| INTRO |

이번에 톺아볼 논문은 바로 '**Hymba: A Hybrid-head Architecture for Small Language Models**'입니다. 간단하게 말하자면 이전에 Diff Transformer를 구현하는 과정에서 만들었던 Transformer(llama)와 Mamba를 병렬로 연결한 모델입니다. 이를 통해 Transformer와 Mamba의 장점을 섞으면서도 단점은 희석시키기 위해 고안된 모델이며, 추가적으로 '**KV-Cache Sharing**', '**Sliding Window Attention**', '**Meta Token**'이 적용된 모델입니다.

이번 포스팅에선 Hymba를 구성하는 고유 구성 요소들을 단계적으로 구현하고(llama block과 mamba는 이전에 구현했던 내용을 그대로 사용합니다. llama block에 대한 내용은 Differential Transformer에 대한 포스팅을, mamba block에 대한 내용은 Mamba에 대한 포스팅을 각각 참조하세요), 이전에 했던 것과 동일하게 작은 셰익스피어 희곡 데이터셋에 학습시켜 성능을 확인해보겠습니다.

본 논문[1]은 마치 이전의 ConvNeXt 논문 때와 같이 꽤나 친절하게 쓰여진 편에 속하는 논문인데요, 특히 아래와 같이 Hymba 구조를 step-by-step으로 구현하는 과정 및 그 과정으로 인해 얻을 수 있는 효과를 잘 정리해두었습니다.

Configuration	Commonsense Reasoning (%)	Recall (%)	Throughput (token/sec)	Cache Size (MB)	Design Reason
Ablations on 300M model size and 100B training tokens					
Transformer (Llama)	44.08	39.98	721.1	414.7	Accurate recall while inefficient
State Space Models (Mamba)	42.98	19.23	4720.8	1.9	Efficient while inaccurate recall
A. + Attention heads (sequential)	44.07	45.16	776.3	156.3	Enhance recall capabilities
B. + Multi-head structure (parallel)	45.19	49.90	876.7	148.2	Better balance of two modules
C. + Local / global attention	44.56	48.79	2399.7	41.2	Boost compute/cache efficiency
D. + KV cache sharing	45.16	48.04	2756.5	39.4	Cache efficiency
E. + Meta tokens	45.59	51.79	2695.8	40.0	Learned memory initialization
Scaling to 1.5B model size and 1.5T training tokens					
F. + Size / data	60.56	64.15	664.1	78.6	Further boost task performance
G. + Extended context length (2K→8K)	60.64	68.79	664.1	78.6	Improve multi-shot and recall tasks

Table 1 | Design roadmap of our Hymba model. We evaluate the models' (1) commonsense reasoning accuracy, averaged over 8 tasks, and (2) recall accuracy, averaged over 2 tasks, which corresponds to retrieving relevant information from past input. The throughput is on NVIDIA A100, sequence length 8k, batch size 128. The cache size is measured with a 8k sequence length, assuming the FP16 format.

Design Roadmap of Hymba (from [1])

뿐만 아니라 본 모델을 학습 시킬 때 적용한 Training Process도 아래와 같이 공개되어 있습니다. **(일반적으로 사용되는 방법론 - 대규모 비지도 학습 > 지도 학습 > 정제 지도 학습 > 강화 학습 - 이지만, 그 데이터의 규모나 특징이 잘 정리돼 있어 유용합니다)**

