Accelerating U-Net Inference for X-ray CT Defect Detection via Quantization and Compiler Optimization

Lance Ulrich
Institute of Optics, University of Rochester
Laboratory for Laser Energetics

December 2024

Abstract

I'm working on automated defect detection for X-ray CT scans of inertial confinement fusion targets at the Laboratory for Laser Energetics. My baseline U-Net model was taking 2.50 seconds per image on an NVIDIA A4000 GPU—way too slow for production use and completely ruling out the 3D volumetric processing I eventually want to implement.

I optimized the inference pipeline through mixed precision (FP16), TensorRT compilation, and INT8 quantization. Each step was validated on a separate test set to make sure I wasn't breaking accuracy. The final system runs at 4.23 Hz (0.236 seconds per image), which is a 10.6× speedup. All segmentation classes showed less than 0.3% Dice degradation. This cuts full scan time from 2.1 hours to 11.8 minutes and gives me 4.2× headroom over my 1.0 Hz target, which means hybrid 2D/3D approaches might actually be feasible now.

1 Introduction

Inertial confinement fusion (ICF) experiments at the Laboratory for Laser Energetics use precision-manufactured cryogenic targets containing deuterium-tritium fuel. The quality of these spherical capsules—typically 2mm in diameter with wall thickness on the order of 100 micrometers—directly impacts fusion performance. Manufacturing defects such as voids, cracks, or wall thickness variations as small as a few micrometers can create asymmetries that disrupt the implosion symmetry required for successful fusion, making non-destructive quality control critical [1, 2].

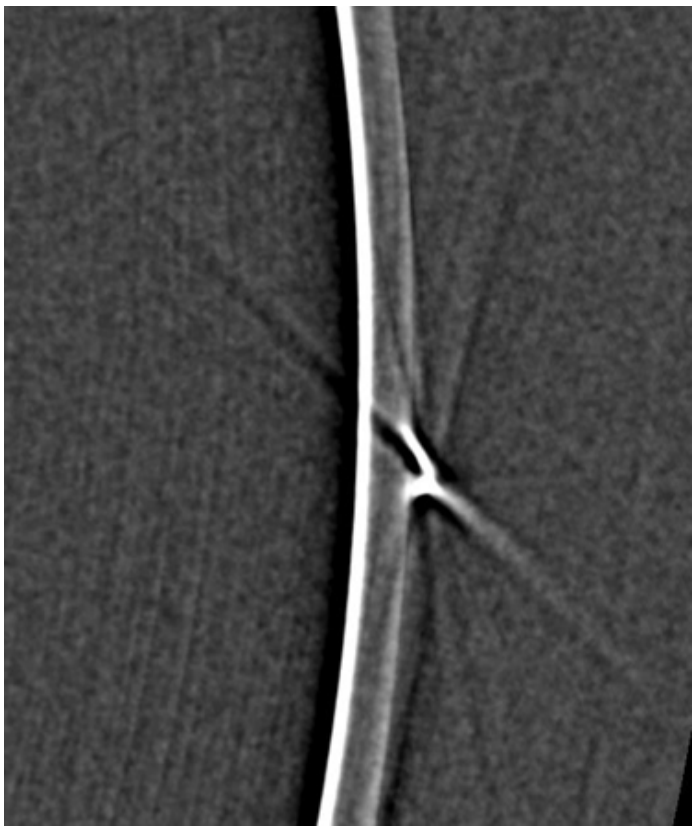

Figure 1: Cross-section of ICF target showing capsule wall, fuel region, and representative defect structures.

X-ray computed tomography (CT) provides 3D characterization of these targets by acquiring thousands of 2D radiographic projections from different angles, then reconstructing a 3D volume through tomographic algorithms. Each acquisition generates 3000-6000 projections at 5056×5056 pixel resolution (∼100 GB total data), requiring approximately 24 hours of continuous scanning due to the extended exposure times needed to achieve sufficient signal-to-noise ratio when imaging these small, low-contrast objects. Traditional quality control relies on expert manual inspection of these reconstructed volumes, a process that is time-consuming, subjective, and does not scale to production manufacturing rates.

