Fast DistilBERT on CPUs

Haihao Shen[*]
Intel Corporation
haihao.shen@intel.com

Ofir Zafrir[*]
Intel Labs
ofir.zafrir@intel.com

Bo Dong
Intel Corporation
bo1.dong@intel.com

Hengyu Meng
Intel Corporation
hengyu.meng@intel.com

Xinyu Ye
Intel Corporation
xinyu.ye@intel.com

Zhe Wang
Intel Corporation
zhe1.wang@intel.com

Yi Ding
Intel Corporation
yi1.ding@intel.com

Hanwen Chang
Intel Corporation
hanwen.chang@intel.com

Guy Boudoukh
Intel Labs
guy.boudoukh@intel.com

Moshe Wasserblat
Intel Labs
moshe.wasserblat@intel.com

Abstract

Transformer-based language models have become the standard approach to solving natural language processing tasks. However, industry adoption usually requires the maximum throughput to comply with certain latency constraints that prevents Transformer models from being used in production. To address this gap, model compression techniques such as quantization and pruning may be used to improve inference efficiency. However, these compression techniques require specialized software to apply and deploy at scale. In this work, we propose a new pipeline for creating and running Fast Transformer models on CPUs, utilizing hardware-aware pruning, knowledge distillation, quantization, and our own Transformer inference runtime engine with optimized kernels for sparse and quantized operators. We demonstrate the efficiency of our pipeline by creating a Fast DistilBERT model showing minimal accuracy loss on the question-answering SQuADv1.1 benchmark, and throughput results under typical production constraints and environments. Our results outperform existing state-of-the-art Neural Magic's DeepSparse runtime performance by up to 50% and up to 4.1x performance speedup over ONNX Runtime.

1 Introduction and related work

Large Transformer-based Language Models (LMs) are evolving rapidly from millions of parameters, e.g., BERT-Large [3], to billions of parameters, e.g., Turing-Megatron [14], and GPT3 [2]. Those large models have demonstrated promising state-of-the-art (SoTA) accuracy on a wide range of NLP tasks. However, those models are inefficient and therefore unsuited to the limited computational sources and strict latency constraints in production.

To enable and increase the efficiency and scale of deployed Transformer models, additional model compression and optimization are usually required along with a dedicated inference engine. For

[*]Equal contribution

example, DistilBERT [12], using knowledge distillation [4], shows minimal impact on downstream NLP tasks at a reduced size by 40%. Additionally, pruning [13] and quantization [17] are well-known techniques to further compress a Transformer model. Recent works have proposed pruning Transformer models at pre-training to create sparse pre-trained LMs. It has been established that fine-tuning these sparse pre-trained LMs to downstream tasks can produce results that are competitive with other pruning methods, obviating the need for pruning. [18, 7]. For example, a DistilBERT with 90% unstructured sparsity was produced with minimal impact on the accuracy of downstream tasks [18]; however, performance gains have not been demonstrated for this model. Similarly, Neural Magic published the throughput performance of a structured sparse DistilBERT with 80% sparsity, batch size 32 and sequence length 128, although latency performance was not provided [9]. Nvidia recently proposed novel 2:4 structured sparsity that is only available on Ampere architecture and above [10]. Quantization [6][15] has been maturing in industry with two well-known approaches: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). Typically, PTQ requires an offline calibration process on a representative calibration dataset, while QAT requires an additional fine-tuning stage simulating quantization inference while training. Although the approaches have proved successful in some typical models [16, 17], it remains challenging to produce a quantized model based on a highly sparse model without sacrificing accuracy.

In this paper, we propose a new pipeline for creating and running Fast Transformer models on CPUs. We extend the model compression approach PruneOFA [18] by enabling CPU-friendly block-wise structured sparsity. We then apply an accuracy-aware post-training quantization approach to generate an 8-bit (INT8) sparse model. Furthermore, we demonstrate our models using our Transformer inference engine dedicated for running sparse & quantized Transformers on CPUs. We apply this pipeline on DistilBERT [12] to create Fast DistilBERT. We show that Fast DistilBERT can meet a requirement of under 1% accuracy loss vs. the DistilBERT baseline on the question-answering benchmark SQuADv1.1 [11]. We demonstrate up to 1.5x performance gain over Neural Magic's DeepSparse engine [8] and up to 4.1x performance gain over ONNX Runtime on common CPUs from Amazon Web Services (AWS) under typical production constraints.

