Deploying Vision Foundation AI Models on the Edge. The SAM2 Experience

Zheshuo Lin[1], Ruben Tous[1][0000−0002−1409−5843], and Beatriz Otero[1][0000−0002−9194−559X]

Universitat Politecnica de Catalunya, Jordi Girona, 1-3, Barcelona, Spain
ruben.tous@upc.edu

Abstract. As AI-driven applications expand across industries, the need for efficient edge computing solutions becomes increasingly critical. Traditional AI models are designed for high-performance cloud infrastructures, but emerging constraints—such as privacy requirements, network limitations, and real-time processing needs—necessitate optimized deployment on resource-constrained edge devices. This study presents a practical experience in adapting Segment Anything Model 2 (SAM2), a vision foundation model, for edge AI environments. The adaptation process involved translating the model to C++ using ONNX Runtime, enabling efficient execution on heterogeneous hardware. Experimental evaluations demonstrate that deploying SAM2 at the edge enhances processing efficiency, reduces reliance on network stability, and improves real-time responsiveness. This research provides valuable insights into AI in pervasive computing environments, contributing to the sustainable and scalable deployment of foundation models on edge devices.

Keywords: Artificial Intelligence · edge computing · foundation models · Segment Anything Model 2 · SAM2 · Computer Vision · segmentation.

1 Introduction

The rapid advancement of Artificial Intelligence (AI) has fueled its integration into diverse domains, from healthcare and industrial automation to consumer electronics and smart cities. Traditionally, AI workloads have been executed on high performance computing (HPC) infrastructures or cloud-based systems, using vast computational resources to train and deploy sophisticated models. However, the growing demand for real-time processing, data privacy, and reduced network dependency has shifted focus toward edge computing—bringing AI closer to the data source.

Edge computing enables AI-driven decision-making on local devices, such as smartphones, IoT sensors, and embedded systems, without relying on cloud-based processing. Although this approach reduces latency and enhances privacy, it also presents significant challenges. Edge devices are often constrained in terms of computational power, memory, and energy efficiency, making the deployment of large-scale AI models particularly difficult. Consequently, optimizing AI models for efficient execution on edge devices has emerged as a critical research focus.

This study explores the adaptation of Segment Anything Model 2 (SAM2) [14], a vision foundation model originally designed for cloud-based deployment, for execution in edge AI environments. The adaptation process involved translating the model to C++ using ONNX Runtime [3], enabling efficient execution on heterogeneous edge hardware.

2 Related Work

Vision Foundation Models (VFMs) are large-scale neural networks trained on massive image datasets to learn general visual representations. Many VFMs, such as SAM2 [14], CLIP [13], and DINO [2], are based on Vision Transformers (ViTs) [4], which offer strong performance but are highly computationally demanding [11].

Improving the efficiency of Vision Transformers (ViTs), especially by reducing the quadratic cost of self-attention with respect to input length, has been a key research focus [19, 20, 11]. While the original ViT adopts a plain, non-hierarchical design, models like Swin [10], MViT [6], PViT [16], and Hiera [15] introduce hierarchical, multi-stage structures. These achieve strong performance but are often slower in practice [15]. To improve speed and efficiency, hybrid models combining ViTs with convolutions—such as EfficientViT [9] and MobileNetV4 [12]—have also been proposed.

This paper focuses on SAM2 [14], a VFM capable of segmenting and tracking any object in video using interactive prompts like points and bounding boxes. SAM2 achieves strong performance and versatility across vision tasks, employing a hierarchical ViT-based image encoder, Hiera [15]. However, its main efficiency bottleneck lies in the memory module, which uses past frame information for consistent tracking. The large number of tokens in cross-attention leads to significant computation and memory overhead [17]. Recent works aim to reduce these costs for both SAM [21, 22, 18] and SAM2 [17].

A recent study [17] proposes a lightweight variant with a non-hierarchical encoder and optimized memory module for efficient mobile deployment. In [8], the authors introduce Efficient Frame Pruning (EFP), which optimizes the memory bank by retaining only the most informative frames. While such methods improve SAM2's efficiency, they overlook deployment challenges on edge devices. Python's high memory use, runtime overhead, and dependency issues make it unsuitable for resource-constrained environments, where users typically prefer self-contained applications over managing interpreters and dependencies. To address these challenges, this paper presents a C++ implementation of SAM2 that optimizes memory usage and reduces computational overhead, while improving compatibility and simplifying deployment across various hardware platforms.