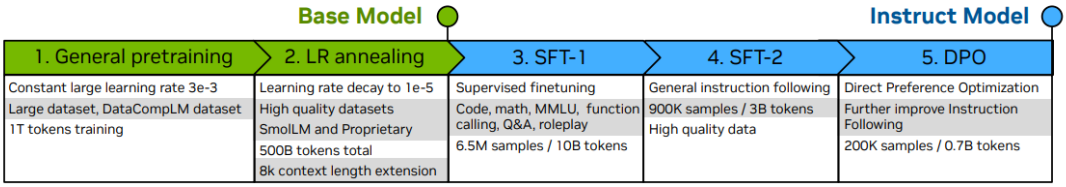


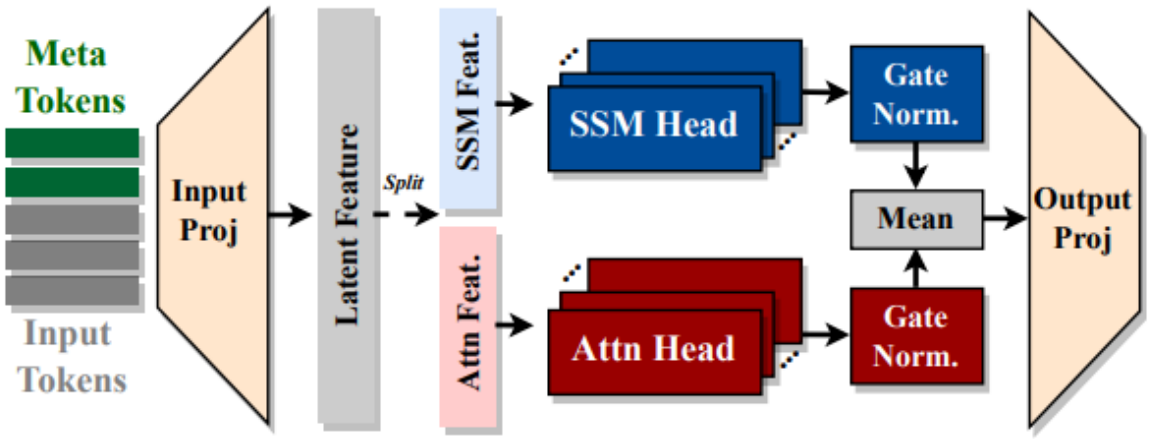
Figure 8 | Training pipeline adapted for Hymba family. For detailed loss curve of Hymba-Base-1.5B see Fig 14.

Training Process (from [1])

본 리뷰에서는 위와 같은 수준의 학습 단계를 직접 실행해보긴 현실적으로 어려우므로, 모델 구조를 온전히 구현하고, 각 구성 요소별로 어떤 효과를 얻을 수 있는지 확인하는 정도로 마치도록 하겠습니다.

| Hymba Structure |

우선 전체적인 구조를 살펴보고 가는 것이 좋습니다.



Hymba Structure (from [1])

가장 핵심은 당연히 위 그림에서도 볼 수 있듯 SSM과 Attention을 병렬로 연결해 Hybrid Head를 구성한다는 것입니다. 이를 통해 Mamba의 전체 시퀀스에 대한 흐름과 방향성을 빠르게 파악하는 능력과, Transformer의 느리지만 특정 정보에 대한 정밀한 작업 능력을 합쳐 성능은 향상시키면서도 실행 속도는 그 중간에 위치하게끔 한다는 전략적 구조인 셈입니다.

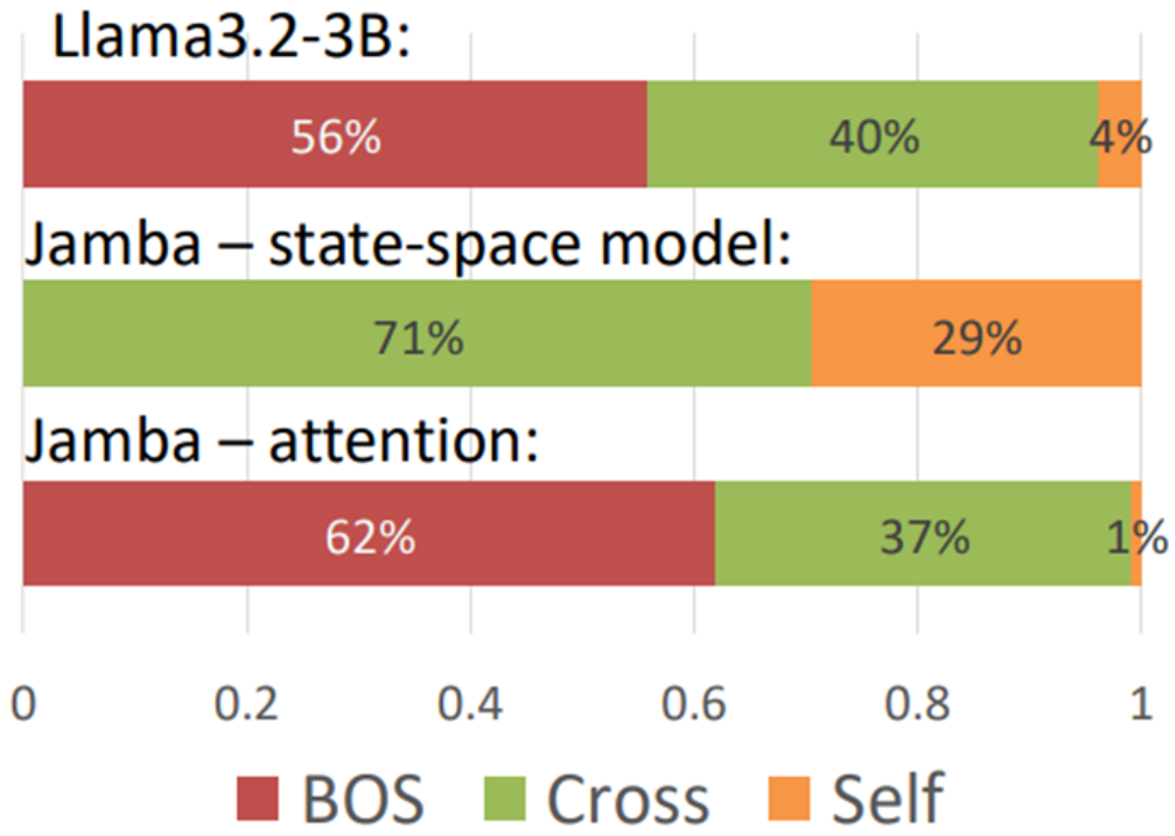
그리고 추가적으로 Transformer의 고질적인 문제를 해결하기 위해 크게 세 가지 대안이