Our main contributions are threefold: 1) Propose a hardware-aware extreme compression technique for fast Transformer models on CPUs. 2) Create an efficient Transformer inference runtime for sparse & quantized Transformer models. 3) Demonstrate new SOTA performance under typical constraints in common production environments.

2 Method description

In this section, we describe how to solve the challenges of applying model compression techniques on Transformer-based LMs.

Block-wise structured sparsity In this work, we accelerate sparse Transformer-based models with specialized sparse GEMM operators which require a structured sparsity of constant size blocks in the output dimension, see Section 3. To that end, we extend the model compression infrastructure proposed by Zafrir et al. [18] to create sparse pre-trained LMs with block-wise structured sparsity. Further details on block-wise pruning in Appendix A.1.

Fine-tuning with knowledge distillation Knowledge distillation is a widely used method for model compression [4, 12]. Knowledge distillation while fine-tuning Transformer-based models may improve the accuracy of the model on the downstream task and bridge the accuracy gap caused by compression methods applied to the model [13, 18, 7]. Following these works we apply knowledge distillation while fine-tuning the block-wise sparse pre-trained LM to downstream tasks to mitigate the accuracy loss.

Post-training quantization PTQ is an effective approach to quantizing a model without additional training steps. It requires a calibration process using a representative dataset to determine the quantization parameters of the model. Converting FP32 operations to INT8 can increase their efficiency by up to 4x [1]. We apply PTQ with automatic accuracy-aware tuning on the sparse Transformer model to produce an optimized model using Intel® Neural Compressor (INC) open-source tool for quantization [5].

3 Software acceleration

We develop a Transformer inference engine on CPU with advanced runtime, graph optimization, and sparse GEMM operators.

Advanced runtime We develop an advanced, cache friendly, memory allocator to enable buffer reuse and to reduce memory allocation overhead, in particular for a continuous demand to allocate/free small memory chunks. Moreover, we implement weight sharing that allows a single copy of weight to be shared across multiple instances running in parallel during inference. More details in Appendix A.3.1,A.3.2.

Graph optimization The computation graph of a Transformer-based model contains many small and/or redundant operations. After obtaining our quantized Transformer-based model we apply several optimizations at the graph level including operations fusion, removal, etc. See Appendix A.3.3 for further details.

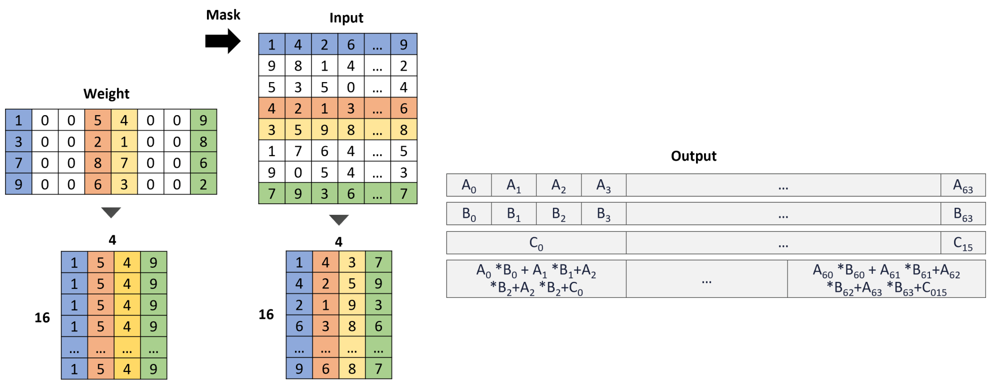
Mask
Input
Weight
Output
4
16
4
16
A_0 A_1 A_2 A_3 … A_{63}
B_0 B_1 B_2 B_3 … B_{63}
C_0 … C_{15}
$A_0 * B_0 + A_1 * B_1 + A_2 * B_2 + A_2 * B_2 + C_0$ … $A_{60} * B_{60} + A_{61} * B_{61} + A_{62} * B_{62} + A_{63} * B_{63} + C_{015}$

Figure 1: INT8 sparse GEMM kernel based on VNNI

Sparse GEMM operators We implement INT8 sparse GEMM kernel to accelerate operations between dense input and sparse weights leveraging AVX512-VNNI instructions. Figure 1 illustrates how the kernel operates. Given a block-wise sparse weight, we broadcast the non-zero weight block to form a VNNI-format block A. Based on the mask in the sparse weight, we re-organize the corresponding input as another VNNI-format block B. Then, the kernel uses VNNI to produce the intermediate output given A and B, and add bias C as the final output. Algorithm 1 in the appendix contains the implementation of the kernel.

