07×
	8	RD (DS)	42.7%	28.9 ms	1×	28.9 ms	1×	28.9 ms	1×
		RD (vLLM)	45.1%	29.7 ms	0.97×	29.7 ms	0.97×	29.7 ms	0.97×
		BASS (ours)	45.1%	11.5 ms	2.51×	19.4 ms	1.49×	14.9 ms	1.94×
INT8	1	RD (DS)	32.3%	16.8 ms	1×	16.8 ms	1×	16.8 ms	1×
		BASS (ours)	31.7%	9.3 ms	1.82×	9.3 ms	1.82×	9.3 ms	1.82×
	2	RD (DS)	36.6%	19.6 ms	1×	19.6 ms	1×	19.6 ms	1×
		BASS (ours)	36.0%	9.3 ms	2.11×	10.9 ms	1.79×	10.1 ms	1.94×
	4	RD (DS)	38.4%	20.4 ms	1×	20.4 ms	1×	20.4 ms	1×
		BASS (ours)	39.0%	9.8 ms	2.07×	13.2 ms	1.54×	11.2 ms	1.81×
	8	RD (DS)	44.5%	21.9 ms	1×	21.9 ms	1×	21.9 ms	1×
		BASS (ours)	42.7%	11.1 ms	1.98×	18.8 ms	1.17×	14.3 ms	1.53×

Table 2: CodeGen-Mono 16B accuracy and latency on HumanEval with auto-regressive regular decoding (RD) with DeepSpeed (DS) and vLLM, and BASS. Temperature is 0.2, nucleus top p is 0.95, and draft model is CodeGen-Mono 350M.

Prec.	Batch	Method	Pass@Batch	Mean per-token latency & Speedup					
				First		Last		All	
BF16	1	RD (DS)	36.6%	14.4 ms	1×	14.4 ms	1×	14.4 ms	1×
		BASS (ours)	34.1%	4.6 ms	3.10×	4.6 ms	3.10×	4.6 ms	3.10×
	2	RD (DS)	45.7%	14.6 ms	1×	14.6 ms	1×	14.6 ms	1×
		BASS (ours)	45.1%	4.6 ms	3.16×	5.3 ms	2.74×	5.0 ms	2.94×
	4	RD (DS)	48.8%	15.1 ms	1×	15.1 ms	1×	15.1 ms	1×
		BASS (ours)	51.8%	4.7 ms	3.23×	7.0 ms	2.17×	5.7 ms	2.64×
	8	RD (DS)	55.5%	16.0 ms	1×	16.0 ms	1×	16.0 ms	1×
		BASS (ours)	53.7%	5.5 ms	2.92×	9.1 ms	1.75×	7.1 ms	2.25×
	16	RD (DS)	59.1%	16.9 ms	1×	16.9 ms	1×	16.9 ms	1×
		BASS (ours)	57.9%	7.3 ms	2.31×	13.0 ms	1.31×	9.6 ms	1.77×
INT8	1	RD (DS)	34.8%	11.0 ms	1×	11.0 ms	1×	11.0 ms	1×
		BASS (ours)	36.6%	3.7 ms	2.99×	3.7 ms	2.99×	3.7 ms	2.99×
	2	RD (DS)	40.9%	11.3 ms	1×	11.3 ms	1×	11.3 ms	1×
		BASS (ours)	43.3%	3.7 ms	3.03×	4.4 ms	2.59×	4.1 ms	2.79×
	4	RD (DS)	46.3%	11.8 ms	1×	11.8 ms	1×	11.8 ms	1×
		BASS (ours)	47.6%	4.1 ms	2.84×	5.7 ms	2.07×	4.8 ms	2.44×
	8	RD (DS)	51.2%	12.3 ms	1×	12.3 ms	1×	12.3 ms	1×
		BASS (ours)	55.5%	4.5 ms	2.73×	7.5 ms	1.66×	5.8 ms	2.15×
	16	RD (DS)	57.3%	13.6 ms	1×	13.6 ms	1×	13.6 ms	1×
		BASS (ours)	57.3%	6.3 ms	2.16×	10.6 ms	1.29×	8.0 ms	1.70×

Table 3: A 7.8B code model's accuracy and latency on HumanEval with regular decoding (RD) with DeepSpeed (DS) and vLLM, and BASS. Temperature is 0.2, nucleus top p is 0.95, and draft model is the first in Table 4.

sixth row which shows the chance of a draft token getting accepted during speculative decoding, and indeed the second draft model aligns better with the main model. However, because the second draft model itself takes higher latency to generate draft tokens, the overall latency of speculative decoding is increased despite accepting more draft tokens.