추가되기도 하는데요, 각각 '**Meta Token**', ' **Attention Combination**', 그리고 '**KV Cache sharing**'입니다. 이를 단계적으로 설명한 후, 코드로 구현해 그 성능을 살펴보도록 하겠습니다.

| #1 Meta Token |

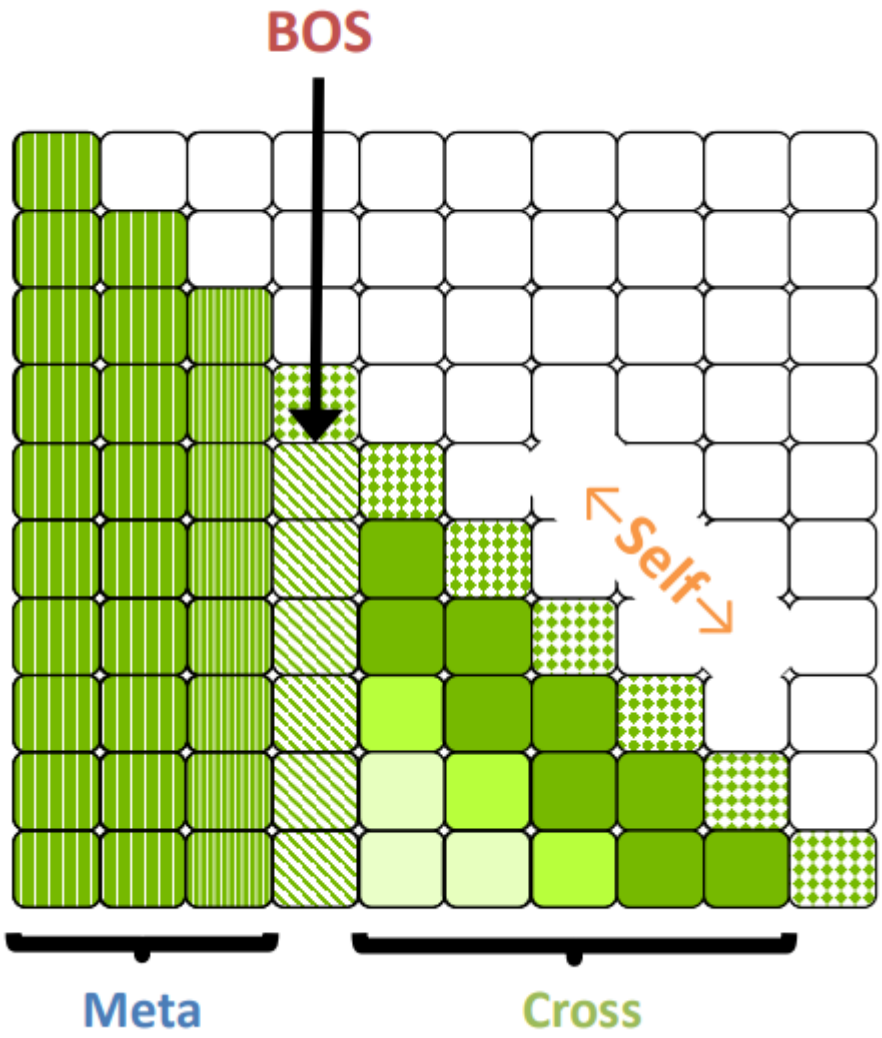
처음으로 살펴볼 것은 바로 Meta Token입니다. 최근의 여러 논문 및 연구들에서 Transformer 기반 모델들이 필요 이상으로 제일 앞의 토큰에 큰 attention score를 분배하는 문제를 확인했습니다. 어느 정도냐면, 일반적으로 문장의 앞에 추가되는 BOS 토큰('Begin of Sentence')에 과반의 attention score가 집중되는 문제가 있었습니다.