4 Experimental setup

To demonstrate the performance of the runtime optimizations described in Section 3, we produce a block-wise sparse pre-trained DistilBERT, and then fine-tune and quantize it for the question-answering SQuADv1.1 benchmark [11] using the methods described in Section 2. Appendix A.2 details our model creation process. Table 1 lists the F1 scores of the DistilBERT baseline and our compressed model. To evaluate the performance of our optimizations, we evaluate our compressed model performance on a AWS c6i.12xlarge CPU instance for 3 scenarios: maximum ThroughPut (TP), minimum latency, and production. The production scenario requires maximum TP under 10 milliseconds (ms) latency per batch. Moreover, we compare our performance to other public inference runtimes for quantized and sparse models, ONNX Runtime and Neural Magic. We include the F1 results for their corresponding models in Table 1.

5 Results

All the results we present are an average over several measurements after several warm-up iterations. Figure 2 presents the relative performance of DistilBERT compressed models for the production

Table 1: DistilBERT baseline and compressed models – SQuADv1.1 results

Model	F1 Score	Model Source/Generation
FP32 dense (baseline)	85.80%	Taken from [12]
INT8 sparse (ONNX Runtime)	85.64%	Generated by INC [5]
INT8 sparse (Neural Magic)	86.26%	Downloaded from [9]
INT8 sparse (Ours)	86.07%	Generated by INC [5]

scenario. Table 4 in Appendix A.4 shows the absolute performance accordingly. Our solution shows a consistent performance gain of 3.6x-4.1x over ONNX Runtime and outperforms Neural Magic by up to 50% for a range of sequence lengths. Table 2) presents the results for the maximum TP and minimum latency scenarios. We observe that our solution yields results comparable to the Neural Magic solution and performs 2x-3x better than ONNX Runtime. Our Transformer inference solution demonstrates high efficiency on common CPU production environments. To the best of our knowledge, we offer the best inference solution for Transformer deployment on CPU.

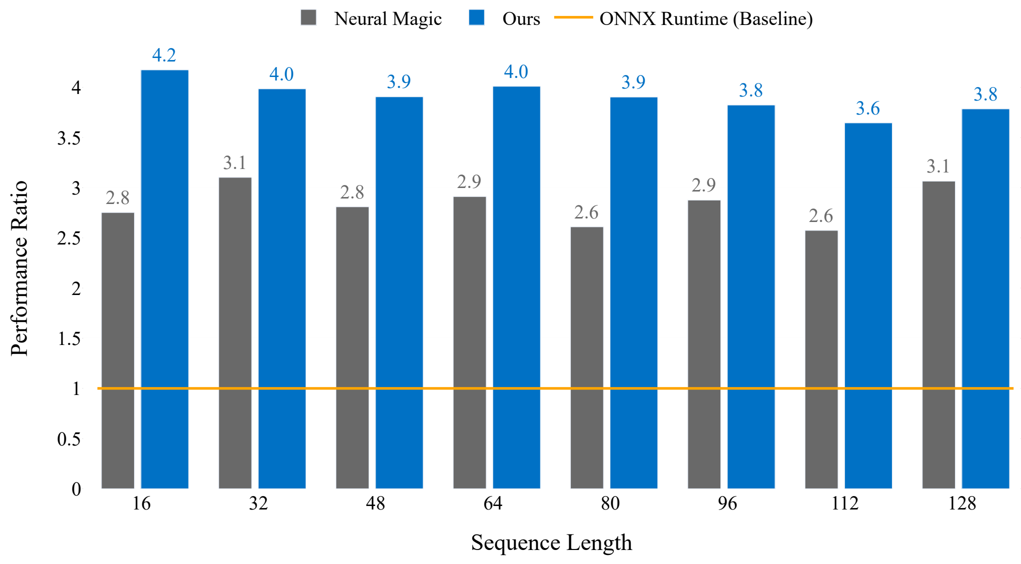

Figure 2: Production scenario results, maximum TP under 10ms inference latency

Table 2: Maximum TP and minimum latency results with sequence length 32

Model	Maximum Throughput (Samples/second)	Minimal Latency (Milliseconds/sample)
INT8 sparse (ONNX Runtime)	1920	2.76
INT8 sparse (Neural Magic)	5904	1.24
INT8 sparse (Ours)	6002	1.27