Table 5 is a similar comparison between two OPT draft models. Surprisingly OPT 350M is worse than OPT 125M in both stand-alone ROUGE-2 score on XSum and token acceptance rate which represents worse alignment with the main model of OPT 13B.

4.5 Benefits of batched speculative decoding

Figure 5 emulates a real-life application scenario where a service returns code recommendations to a user within a time budget. One of the recommendations is first displayed and the user has the option to flip through others. Ranking (here simply mean-logP based) is applied to pick the first displayed one. The Pass@First metric is the probabil-

draft model		A	B	C
#layer		4	8	4
#head		16	16	32
hidden dimension		2048	2048	4096
#param		310M	510M	1B
HumanEval pass@1		5.1%	11.4%	5.8%
token acceptance rate		87.4%	88.5%	87.2%
draft PTL (ms)	batch size 1	1.9	2.6	2.5
	batch size 2	2.0	2.6	2.6
	batch size 4	2.0	2.7	2.7
	batch size 8	2.0	2.7	2.8
	batch size 16	2.1	3.0	3.1
1st Seq PTL (ms)	batch size 1	3.7	4.5	4.5
	batch size 2	3.7	4.4	4.8
	batch size 4	4.1	5.0	5.2
	batch size 8	4.5	5.9	6.0
	batch size 16	6.3	7.6	7.7

Table 4: Comparisons between three draft models. *PTL* stands for per-token latency, and *1st Seq PTL* stands for that of the first finished sequence with BASS.

draft model		A	B
#layer		12	24
#head		12	16
hidden dimension		768	1024
#param		125M	350M
XSum ROUGE-2		0.023	0.015
token acceptance rate		78.5%	76.3%
draft PTL (ms)	batch size 1	3.1	6.9
	batch size 2	5.0	8.6
	batch size 4	5.0	8.5
	batch size 8	5.1	8.9
1st Seq PTL (ms)	batch size 1	8.5	14.2
	batch size 2	7.8	14.7
	batch size 4	8.2	15.7
	batch size 8	9.6	16.6

Table 5: Comparisons between two OPT draft models. *PTL* stands for per-token latency, and *1st Seq PTL* stands for that of the first finished sequence with BASS.

ity that the first displayed recommendation solves the problem correctly. The Pass@Finished metric is the probability that at least one of the finished recommendations within the time budget solves the problem correctly. These two metrics together quantify the accuracy quality of the service.

Figure 5 uses an end-to-end time budget of 2.5 seconds to generate 256-long sequences for any given prompt from HumanEval. According to Table 3, regular decoding under any setting would be unable to finish before time runs out, while single-sequence speculative decoding is able to return one recommendation and its Pass@First and Pass@Finished are the same and correspond to the left end point of the curves. With BASS and as the batch size increases, Pass@Finished is increased up to 61% and Pass@First is increased up to 43% with a simple ranking strategy using model confidence of mean-logP value. Both numbers are substantially higher than the mid-thirties accuracy by single-sequence speculative decoding. A real-life application would use a domain-specific stopping criteria instead of a fixed length and a more sophisticated ranking method, but the relative comparisons among the competing methods are as captured by Figure 5 and BASS is clearly superior.

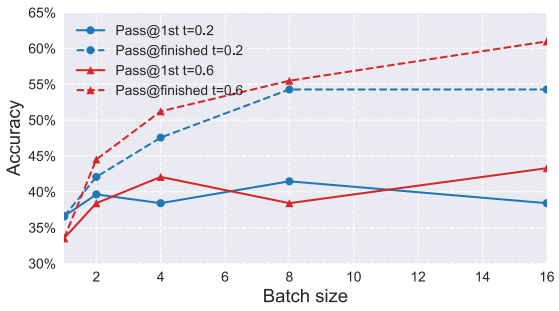

Figure 5: A 7.8B code model's accuracy on HumanEval with BASS, within a time budget of 2.5 seconds. t is temperature.