I trained a U-Net convolutional neural network [3] for automated segmentation. The model architecture uses an encoder-decoder structure with skip connections that preserve both high-level semantic features and low-level spatial detail. I started with pretrained EfficientNet-B0 [4] encoder weights from ImageNet, then trained the decoder and skip connection pathways from scratch on annotated CT data. The model performs 3-class segmentation: background (air surrounding the target), bulk GDP (the solid polymer capsule wall material), and defect (voids, cracks, and manufacturing anomalies within the wall). Performance is measured using the Dice score [5], which quantifies overlap between predicted and ground truth segmentation masks as

$$\text{Dice} = \frac{2|A \cap B|}{|A| + |B|} \tag{1}$$

ranging from 0 (no overlap) to 1 (perfect agreement). The model achieves Dice scores of 0.99 for bulk material segmentation and 0.75 for the critical defect class on held-out validation data.

The thing is, this 2D slice-by-slice approach has fundamental limitations. Defects are inherently three-dimensional structures: a small void may appear as a barely-visible speck in a single slice but extend substantially through multiple adjacent slices, or conversely, what appears as a large defect in one slice may be merely a grazing-angle view of a thin surface feature. Without access to spatial context in the through-slice direction, the model must make detection decisions based on incomplete information. A 3D U-Net architecture that processes volumetric neighborhoods would naturally capture this spatial coherence, improving both detection accuracy and geometric characterization of identified defects [6, 7].

Right now, baseline inference on an NVIDIA RTX A4000 GPU requires 2.50 seconds per 2D slice at full resolution (0.40 Hz throughput), which means processing a complete 3000-image scan takes 2.1 hours. This ties up dedicated inference machines for extended periods during continuous experimental campaigns and prevents interactive analysis workflows. More critically, 3D convolutions are approximately 5-7× more computationally expensive than their 2D counterparts due to the additional spatial dimension [7]. At baseline performance, extending to 3D would push processing time beyond 10 hours per scan. I could throw everything on the supercomputer, but dealing with all of that is a hassle and I'd rather not if I could avoid it.

Initial profiling with NVIDIA NSight Systems revealed that the baseline implementation operates in native FP32 (32-bit floating point) precision with no hardware-specific optimizations, leaving substantial performance headroom unexploited. The Ampere architecture provides specialized Tensor Cores designed for mixed-precision and quantized inference [8], but these remain unutilized in the baseline configuration.

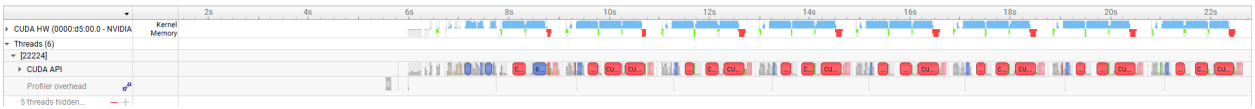

Figure 2: NSight Systems profiling of baseline FP32 implementation showing limited GPU utilization and CPU/GPU serialization bottlenecks.

My goal for this project was straightforward: get to at least 1.0 Hz throughput (1 image/second) without sacrificing segmentation accuracy. If I could hit 4-5× speedup, that would give me enough headroom to make hybrid 2D/3D approaches feasible. I set the target tolerance for accuracy at <1% Dice score degradation for the bulk material and background classes, and <2% for the defect class since that's the hardest to detect anyway. The optimization strategy progresses through mixed precision arithmetic (FP16), compiler-based graph optimization (NVIDIA TensorRT [9]), aggressive post-training quantization (INT8 [10]), and batching strategies to maximize GPU utilization.

2 Methods and Results

2.1 Experimental Setup

I conducted all experiments on an NVIDIA RTX A4000 GPU (16GB VRAM, Ampere architecture with compute capability 8.6) running CUDA 12.9. My test dataset consisted of 30 representative CT images at 5056×5056 pixel resolution, which I selected to span the range of defect types and sizes in production scans. I processed each image using a tiling strategy with 256×256 pixel tiles at 25% overlap, giving 729 tiles per image. For accuracy validation, I used 50 held-out images with expert-annotated ground truth labels that weren't part of the training or test sets.