3 Methodology

3.1 Preliminaries

SAM2 Architecture. SAM2 builds upon SAM [5], a model designed for the Promptable Visual Segmentation (PVS) task, where the goal is to generate a valid segmentation mask based on an input prompt, such as a bounding box or a point, that indicates the object of interest. SAM2 extends SAM's capabilities to the video domain, using point, box, and mask prompts on individual frames to define the spatio-temporal extent of the object to be segmented. Prompts can be applied at any frame in the video—enabling the correction of incorrectly tracked masks—and can be either positive or negative.

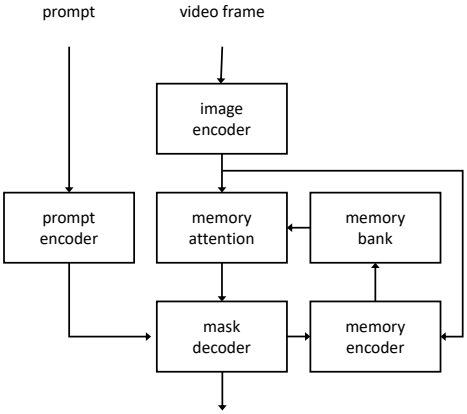


Fig. 1: Outline of the SAM 2 general architecture.

The SAM2 architecture comprises an image encoder, a prompt-guided mask decoder, and a memory mechanism (see Figure 1). The *image encoder* processes each frame (with a fixed input resolution of 1024^2) once, generating unconditioned tokens (feature embeddings) that represent the content of each frame. SAM2 uses a hierarchical vision transformer, Hiera [12], to handle image encoding. The encoded features of the current frame are then passed into the memory attention phase, where they are conditioned on features from previous frames, past predictions, and any new prompts. This phase consists of multiple stacked transformer blocks that perform both self-attention and cross-attention with memories of prior frames stored in a memory bank.

The *prompt encoder* processes various input types, such as points (positive or negative), bounding boxes, or masks, according to the design described in [20]. Sparse prompts are encoded using positional encodings combined with learned embeddings specific to each prompt type. Input masks are embedded through convolutional layers and integrated with the frame embedding. The *mask decoder*

consists of stacked transformer blocks that update both the prompt and frame embeddings. It can generate multiple candidate masks for each frame. If no follow-up prompts clarify ambiguities, the model propagates only the mask with the highest predicted Intersection over Union (IoU) for the current frame. In cases where no valid object is present in a frame, the model is designed to handle this scenario appropriately.

The *memory encoder* generates inputs for the *memory bank* by down-sampling the output mask and combining it element-wise with the unconditioned frame embedding from the *image encoder*. The *memory bank* stores spatial feature maps for the past N frames ($N = 6$) and separately for the past prompted frames. Furthermore, the *mask decoder's* 256-dimensional output tokens for each frame are retained as object pointers, serving as compact vectors that encode high-level semantic information about the object to be segmented.

C++ and ONNX Runtime. To facilitate the deployment of SAM2 on edge devices and commodity hardware, this paper presents the migration of the model from Python to C++, addressing challenges like high memory usage, runtime overhead, and the complexity of managing dependencies inherent in Python. One of the main practical constraints is that many personal computers lack a proper Python setup, and end users generally prefer self-contained applications rather than managing interpreters and dependencies manually. Additionally, Python utilities designed to generate self-contained executables often struggle with complex deployments, leading to inefficient and excessively large executables.

To overcome these challenges and enable efficient, flexible deployment, SAM2 was converted to the ONNX format [1], a standardized representation that supports interoperability across different frameworks. When combined with ONNX Runtime [3], this conversion allows for execution in C++ while leveraging built-in optimizations such as node pruning and operator fusion. These features are essential for performance in resource-constrained environments, ensuring that the model can run effectively on a wide range of hardware.

ONNX Runtime is a high-performance execution engine that supports running the model across diverse hardware platforms, including CPUs, GPUs, and specialized accelerators like TensorRT and OpenVINO. Its optimizations reduce both latency and overhead, making it particularly well-suited for deployment on edge and commodity hardware, where computational resources are often limited.