4.6 Ablation studies

In Table 6, we compare latency when certain alternative implementation choices are used. With BASS-SPLIT, we launch per-sequence kernels to handle ragged tensors as illustrated in Figure 4(c). The results suggest that the cost of launching more CUDA kernels in BASS-SPLIT out-weights the cost of wasted compute in BASS-PAD. Note that this relation is task dependent and may change when the sequence lengths across a batch vary by a large margin. For example, when applied on tasks of batch generation from a set of different prompts, the advantages of BASS-SPLIT could out-weights the cost. With "fixed draft size", we use a constant draft length instead of Algorithm 1 that dynamically modifies draft length. The results suggest that both the efficient attention calculation and the draft-length heuristic are important to the performance of BASS.

5 Related Work

Efficient inference of LLMs has been a popular research topic in recent years. Model quantization techniques (Yao et al., 2022; Lin et al., 2023; Frantar et al., 2022; Kuzmin et al., 2022) employ lower-precision representations for model parameters (e.g., INT8, INT4, FP8) without significantly compromising accuracy. Pruning (Frantar and Alistarh, 2023) reduces memory footprints via sparsity.

OPT 13B, XSum	1st Seq PTL (ms)		
batch	2	4	8
BASS	**7.8**	**8.2**	**9.6**
BASS-SPLIT	8.6	9.2	11.3
fixed draft size 4	8.9	9.6	11.3
fixed draft size 6	8.9	9.0	10.2
fixed draft size 8	8.9	9.1	10.3
CG 16B, HumanEval	1st Seq PTL (ms)		
batch	2	4	8
BASS	9.3	9.8	**11.1**
BASS-SPLIT	10.1	11.7	12.7
fixed draft size 4	9.7	10.2	12.3
fixed draft size 6	**9.1**	9.7	11.6
fixed draft size 8	9.7	**9.5**	13.1
Code 7.8B, HumanEval	1st Seq PTL (ms)		
batch	2	4	8
BASS	**3.7**	**4.1**	**4.5**
BASS-SPLIT	4.0	4.4	5.2
fixed draft size 4	4.6	5.1	6.3
fixed draft size 6	4.3	4.9	5.8
fixed draft size 8	4.0	4.3	5.1

Table 6: Ablation studies on latency impact of implementation choices. *1st Seq PTL* is per-token latency of the first finished sequence. BASS is the default setting used in all other tables.

Sparse attention techniques (Beltagy et al., 2020; Child et al., 2019) limit the number of tokens to attend to in order to reduce the complexity of attention layers, and thereby extend the maximum allowable sequence length.

Since its introduction, speculative decoding has seen numerous variations and improvements. Some proposals take a draft-model-free approach, by using an n-gram model to predict draft tokens (Fu et al., 2023), or by using the model embedding to predict drafts (Cai et al., 2024; Li et al., 2024). SpecInfer (Miao et al., 2023) uses a *draft tree* to generate and organize multiple drafts for the main-model sequence in order to maximize the number of tokens accepted per step. Su et al. (2023) study the relation between batch size and the optimal fixed draft length for max throughput; it is however based on a primitive prototype implementation: rejected draft tokens are masked rather than discarded, which achieve sequence lengths that are uniform across a batch yet are unnecessarily large and inefficient. The above works on speculative decoding are orthogonal to the discussions and ideas in this paper and can be combined. The conclusion presented in (Su et al., 2023) may change with the kernel implementations in this paper.

6 Conclusion

This paper presents Batched Attention-optimized Speculative Sampling (BASS), a system that advances the state of the art in fast multi-sequence generation by LLMs. By addressing the unique challenges of extending speculative decoding to batched inference without sacrificing latency, we demonstrate superior latency, GPU utilization as well as accuracy of generations within a time limit.

7 Limitations

This work, while advancing the state of the art, does not solve the efficient inference challenge of LLMs. For example, GPU utilization during the context encoding phase of LLM inference can be over 70% in our system, while the best achievable utilization during the incremental decoding phase is 15.8% in this paper. Although this is already significantly better than previous works, there is clearly substantial room to innovate and improve.

8 Ethical Impact

This work aims to increase the efficiency of deploying LLMs by utilizing the compute resources efficiently. It can reduce carbon emissions associated with LLM deployment. Additionally, driving down infrastructure costs can potentially encourage broader LLM adoption. Impact of increased LLM usage on the society and associated risks are difficult to forecast.

References

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE.

Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph,

Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *URL https://openai.com/blog/sparse-transformers*.

Elias Frantar and Dan Alistarh. 2023. Massive language models can be accurately pruned in one-shot. *arXiv preprint arXiv:2301.00774*.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate post-training compression for generative pretrained transformers. *arXiv preprint arXiv:2210.17323*.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2023. Breaking the sequential dependency of llm inference using lookahead decoding. *URL https://lmsys.org/blog/2023-11-21-lookahead-decoding*.

Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. 2019. A study of bfloat16 for deep learning training. *arXiv preprint arXiv:1905.12322*.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *CoRR*, abs/2001.08361.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Andrey Kuzmin, Mart Van Baalen, Yuwei Ren, Markus Nagel, Jorn Peters, and Tijmen Blankevoort. 2022. Fp8 quantization: The power of the exponent. *arXiv preprint arXiv:2208.09225*.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. 2023. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv*.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2023. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*.

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 preprint arXiv:1808.08745*.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. 2018. Blockwise parallel decoding for deep autoregressive models. *Advances in Neural Information Processing Systems*.

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*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*.

Heming Xia, Tao Ge, Si-Qing Chen, Furu Wei, and Zhifang Sui. 2022. Speculative decoding: Lossless speedup of autoregressive translation.

Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in Neural Information Processing Systems*, 35:27168–27183.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*.

A Appendix

A.1 Quantization schemes and mechanisms

This section describes the quantization schemes and kernels used for INT8 inference. Since granular assignment of precision improves the accuracy of quantized models, we assign the quantization ranges to the smallest granularity that allows us to compute the matrix multiplications in integer arithmetic, i.e., the granularity is set to the inner-product dimension. This translates to per-channel quantization for weights, dynamic per-token quantization

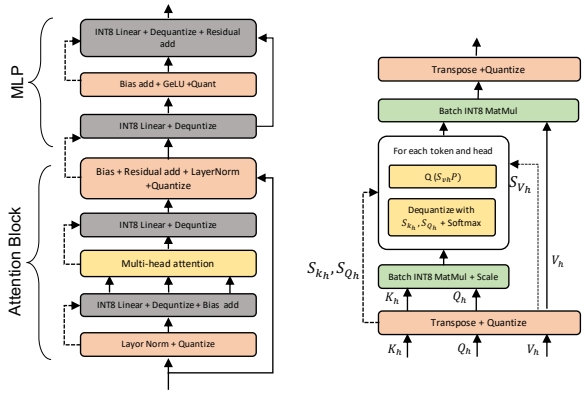

Figure 6: Inference data-flow with quantization

for activation, and dynamic per-head and per-token quantization for keys, queries, and values.

To mitigate the overheads associated with quantize and dequantize operations in the inference pipeline on a GPU, we have employed kernel fusion techniques as shown in Figure 6. This involves amalgamating multiple operations to minimize CUDA kernel calls and memory fetches, thereby minimizing computation time.

Dynamic quantization can incur substantial overheads unless it is integrated with other operations in the inference pipeline. We fuse the quantize operation with layer-norm, GeLU, and transpose operations across the network. This approach eliminates the need for redundant memory reads, for example, reading the same data for layer-norm and quantization separately.

We use CUTLASS INT8 GEMM kernels to fuse the dequantize operation and other element-wise computations, such as bias and residual additions with the GEMM operations. To optimize performance for small batch sizes, we adopt a strategy of pre-multiplying the weights and activation scales during the activation quantization operation, and subsequently retrieving them during the epilogue phase of the GEMM operations. The resulting fused GEMM blocks yield floating-point precision outputs, namely FP16, BF16 or FP32, depending on the selected format.

A.2 Draft model choice and training

For a draft model to be effective we need to maximize the number of tokens accepted by the main model while maintaining low per-token latency. Given the requirement, we have investigated architecture selection and observed that, at a fixed parameter count, having more attention heads (wider model) is better than having more layers with fewer attention heads (deeper model) since we have similar representation capabilities with wider-but-shallow models as narrow-but-deep models but at a much lower latency. We summarize the draft model architecture choices in Table 4.

We trained each of these draft models using the same data for the main model with a context size of 2048 and a global 512 batch size across 8 servers each with 8 40GB Nvidia A100 GPUs. This translates to approximately 1 million tokens per batch. Using a learning rate of 3.5×10^{-4} we train for 300k steps. We use AdamW optimizer (Kingma and Ba, 2014) with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and $\epsilon = 10^{-8}$. The warm-up steps were set to 2000, and a cosine annealing learning rate schedule was employed after reaching the peak learning rate. The minimum learning rate was set to 10% of the peak learning rate. We use BF16 (Kalamkar et al., 2019) precision and set gradient clipping to 1.0 to enhance training stability.