즉, 큰 의미가 없는 토큰에 attention score를 낭비하는 문제가 발생하고 있고 이는 모델의 성능이 온전하게 발휘되지 못하는 문제를 야기하는데요, 아래 그림에서 확인할 수 있듯 현대 Transformer 구조라 칭해지는 Llama에서는 56%가, 2024년 기준 가장 일반적으로 사용되는 Foundation Model Structure 중 하나인 Jamba(attention과 mamba를 직렬로 연결한 hybrid 구조)에서의 attention에서는 무려 62%의 attention score가 할당된 것을 아래 그림을 통해 확인할 수 있습니다.



Sum of attention score from different categories (from [1])

이전에 리뷰했던 Differential Transformer에서는 이러한 문제 - 의미없는 토큰에 attention score를 할당하는 Attention Noise 문제 - 를 해결하기 위해 Differential Attention이라는 차등주의 어텐션을 통해 무의미한 attention을 제거했는데요. Hymba에서는 조금 다르게 이 문제를 해결합니다. 바로 Meta Token이라는 전체 시퀀스에 영향을 주는 토큰을 임의로 제일 앞 시퀀스에 추가하는 것입니다. 이를 그림으로 나타내면 아래와 같습니다.



add meta token

(from [1])

위 그림에서 볼 수 있듯이, 세 개의 메타 토큰이 <BOS> 토큰 앞에 추가되었습니다. 즉, 제일 앞에 있다는 이유만으로 <BOS> 토큰에 집중되던 attention score를 메타 토큰에 집중시킬 수 있게 되었는데요. 이는 몇 가지 장점이 있습니다.

자연어 모델이라면 단순 언어능력뿐 아니라 이를 기반으로 하는 특정 도메인(ex. 법률, 의학, 수학, 과학, 코드 등) 별로 상이한 입출력 스타일 및 결과를 요구받게 됩니다. 이러한 문제는 단순히 자연어 모델뿐 아니라 컴퓨터 비전이나 시계열, 이상탐지와 같은 영역에도 해당할 수 있는 문제인데요, 이때 본 논문에서는 각기 다른 도메인에서 프람프트가 입력됐을 때 각기 다른 메타토큰들이 활성화되는 현상이 관찰되었으며 이는 메타토큰들이 각 도메인 별 정보 등을 캡슐화한 것으로 보인다라고 말합니다.