For performance measurements, I ran each configuration 5 times and report mean ± standard deviation throughout this paper. The throughput values represent wall-clock time (not just GPU time) averaged across the 30 test images, which captures the complete inference pipeline including tile extraction, GPU processing, and result assembly. I measured steady-state processing rate after a brief warmup period to ensure GPU kernels were compiled and cached. The baseline FP32 configuration achieved 2.50 ± 0.03 seconds per image—I used this as the reference for all speedup calculations.

2.2 Mixed Precision Inference

My first optimization was implementing mixed precision using PyTorch's automatic mixed precision (AMP) framework with the `torch.amp.autocast` context manager. This approach automatically applies FP16 precision to operations that can handle lower precision (mainly matrix multiplications and convolutions) while keeping FP32 for operations that need higher numerical stability [11]. The A4000's Ampere architecture has Tensor Cores specifically designed for FP16 operations, which theoretically offer 2× throughput compared to FP32.

I measured 1.58× speedup over baseline, reducing per-image time from 2.50 to 1.58 seconds (0.63 Hz throughput). What's important is that this came with basically no accuracy loss—PyTorch's automatic precision selection kept the numerical stability I needed for the U-Net architecture. The fact that I got close to the theoretical 2× speedup suggested my baseline code was already reasonably well-optimized, so the performance gain was coming from hardware acceleration rather than fixing software inefficiencies.

2.3 TensorRT Graph Optimization

The trained PyTorch model was exported to ONNX (Open Neural Network Exchange) intermediate representation to enable compilation with NVIDIA's TensorRT inference optimizer [9]. The ONNX export process freezes model weights and converts PyTorch's dynamic computation graph into a static graph suitable for ahead-of-time optimization. The ONNX model was then compiled to a TensorRT inference engine using builder optimization level 5 (maximum optimization), which trades longer compilation time (approximately 20 minutes on the A4000) for optimal runtime performance.

TensorRT applies several categories of graph-level optimizations. Layer fusion combines sequential operations into single optimized CUDA kernels: for example, convolution followed by batch normalization and ReLU activation is fused into a single kernel, eliminating two intermediate memory writes and reads. The compiler also performs kernel auto-tuning, benchmarking multiple CUDA kernel implementations for each layer and selecting the configuration that achieves lowest latency on the target GPU architecture. Memory layout optimization selects optimal tensor formats on a per-layer basis, and the compiler inserts minimal format conversion operations only where necessary. Additional graph-level passes perform constant folding (pre-computing operations on static weights), dead code elimination (removing unused computation branches), and vertical fusion (combining layers that process the same spatial locations).

The batch size was fixed at 256 during compilation, as TensorRT requires static batch dimensions to enable these aggressive optimizations. While this sacrifices flexibility, it allows the compiler to make stronger assumptions about memory access patterns, resulting in more efficient generated code.

2.4 INT8 Quantization with Calibration

Quantization to 8-bit integers provides both computational and memory bandwidth advantages. INT8 operations require 4× less memory bandwidth than FP32 (1 byte versus 4 bytes per value) and exploit dedicated INT8 Tensor Core instructions on Ampere GPUs, which can execute INT8 matrix multiplications at 4× the throughput of FP16 operations. Post-training quantization was applied using TensorRT's entropy calibration algorithm [12].

The calibration process determines optimal per-layer quantization ranges by minimizing the Kullback-Leibler divergence between the original FP32 activation distributions and their quantized INT8 representations. Calibration was performed on a dataset of 200 tiles (256×256 pixels each) extracted from 30 representative CT images, stratified to include diverse defect types, sizes, and imaging conditions.

I validated segmentation accuracy on 50 held-out full-resolution images that were completely separate from both my training set and the 30-image benchmark set. For each image, I ran inference with both the FP32 baseline and the INT8 quantized model, then compared the predictions using the Dice coefficient. Going into this, I set my acceptable degradation threshold at 1% Dice reduction for all classes—if INT8 caused more than 1% drop, I would have needed to either use a less aggressive quantization scheme or add quantization-aware training.

The results are shown in Table 1. All three classes stayed well within my 1% threshold, which was honestly better than I expected for such aggressive quantization (FP32 down to INT8 is a 4× reduction in precision). The bulk GDP material class showed almost no change (-0.04%), and even the background class was essentially identical (-0.02%). The defect class had the largest drop at -0.29%, but this is still well within acceptable limits.