3.2 SAM2 Migration to C++

To achieve a C++ implementation of SAM2, the involved models were exported to the ONNX format, while the remaining code was directly translated to C++. This approach ensures that the inference code relies exclusively on the ONNX Runtime libraries, which are optimized for efficient and hardware-aware deployment. The process involved determining which parts could be directly translated into C++ and which required export to the ONNX format. The decision was driven by factors such as compatibility, computational efficiency, and the specific needs of the pipeline. This work has focused on migrating the image

segmentation-related parts, with plans to extend support for video segmentation in the future.

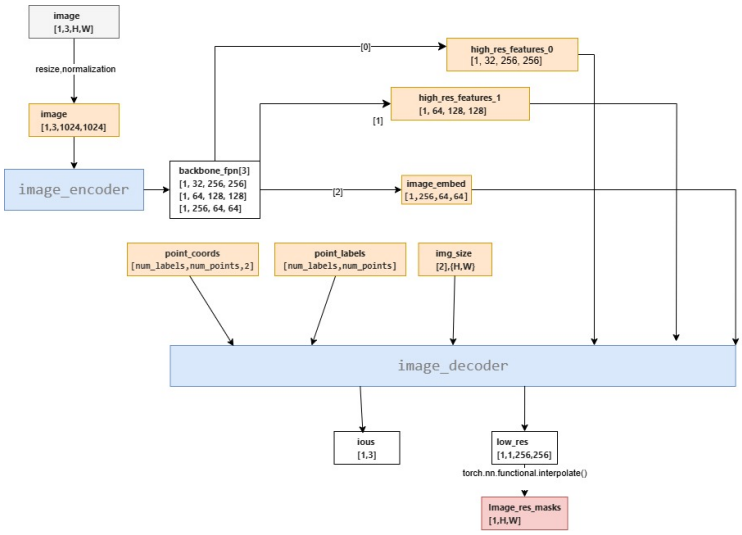


Fig. 2: Workflow of the resulting C++ implementation of SAM2.

The original *mask decoder* and *prompt encoder* components have been combined into a single model, referred to as the *image decoder*. Figure 2 illustrates the resulting overall workflow. First, the image is normalized and resized to fit the dimensions required by the image encoder (1024×1024). Then, the image is passed through the image encoder, which produces three outputs: `image_embed`, a compact representation of the image containing semantic information, and `high_res_features` 0 and 1, which hold spatial details useful for accurate segmentation. Next, specific points are provided as prompts to guide the segmentation process. Finally, these points, along with the outputs from the image encoder, are passed to the image decoder, which segments the image and generates the corresponding masks. Once the masks are generated, they are resized to match the original size of the input image.

In the *image encoder*, the normalization and resizing process was directly translated to C++, as these steps involve operations incompatible with ONNX. Additionally, performing these calculations in C++ was deemed more computationally efficient. The output was also restructured, as ONNX does not support tuple-based data formats.

The process of exporting models to ONNX format, programmed in Python, depends on both the original code and the checkpoints. It involves several steps. First, the model inputs must be prepared with the correct dimensions, ensuring each input has the expected shape, including tensors with initial values. Next, clear and descriptive names are assigned to the inputs and outputs for easier

identification in the exported format. For inputs with variable dimensions, such as point coordinates, dynamic axes are used to define which dimensions can vary, for example, the number of points, making the x and y coordinates dynamic. Finally, the model is exported to the ONNX format using `torch.onnx.export`, ensuring it meets the format's requirements. Additionally, the `onnx-simplifier` library (`onnxsim`) is used to simplify the model by reducing unnecessary computation nodes, removing redundant operations, and optimizing the structure of the computation graph, resulting in a more compact and efficient model for inference.

4 Experiments and results

4.1 Experimental setup

To evaluate the performance of the final implementation, a series of experiments were conducted to ensure that the accuracy of the original model was preserved. Computational performance improvements, including latency and memory usage, were assessed, along with an evaluation of the model's enhanced adaptability to heterogeneous hardware, such as CPUs and mid-range GPUs. The COCO dataset (Common Objects in Context) [7], a widely recognized benchmark offering a diverse range of images and annotations, was used to provide a comprehensive test for segmentation models.