6 Summary and future work

In this paper, we presented a combined hardware-aware model compression technique (block-wise structured sparsity, knowledge distillation, and quantization) and demonstrated it with DistilBERT. We created a new dedicated inference engine which unlocks the performance of extremely compressed Transformer-based LMs on CPUs. Our results outperform Neural Magic's inference solution by up to 50% and demonstrate up to 4.1x better performance than ONNX Runtime under production constraints. We plan to apply the new model compression technique to other popular Transformer-based models and demonstrate the inference efficiency using our inference solution. Code and models for reproducing the published results will be released upon paper publication.

References

[1] A. Bhandare, V. Sripathi, D. Karkada, V. Menon, S. Choi, K. Datta, and V. Saletore. Efficient 8-bit quantization of transformer neural machine language translation model. *arXiv preprint arXiv:1906.00532*, 2019.

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[4] G. E. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *ArXiv*, abs/1503.02531, 2015.

[5] Intel. Neural compressor. *https://github.com/intel/neural-compressor*, 2020.

[6] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[7] E. Kurtic, D. Campos, T. Nguyen, E. Frantar, M. Kurtz, B. Fineran, M. Goin, and D. Alistarh. The optimal bert surgeon: Scalable and accurate second-order pruning for large language models. *arXiv preprint arXiv:2203.07259*, 2022.

[8] M. Kurtz, J. Kopinsky, R. Gelashvili, A. Matveev, J. Carr, M. Goin, W. Leiserson, S. Moore, B. Nell, N. Shavit, and D. Alistarh. Inducing and exploiting activation sparsity for fast inference on deep neural networks. In H. D. III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5533–5543, Virtual, 13–18 Jul 2020. PMLR. URL `http://proceedings.mlr.press/v119/kurtz20a.html`.

[9] NeuralMagic. Sparsezoo. *https://sparsezoo.neuralmagic.com/*, 2022.

[10] J. Pool, A. Sawarkar, and J. Rodge. Accelerating inference with sparsity using the nvidia ampere architecture and nvidia tensorrt. *NVIDIA Developer Blog*, 2021.

[11] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[12] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[13] V. Sanh, T. Wolf, and A. Rush. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in Neural Information Processing Systems*, 33:20378–20389, 2020.

[14] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[15] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8612–8620, 2019.

[16] Z. Yao, R. Y. Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *arXiv preprint arXiv:2206.01861*, 2022.

[17] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 36–39. IEEE, 2019.

[18] O. Zafrir, A. Larey, G. Boudoukh, H. Shen, and M. Wasserblat. Prune once for all: Sparse pre-trained language models. *arXiv preprint arXiv:2111.05754*, 2021.

[19] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.

A Appendix

Due to the space limitation of the main , we provide the additional details on model compression, software acceleration, and performance measurement in this appendix.

A.1 Block-wise structured sparsity

The required sparsity pattern by our sparse GEMM operators, described in Section 3, is 1-dimension blocks of size 4 in the output dimension of the weight in the case of a Linear layer. We extend the pruning framework proposed by Zafrir et al. [18]. In each pruning step a score is computed for each block with a pre-defined heuristic. The blocks with the lowest scores' magnitude are pruned to reach the required sparsity ratio. In this work, the use the average heuristic. The score for each block is the average of the magnitudes of the weights inside the block.

A.2 Model preparation

Following the PruneOFA [18] process we create a sparse pre-trained LM with the required block-wise sparsity as described in Section 2. We take English Wikipedia and BookCorpus [19] as our pre-training dataset. We fine-tune BERT-Base[*] on the pre-training dataset in the teacher preparation step. In the student pruning step we prune DistilBERT[*] in the required block-wise structured pattern with knowledge distillation from the teacher we prepared in the previous step.