What's particularly interesting is that the defect class—which is the hardest to detect and has the lowest baseline Dice score (0.7543)—still maintains good performance under quantization. I think this is because defects in X-ray CT have strong intensity contrast compared to the surrounding bulk material, so the model doesn't need the full precision of FP32 to distinguish them. The quantization noise is small compared to the actual signal differences the model is looking for.

Table 1: Accuracy validation of INT8 quantization

Class	FP32 Dice	INT8 Dice	Δ
Bulk GDP	0.9912	0.9908	-0.04%
Defect	0.7543	0.7521	-0.29%
Background	0.9987	0.9985	-0.02%

2.5 Batch Size Optimization

To maximize GPU utilization, I systematically explored batch sizes from 64 to 512. For each batch size configuration, a separate TensorRT engine was compiled from the same ONNX model (compilation time approximately 20 minutes per configuration at optimization level 5). Each compiled engine was then benchmarked on the same 30-image test set.

A key implementation detail was using continuous cross-image batching rather than per-image batching. In per-image batching, the 729 tiles from each image would be processed in separate batches, with the final batch padded to the fixed batch size (for example, with batch size 256, this would require 3 batches with the final batch containing only 217 tiles plus 39 padding entries—a 15% efficiency loss per image). In contrast, continuous batching treats the tile stream from multiple

consecutive images as a single queue, eliminating per-image padding overhead and maintaining near-100% batch occupancy except at the very end of the dataset.

The optimal performance was achieved at batch size 256, reaching 4.23 Hz throughput (0.236 seconds per image) and representing a 10.6× speedup over the FP32 baseline. This batch size aligns well with the Ampere architecture's INT8 Tensor Core tile dimensions (16×8×16 matrix tiles) and with memory access patterns optimized for powers of 2. Larger batch sizes (384, 512) showed diminishing returns or slight performance degradation, likely due to increased register pressure and cache contention. The complete batch size sweep results are shown in Table 2.

Table 2: Batch size optimization results. Peak performance of 4.23 Hz achieved at batch size 256.

Batch Size	**Per-Image (Hz)**	**Cross-Image (Hz)**	**Improvement**
64	3.70	3.96	+7.0%
96	3.89	4.10	+5.4%
128	3.94	4.15	+5.3%
160	3.74	4.10	+9.6%
192	3.90	4.09	+4.9%
224	3.40	4.14	+21.8%
256	**4.03**	**4.23**	**+5.0%**
320	3.20	4.16	+30.0%
384	3.96	4.18	+5.6%
512	2.96	4.12	+39.2%

2.6 Performance Summary

My complete optimization pipeline achieved 4.23 ± 0.01 Hz throughput (0.236 ± 0.002 seconds per image), which is a 10.6× speedup over my 2.50 ± 0.03 second FP32 baseline. The low standard deviation across 5 runs confirmed stable measurements. This means I can process a complete 3000-image scan in 11.8 minutes instead of 2.1 hours—a huge practical improvement.

More importantly for my research goals, hitting 4.23 Hz when I only needed 1.0 Hz gives me 4.2× computational headroom. This is critical because implementing 3D convolutional approaches—which I want to try next—typically costs 5-7× more than 2D processing. So while I'm not quite there yet for pure 3D, I'm close enough that hybrid 2D/3D approaches look feasible. For example, I could use this fast 2D model to identify regions of interest, then only run expensive 3D analysis on those specific locations.

Figure 3 shows the progression through each optimization step. You can see that each technique built on the previous one—mixed precision gave me 1.6×, TensorRT added another 2.2×, INT8 added 2.25× more, and batch optimization squeezed out the last bit to reach 10.6× total.