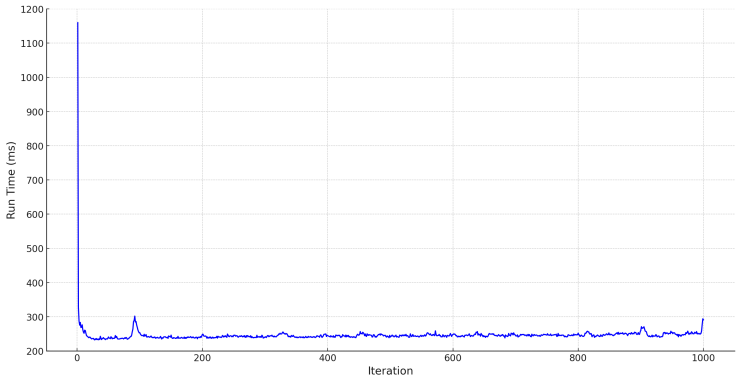


Figure 3: Execution Time on a Local Machine

The experiments were conducted on a local machine equipped with a mid-range GPU. Although the primary goal was not to improve results on a high-performance cloud server, results from a standard cloud setup were also presented to assess latency differences and confirm that model accuracy remained intact. The local hardware, used for testing the C++ implementation, featured an Alder Lake i7-12650H integrated SoC, 16GB of DDR4 RAM (8GB*2 at 3200MHz), a

1TB NVMe PCIe Gen4x4 SSD without DRAM, and an RTX 4050 graphics card with 6GB of GDDR6 memory, 120 tensor cores, and a 96-bit memory interface. The cloud hardware, used to obtain baseline results with the original Python implementation, included a Tesla T4 GPU with 16GB of GDDR6 memory, 320 tensor cores, and a 256-bit memory interface.

For robustness, each experiment was repeated five times, and the mean performance values are reported.

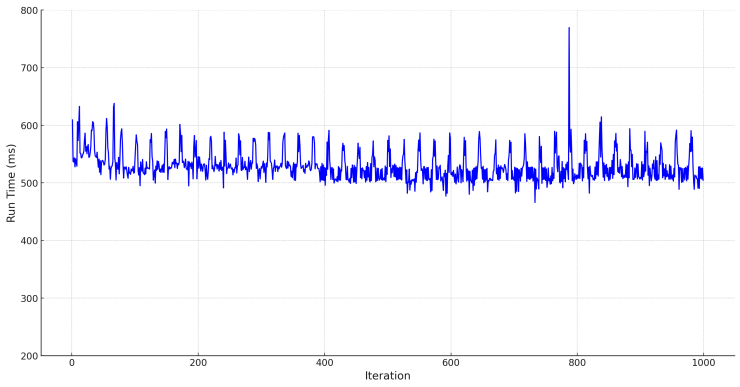


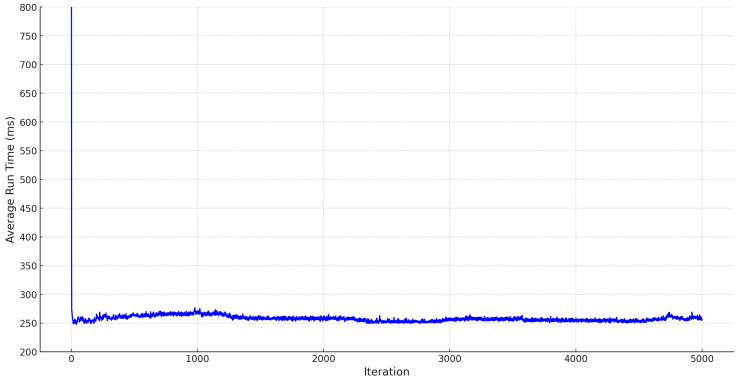
Figure 4: Execution Time in a Cloud Environment

4.2 Latency evaluation

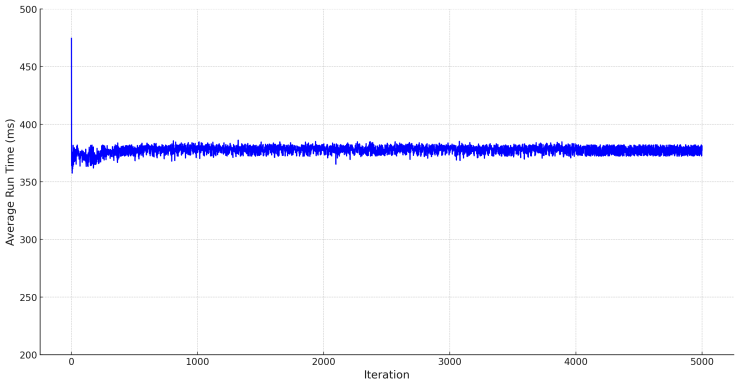
Several experiments were conducted to evaluate the average latency and stability of the implementation. A total of 1000 consecutive executions with varying images and prompts were performed on both the local machine and the cloud server. Figure 3 displays the execution time results for all 1000 runs using identical input (image and prompt).