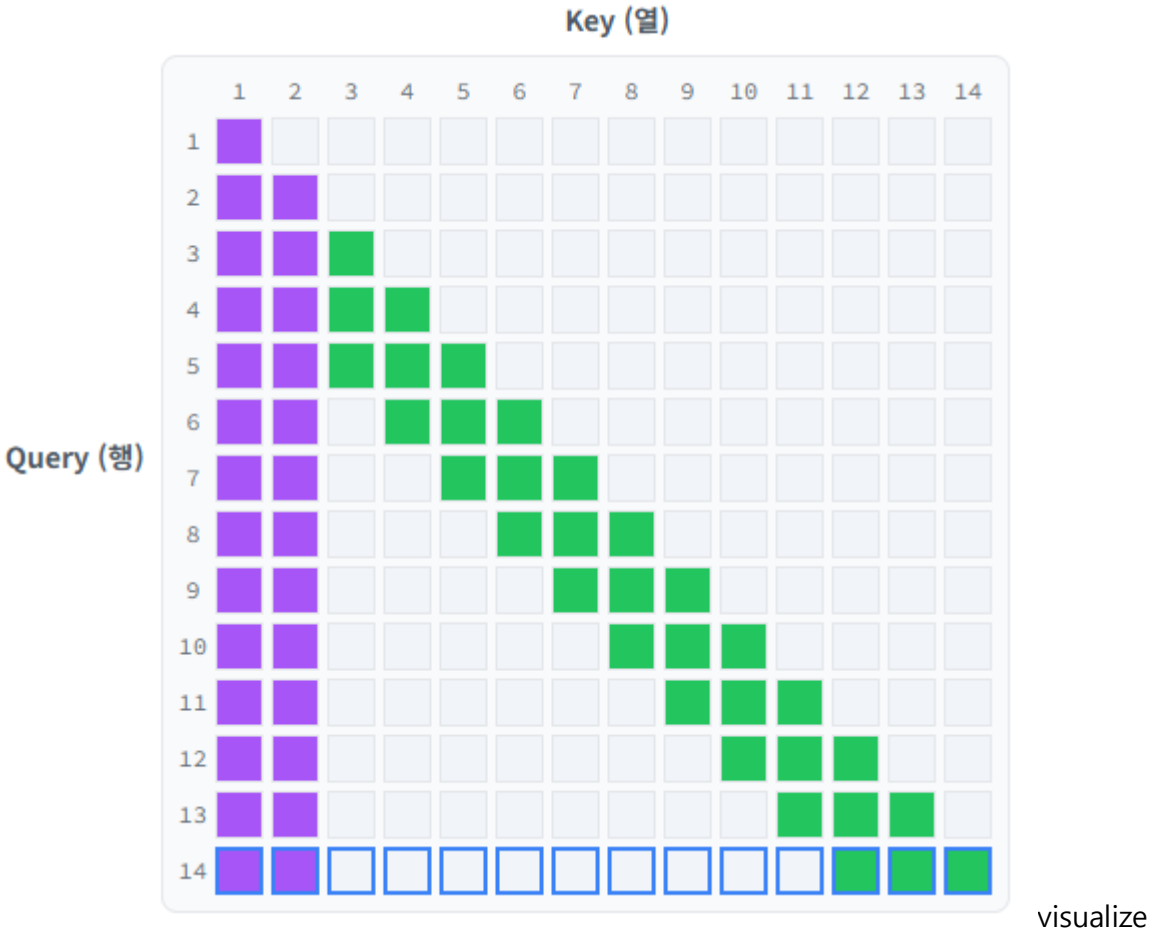
즉, 제일 앞에 있는 무의미한 토큰에 집중되던 attention score를 활용하기 위한 발상의 전환으로, 제일 앞에 의미있는 토큰을 두는 방식을 채택함으로써 단순히 일반적인 성능을 향상시킬뿐 아니라 여러 도메인에 걸쳐 특화될 수 있는 능력까지도 향상시킨 것입니다. 다만 이러한 전략적 선택은 한 가지 큰 문제를 발생시키는데요. 바로 시퀀스의 길이가 길어짐에 따라 가뜩이나 고질적인 문제로 지적받던 Transformer의 시퀀스 길이(n)에 quadratic하게 증가하던 연산 복잡도를 (n+len_meta_token)에 quadratic하게 증가시키는 문제를 발생시킨 것입니다.

때문에 다음의 두 가지 방법이 더 적용되었습니다.

| #2 Combination with Global Attention & Local Attention(SWA) |

첫 번째로 적용된 방식은 바로 기존의 Global Attention(시퀀스의 모든 토큰들과의 attention) 방식과 Local Attention(Sliding Window Attention, 특정 길이의 윈도우 길이 내의 토큰들과만 attention) 방식을 혼합해 사용하는 방식입니다.

이를 통해 attention의 대상이 되는 sequence 길이를 meta token의 길이와 sliding window의 크기를 더한 값 이하로 하도록 함으로써 meta token의 추가로 인한 연산량 증가를 줄이는 전략입니다. 이를 시각화하면 다음과 같습니다.



visualize

SWA w/ Meta Token (from author)

위 그림은 두 개의 메타토큰이 추가된, window size = 3의 Local Attention을 시각화한 그림입니다. 위 그림에서 알 수 있듯 모든 토큰에 대해서 보라색으로 표시된 메타토큰이 attention score 계산 대상임을 확인할 수 있으며, 현재 토큰 인덱스 이전의 것들 중 window size의 범위 내에서 attention이 이뤄지고 있는 것을 확인할 수 있습니다.