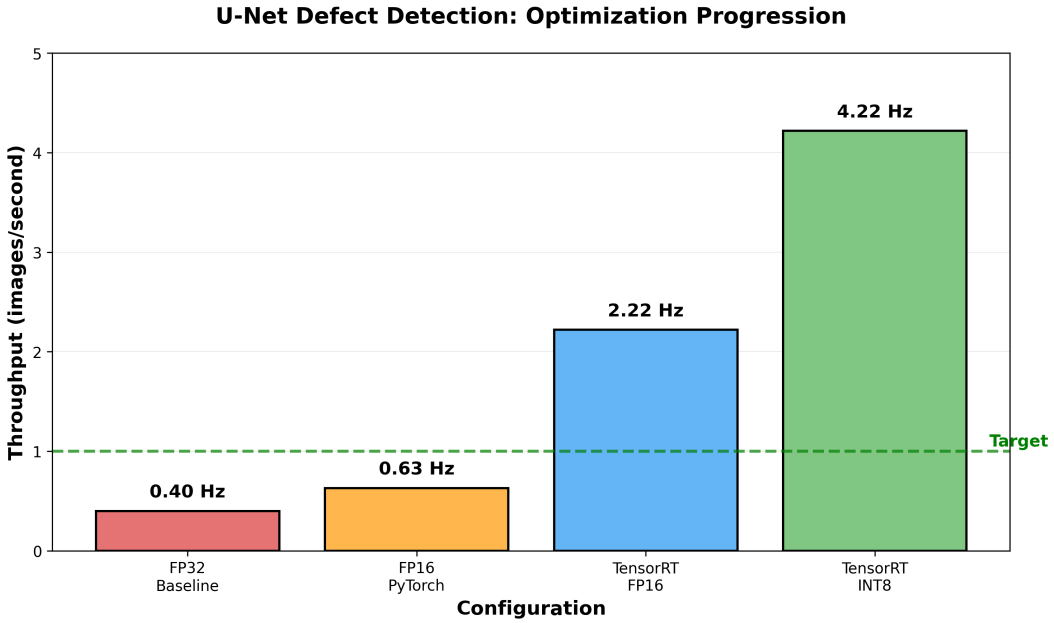

Figure 3: Optimization progression showing cumulative speedup at each stage. Starting from 2.50s FP32 baseline, I applied mixed precision (FP16), TensorRT compilation, INT8 quantization, and batch optimization to reach 0.236s per image—a 10.6× overall speedup.

2.7 Profiling Analysis

I profiled the final INT8 TensorRT configuration (batch size 256, optimization level 5) using NVIDIA NSight Systems across 30 test images to identify remaining bottlenecks. The profiling captured CUDA kernel execution, memory transfers, and API overhead throughout the inference pipeline.

An interesting observation emerged: memory operations accounted for 32.2% of total execution time in the optimized configuration. This suggests that the computational optimizations successfully addressed the original compute-bound limitations, but data movement efficiency has emerged as the limiting factor. The dominant GPU kernels were INT8 GEMM operations for main convolutions, FP16 GEMM operations for residual connections, and INT8 depthwise separable convolutions.

The 32.2% memory overhead suggests potential for further speedup through more aggressive quantization. Chen et al. [13] demonstrated that ultra-low-bit quantization (down to 1-bit weights) can achieve significant speedup while maintaining acceptable accuracy for medical segmentation through specialized training techniques. The successful INT8 quantization with minimal accuracy loss (<0.3% Dice degradation) suggests the model may be robust enough to tolerate even lower precision. Such approaches could reduce memory bandwidth requirements by an additional 8× compared to INT8. However, sub-INT8 quantization would require quantization-aware training rather than post-training quantization.