The execution time of the ONNX-formatted model initially starts slow, with the first execution taking approximately 1 second. However, after this initial execution, the times stabilize around 250 ms (240 ms for the image encoder and 10 ms for the image decoder), with an average of 246 ms. This initial delay is attributed to factors such as library loading, initialization, caching, and computation graph optimizations.

The execution time results from the cloud environment are shown in Figure 4. Here, significant variability in execution times is observed, ranging from 470 ms to 750 ms, indicating a notably wide span. This variability is primarily due to network instability, which introduces high and fluctuating latency. Furthermore, the average execution time is 529 ms, which is approximately twice the execution time observed with the C++ implementation.



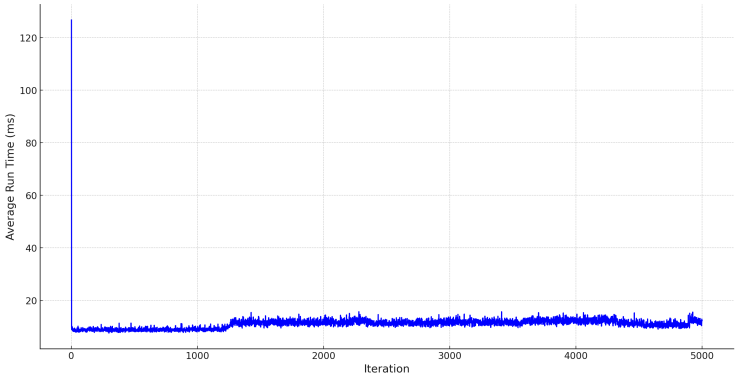
(a) Local Machine



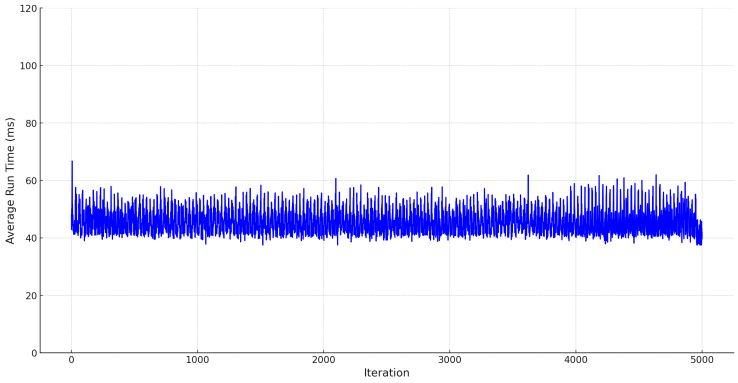
(b) Cloud Environment

Figure 5: Comparison of Average Image Encoder Execution Times: (a) Local Machine vs. (b) Cloud Environment

The same experiments were repeated with varying inputs, including different images and a varying number of points in the prompts. As expected, execution times remained consistent. Figure 5 shows the average execution times of the image encoder for different images, while Figure 6 presents the execution times for varying prompts.



(a) Local Machine



(b) Cloud Environment

Figure 6: Comparison of Average Image Decoder Execution Times: (a) Local Machine vs. (b) Cloud Environment

4.3 Accuracy evaluation

To verify that the resulting ONNX/C++ version behaves consistently with the original model, we compared the segmentation accuracy of both implementations using the COCO dataset, which contains segmentation masks. Since the COCO dataset does not provide points as input, we generated these points from the