After obtaining the pre-trained sparse DistilBERT we fine-tune it with pattern-lock[18] to SQuADv1.1 question-answering benchmark using the following hyper-parameters in Table 3, where λ_{kd} denotes student-teacher paired loss and λ_{MLM} denotes student-ground truth loss.

Table 3: Hyper-parameters for sparse DistilBERT

Hyper-parameter	Value
Learning rate	0.00018
Batch Size	12
Weight decay	0.01
Epochs	8
Learning rate decay	Linear
Warmup ratio	0.05
Sequence length	384
λ_{MLM}	0
λ_{kd}	1
Temperature	2

Note that we successfully demonstrate an end-to-end CPU solution on AWS from fine-tuning with above hyper-parameters on a c6i.32xlarge instance to inference on a c6i.12xlarge instance without any special hardware like GPUs or purpose-built accelerators.

[*] `https://huggingface.co/bert-base-uncased`

[*] `https://huggingface.co/distilbert-base-uncased`

A.3 Software acceleration

In this section, we describe the advanced memory allocator, weight sharing, graph optimization, and sparse GEMM operators which are required to build up our inference solution and demonstrate the high inference efficiency.

A.3.1 Advanced memory allocator

The default memory allocator usually creates a new buffer each time when receiving a memory allocation request, and therefore the data is less likely to be reused. To maximize the buffer reuse and make the data more cache friendly, we develop an advanced memory allocator as shown in Figure 3.

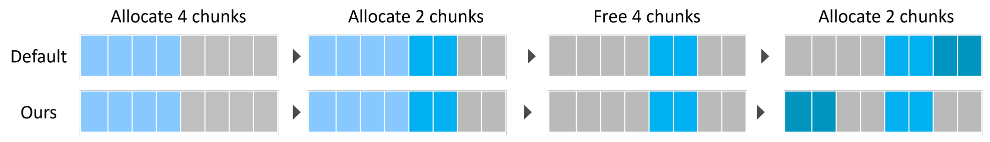

Figure 3: Advanced memory allocator (Ours) vs. default memory allocator (Default)

A.3.2 Weight sharing

Weight sharing is another useful runtime optimization used for efficient weight reuse in inference. Figure 4 shows how weight sharing works. The left one shows each inference instance (blue box) with the separate weight (grey box). If users want to run eight instances in parallel, eight copies of weight are needed, therefore lowering the overall inference efficiency due to lack of memory reuse. The right one shows one shared weight across multiple instances after applying weight sharing mechanism.

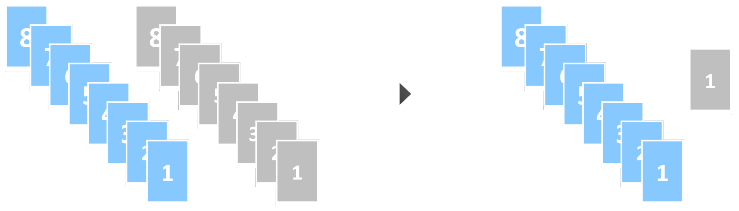

Figure 4: Weight sharing

A.3.3 Graph optimization

Figure 5 shows the staging transformation from FP32 graph (a), default INT8 graph (b), and optimized INT8 graph (c). Note that the optimized INT8 graph fuses all the post-operators (e.g., Bias + Reshape, LayerNorm) followed by InnerProduct or Matmul.

A.3.4 Sparse GEMM operators

We develop the optimized kernels for sparse GEMM operators based on the pre-defined block-wise structured sparsity pattern with constant block size 4. As described in Section 1, non-zero weight block is broadcasted to form a VNNI-format block; the sparse weight is compressed and the mask is used to filter the corresponding input as another VNNI-format block for matrix multiplication using VNNI. Algorithm 1 describes the code snippet of INT8 sparse GEMM kernel.

Note that for the weight with 4 non-dividable sparsity dimension, the additional padding is needed while this is not common in NLP models. For simplicity, we omit the special handling of padding in the sparse GEMM kernel implementation.