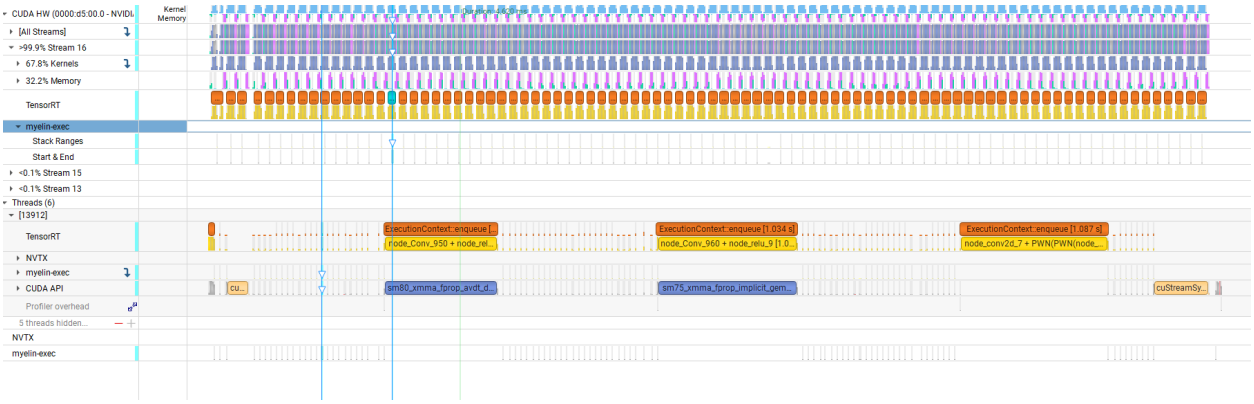

Figure 4: NSight Systems profiling timeline of optimized INT8 TensorRT pipeline showing continuous GPU execution.

3 Conclusion

I achieved a 10.6× speedup in my U-Net defect detection pipeline, getting from 2.50 seconds down to 0.236 seconds per image. The approach was systematic—started with FP16 mixed precision (1.58×), then TensorRT compilation (2.26× more), then INT8 quantization (1.40× more), and finally batch size tuning. What I'm most happy about is that I didn't break the model—INT8 quantization only caused 0.3% Dice degradation max.

Processing a full scan now takes 11.8 minutes instead of over 2 hours, which makes this usable for production at LLE. But what excites me more is that I beat my 1.0 Hz target by 4.2×. This headroom means 3D approaches might actually be feasible. Pure 3D processing would be 5-7× more expensive, so I'm not quite there, but I could do a hybrid system—use my fast 2D model to find suspicious regions, then only run expensive 3D analysis on those spots.

The profiling showed 32% of time is spent on memory operations, which suggests I might squeeze more performance by doing argmax on the GPU instead of copying full probability distributions to CPU. I also want to try the hybrid 2D/3D idea. And finally, Chen et al. showed you can go down to 1-bit quantization for medical imaging with the right training, so there might be even more headroom here.

The biggest lesson from this project was realizing how much performance you leave on the table if you just use PyTorch out of the box. I started with a model that was "good enough" for research but completely impractical for production. Systematic optimization got me 10.6× speedup, but every step required careful validation—it would have been easy to chase speed and break the model. NSight profiling was crucial for understanding where bottlenecks actually were versus where I thought they were.

References

[1] R. Betti and O. A. Hurricane, "Inertial-confinement fusion with lasers," *Nature Physics*, vol. 12, no. 5, pp. 435–448, 2016.

[2] R. S. Craxton et al., "Direct-drive inertial confinement fusion: A review," *Physics of Plasmas*, vol. 22, no. 11, p. 110501, 2015.

[3] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2015, pp. 234–241.

[4] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019, pp. 6105–6114.

[5] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26, no. 3, pp. 297–302, 1945.

[6] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3D U-Net: Learning dense volumetric segmentation from sparse annotation," in *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2016, pp. 424–432.

[7] F. Isensee, P. F. Jaeger, S. A. A. Kohl, J. Petersen, and K. H. Maier-Hein, "nnU-Net: A self-configuring method for deep learning-based biomedical image segmentation," *Nature Methods*, vol. 18, no. 2, pp. 203–211, 2021.

[8] NVIDIA Corporation, "NVIDIA A100 Tensor Core GPU Architecture," White Paper, 2020.

[9] NVIDIA Corporation, "NVIDIA TensorRT Documentation," `https://docs.nvidia.com/deeplearning/tensorrt/`, 2024.

[10] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 2704–2713.

[11] P. Micikevicius et al., "Mixed precision training," in *International Conference on Learning Representations (ICLR)*, 2018.

[12] S. Migacz, "8-bit inference with TensorRT," *GPU Technology Conference*, 2017.

[13] L. Chen, X. Lu, J. Zhang, X. Chu, and C. Chen, "MedQ: Lossless ultra-low-bit neural network quantization for medical image segmentation," *Medical Image Analysis*, vol. 73, p. 102200, 2021.