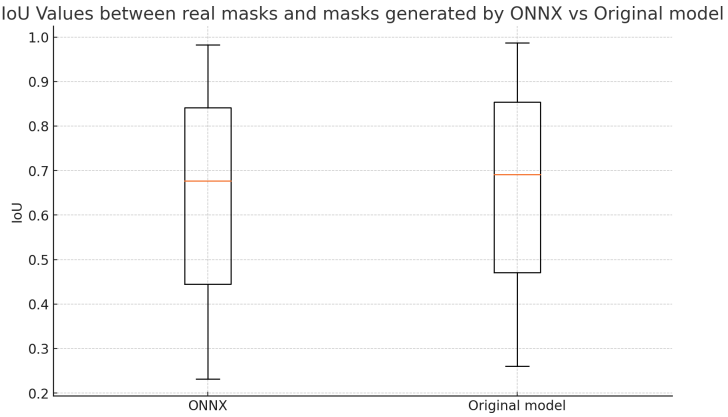
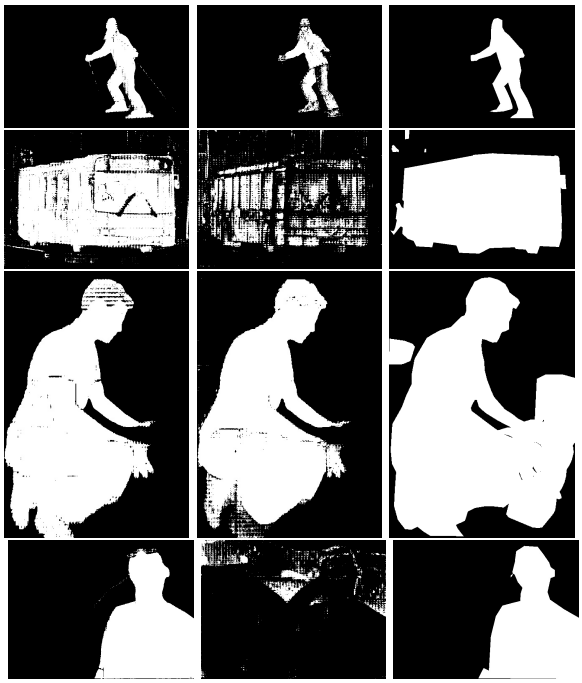


Figura 7: Box plot of the Jaccard index between the actual mask and the predicted mask.



(a) (b) (c)

Fig. 8: Example segmentation results. (**a**) Our C++/ONNX implementation; (**b**) original SAM2 implementation; (**c**) ground truth.

ground truth masks. The points were randomly sampled within the segments defined by the masks and then used as input to the image decoder. Figure 7 presents a boxplot comparing the IoU values obtained from both versions. This visualization reveals that the ONNX model exhibits a slight decrease in accuracy. However,rall performance remains within acceptable bounds for practical applications. Figure 8 shows some example segmentation results.

5 Conclusions

This work explored the challenges and strategies involved in optimizing the deployment of vision foundation models on Edge devices through a practical use case. The Segment Anything Model 2 (SAM2) was adapted and deployed by translating it to C++. ONNX and ONNX Runtime were used to support this transition while preserving model performance and flexibility. The results obtained show that the approach is feasible, achieving a functional deployment with an improvement in latency 50%. Our study demonstrates that an optimized Edge AI deployment not only enhances processing efficiency but also reduces the dependency on stable network connections, making AI applications more resilient and responsive. By investigating the trade-offs and strategies involved in adapting cloud-native AI models for edge computing, this research contributes to the broader goal of sustainable and scalable AI in ubiquitous computing environments.

Acknowledgements

This work is partially funded by the Spanish Ministry of Science and Innovation under contracts PID2019-107255GB and PID2021-124463OB-IOO, as well as by the SGR programs 2021-SGR-00478 and 2021-SGR-00326 of the Catalan government. Finally, the research that led to these results has received funding from the Collaboration Grant for the 2024/2025 period from the Spanish Ministry of Education.

References

1. Ahmed, S., Bisht, P., Mula, R., Dhavala, S.S.: A deep learning framework for interoperable machine learning. In: Proceedings of the First International Conference on AI-ML Systems. AIMLSystems '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3486001.3486243, https://doi.org/10.1145/3486001.3486243
2. Caron, M., Touvron, H., Misra, I., Jegou, H., Mairal, J., Bojanowski, P., Joulin, A.: Emerging properties in self-supervised vision transformers. In: 2021 IEEE/CVF International Conference on Computer Vision (ICCV). pp. 9630–9640 (2021). https://doi.org/10.1109/ICCV48922.2021.00951
3. developers, O.R.: Onnx runtime. https://onnxruntime.ai/ (2021)

4. Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al.: An image is worth 16x16 words: Transformers for image recognition at scale. In: International Conference on Learning Representations (2020)
5. Kirillov, A., Mintun, E., Ravi, N., Mao, H., Rolland, C., Gustafson, L., Xiao, T., Whitehead, S., Berg, A.C., Lo, W.Y., et al.: Segment anything. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 4015–4026 (2023)
6. Li, Y., Wu, C.Y., Fan, H., Mangalam, K., Xiong, B., Malik, J., Feichtenhofer, C.: Mvitv2: Improved multiscale vision transformers for classification and detection. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. pp. 4804–4814 (2022)
7. Lin, T., et al.: Microsoft COCO: common objects in context. CoRR **abs/1405.0312** (2014), http://arxiv.org/abs/1405.0312
8. Liu, H., Zhang, E., Wu, J., Hong, M., Jin, Y.: Surgical SAM 2: Real-time segment anything in surgical video by efficient frame pruning. CoRR **abs/2408.07931** (2024). https://doi.org/10.48550/ARXIV.2408.07931
9. Liu, X., Peng, H., Zheng, N., Yang, Y., Hu, H., Yuan, Y.: Efficientvit: Memory efficient vision transformer with cascaded group attention. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 14420–14430 (2023)
10. Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., Guo, B.: Swin transformer: Hierarchical vision transformer using shifted windows. In: Proceedings of the IEEE/CVF international conference on computer vision. pp. 10012–10022 (2021)
11. Papa, L., Russo, P., Amerini, I., Zhou, L.: A Survey on Efficient Vision Transformers: Algorithms, Techniques, and Performance Benchmarking . IEEE Transactions on Pattern Analysis & Machine Intelligence **46**(12) (2024). https://doi.org/10.1109/TPAMI.2024.3392941
12. Qin, D., Leichner, C., Delakis, M., Fornoni, M., Luo, S., Yang, F., Wang, W., Banbury, C., Ye, C., Akin, B., et al.: Mobilenetv4-universal models for the mobile ecosystem. arXiv preprint arXiv:2404.10518 (2024)
13. Radford, A., et al.: Learning transferable visual models from natural language supervision. In: Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event. Proceedings of Machine Learning Research, vol. 139, pp. 8748–8763. PMLR (2021)
14. Ravi, N., Gabeur, V., Hu, Y.T., Hu, R., Ryali, C., Ma, T., Khedr, H., Rädle, R., Rolland, C., Gustafson, L., et al.: Sam 2: Segment anything in images and videos. arXiv preprint arXiv:2408.00714 (2024)
15. Ryali, C., Hu, Y.T., Bolya, D., Wei, C., Fan, H., Huang, P.Y., Aggarwal, V., Chowdhury, A., Poursaeed, O., Hoffman, J., et al.: Hiera: A hierarchical vision transformer without the bells-and-whistles. In: International Conference on Machine Learning. pp. 29441–29454 (2023)
16. Wang, W., Xie, E., Li, X., Fan, D.P., Song, K., Liang, D., Lu, T., Luo, P., Shao, L.: Pyramid vision transformer: A versatile backbone for dense prediction without convolutions. In: Proceedings of the IEEE/CVF international conference on computer vision. pp. 568–578 (2021)
17. Xiong, Y., et al.: Efficient track anything. preprint arXiv:2411.18933 (2024)
18. Xiong, Y., Varadarajan, B., Wu, L., Xiang, X., Xiao, F., Zhu, C., Dai, X., Wang, D., Sun, F., Iandola, F., et al.: Efficientsam: Leveraged masked image pretraining

for efficient segment anything. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 16111–16121 (2024)
19. Xiong, Y., Zeng, Z., Chakraborty, R., Tan, M., Fung, G., Li, Y., Singh, V.: Nyströmformer: A nyström-based algorithm for approximating self-attention. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 35, pp. 14138–14148 (2021)
20. You, H., et al.: Castling-vit: Compressing self-attention via switching towards linear-angular attention at vision transformer inference. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 14431–14442 (2023)
21. Zhang, C., Han, D., Qiao, Y., Kim, J.U., Bae, S.H., Lee, S., Hong, C.S.: Faster segment anything: Towards lightweight sam for mobile applications. arXiv preprint arXiv:2306.14289 (2023)
22. Zhao, X., et al.: Fast segment anything. arXiv preprint arXiv:2306.12156 (2023)