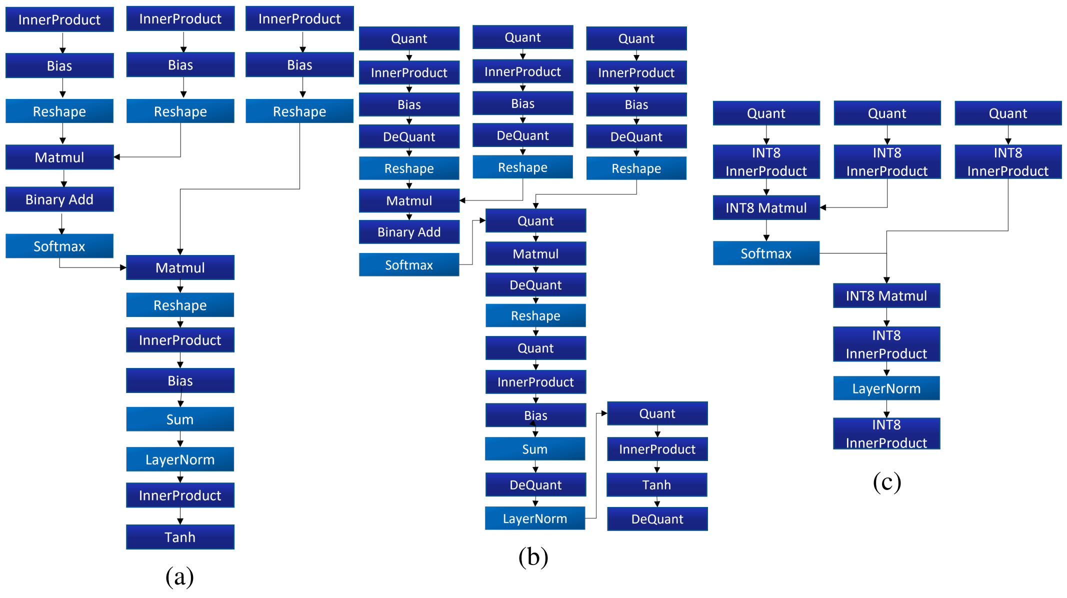

Figure 5: Graph optimization

Algorithm 1: Code snippet of INT8 sparse GEMM kernel

```
// $M, N, K$ as three dimensions of GEMM
// $m\_block$ = 4, $n\_block$ = 64, $k\_block$ = 4
// $weight\_ptr$ refers to weight tensor, $src\_ptr$ refers to input tensor
for $m = 0; m < M; m+ = m\_block$ do
    for $n = 0; n < N; n+ = n\_block$ do
        for $k = 0; k <= K; k+ = k\_block$ do
            $vbroadcastss(\_m32i(weight\_ptr))$
            $vbroadcastss(\_m32i(weight\_ptr))$
            $vbroadcastss(\_m32i(weight\_ptr))$
            $vbroadcastss(\_m32i(weight\_ptr))$
            for $i = 0; i < 4; + + i$ do
                $vmovdqu8(\_m128i, src\_ptr)$
                $vmovdqu8(\_m128i, src\_ptr)$
                $vbroadcasti32x4(\_m512i, \_m128i)$
                $vbroadcasti32x4(\_m512i, \_m128i)$
                $vpermt2d(\_m512i, \_m512i, \_m512i)$
                $vpshufb(\_m512i, \_m512i, \_m512i)$
            end for
            $vpdpbusd(\_m512i, \_m512i, \_m512i)$
            $vpdpbusd(\_m512i, \_m512i, \_m512i)$
            $vpdpbusd(\_m512i, \_m512i, \_m512i)$
            $vpdpbusd(\_m512i, \_m512i, \_m512i)$
            // downconvert and post-operator fusion
        end for
    end for
end for
```

A.4 Performance measurement

A.4.1 Software

We use the latest stable release ONNX Runtime v1.11.1 and Neural Magic v1.1.0 community edition (a436ca67) to measure the performance.

A.4.2 Performance

Table 4 shows the absolute maximum throughput (under 10 ms latency constraint) which is used to derive the relative performance in Figure 2.

Table 4: Maximum TP under latency constraint on DistilBERT

Sequence Length	ONNX Runtime (Samples/second)	Neural Magic (Samples/second)	Ours (Samples/second)
16	2713	7467	11323
32	1365	4236	5440
48	930	2614	3634
64	683	1990	2741
80	515	1344	2010
96	437	1257	1671
112	355	913	1293
128	299	915	1130