이를 통해 기존의 첫 번째 토큰이었을 idx=3 토큰은 메타토큰과 자기 자신까지 3개의 토큰에 대해 attention score를 계산하게 되며, idx=4 토큰은 메타토큰 2개와 자기 자신 및 그 이전 토큰의 2개를 합쳐 4개 토큰에 대해, 그 다음은 최대 윈도우 크기인 3이 적용돼어 5개 토큰에 대해 attention score를 계산하게 되는 식입니다.

당연히 이러한 attention 방식은 정보 손실을 야기할 수 있으며, 이로 인해 성능의 하락을 피할 수 없는데요. 때문에 본 논문[1]에서는 모든 레이어에 SWA를 쓰는 게 아니라, 제일 처음과 중간, 그리고 마지막에는 Global Attention을 적용해 정보 손실을 막거나 복원하도록 해 성능 하락은 최소화하거나 없애면서도 실행 속도는 빠르게 만들 수 있었다

고 합니다.

| #3 KV Cache Sharing |

다음으로는 KV Cache Sharing(정확히는 'Cross-layer KV Cache Sharing'이 본 논문에서의 명칭)입니다. 이는 간단히 말해 Local Attention이 적용되는 모든 레이어마다 별도의 KV Cache를 생성해 활용하는 기존의 방식 대신 두 개의 레이어마다 하나의 Cache를 활용하는 방법입니다.

이에 대한 근거는 인접한 레이어 간의 KV Cache가 거의 유사했다는 기존의 실험 결과에 기반합니다. 개인적으로 추측컨데, 이는 residual connection으로 인한 레이어 간 유사성이라는 특징으로 인한 것으로 보여집니다. 잔차 연결이라는 방법은 딥러닝 모델이 그 이름답게 모델이 깊고, 복잡해질수록 성능이 향상되는 것을 담보하는 역할을 했지만, 반대로 각 레이어들이 이전 레이어와 아주 미세한 차이만을 만들어내도록 학습되는 문제가 발생하기도 했습니다.

즉, 모델의 깊이와 크기에 비해 그 성능을 온전히 내고 있지 못한 것이기도 한 것입니다. 하지만 그렇다고 잔차 연결이라는 방식을 무턱대고 제거하기엔 작금의 딥러닝 모델 구조에서 오는 이점을 제대로 살리지 못하는 문제가 있어, 아직은 실험적으로 일부 잔차 연결을 제거하거나 학습 레시피 상의 후반 단계에서 일부 연결 제거, 띄엄띄엄 적용하는 등의 실험이 이뤄지고 있는듯 합니다.

여튼 다시 본래의 주제로 돌아와서 본 논문[1]에서는 이러한 점을 고려해 기존의 재귀적으로 수행되는 next token prediction의 과정에 필수적으로 자리잡은 KV Cache를 굳이 모든 레이어마다 저장하지 않고, 인접 레이어들끼리 공유하는 전략을 취함으로써 KV Cache를 만들고, 저장하는 과정을 최적화하였습니다. 이는 아래 표를 통해서 확인할 수 있습니다.

Configuration	Commonsense Reasoning (%)	Recall (%)	Throughput (token/sec)	Cache Size (MB)	Design Reason
Ablations on 300M model size and 100B training tokens					
Transformer (Llama)	44.08	39.98	721.1	414.7	Accurate recall while inefficient
State Space Models (Mamba)	42.98	19.23	4720.8	1.9	Efficient while inaccurate recall
A. + Attention heads (sequential)	44.07	45.16	776.3	156.3	Enhance recall capabilities
B. + Multi-head structure (parallel)	45.19	49.90	876.7	148.2	Better balance of two modules
C. + Local / global attention	44.56	48.79	2399.7	41.2	Boost compute/cache efficiency
D. + KV cache sharing	45.16	48.04	2756.5	39.4	Cache efficiency
E. + Meta tokens	45.59	51.79	2695.8	40.0	Learned memory initialization

Design roadmap of Hymba (from [1])

위 표에서 볼 수 있듯이 KV Cache Sharing을 적용함으로써 처리 속도 및 캐시 사이즈는 줄어든 반면, 성능 하락은 뚜렷하지 않습니다(정확히는 어느정도 trade off가 있는 것 같지만, meta token을 통해 무시할 수 있는 수준이 되는듯 합니다).