Optimizing Adversarial Defenses With CUDA

Johnny Tavares
University of Rochester
jtavare3@u.rochester.edu

***Abstract*—Deep convolutional neural networks such as ResNet-50 are increasingly relied on to provide image recognition and classification tasks. As they're adopted in more sensitive areas, researchers have become aware of the vulnerabilities to adversarial attacks, specifically Projected Gradient Descent (PGD), and have offered various defenses. However, these defenses may not initially be optimal enough to be implemented in practice. Focusing on the widely used PyTorch framework, this paper demonstrates the ease and efficacy of leveraging CUDA to mitigate Python library call overhead and optimize memory traffic through the fusion of proposed defensive operations. The defense introduced a negligible 1.4% overhead in total inference latency, much less than the 68.3% overhead reported for the Kornia library.**

I. Introduction

Deep neural networks are highly vulnerable to adversarial attacks such as PGD, which introduce small, carefully crafted perturbations that cause misclassification while remaining difficult for humans to detect. This paper will extend the work presented in [1], which demonstrated that combining spatial smoothing with feature squeezing (via bit-depth reduction) substantially improves the robustness of models such as ResNet50 against adversarial examples.

Their implementation used the TensorFlow framework with the Adversarial Robustness Toolbox (ART), which, like many reference implementations, prioritizes correctness and flexibility over low-level performance optimization. To replicate their approach in our PyTorch environment, we use Kornia [6] implementations of spatial smoothing and feature squeezing, which similarly rely on unfused, multi-kernel tensor operations. This design introduces significant computational overhead and may be unsuitable for the high-throughput inference required in applications such as autonomous driving, even though the underlying security threat remains substantial.

To address this limitation, we present a fused GPU kernel exposed as a PyTorch custom operator. It integrates spatial smoothing and feature squeezing in a single pass to reduce inference latency. The kernel is intentionally simple and straightforward to implement, requiring only modest CUDA programming effort while still providing substantial performance benefits.

Availability: The source code for this project is publicly available at https://github.com/johnny-tavares/Optimizing-Adversarial-Defenses-With-CUDA

II. Background

A. Spatial Smoothing

As illustrated in Figure 1, Spatial Smoothing uses a sliding window across the entire image to filter out high frequency noise. This window typically contains the pixels around the target pixel, and uses them to generate a new pixel value as illustrated in Figure 2. The window can start at a size of 3×3 pixels, and the smoothing becomes more aggressive as you increase the window's size.

Regarding the calculation of the new pixel value, two popular options exist. Mean blur and median blur take the window's pixels, and choose the mean or median as the new pixel value respectively. Median blur will be more expensive to compute as you need to store each value and then sort, which turns out to be significant later on.

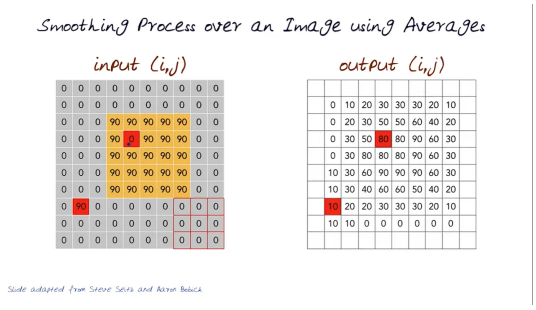

Fig. 1. The effect of smoothing on removal of outliers. (Reprinted from [4]).

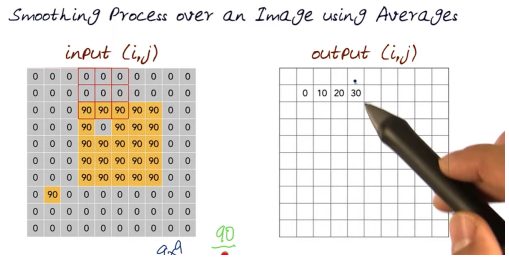

Fig. 2. Two-dimensional image smoothing example using a 3×3 average filter. (Reprinted from [4]).

B. Feature Squeezing

Bit-depth reduction reduces the color bit-depth of the image (e.g., from 8-bit to 4-bit). Figure 3 illustrates the results of feature squeezing, where colors are noticeably more "blocky". This is commonly implemented by quantizing the color space,

where each color channel's 2^n possible values (e.g., $2^8 = 256$ for 8-bit) are mapped down to a smaller set, effectively removing the least significant bits (LSBs) of the pixel values.

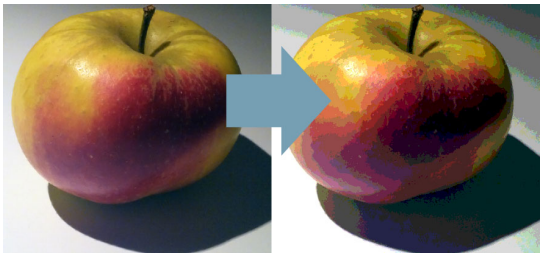

Fig. 3. The effects of reducing the color bit-depth of an image, demonstrating the quantization effect on pixel values. (Reprinted from [5]).

Similarly to Spatial Smoothing, by eliminating these small variances in the LSBs, bit-depth reduction destroys the carefully crafted gradient signal that the adversarial example relies upon to misclassify the model. This "squeezes" out small perturbation values, effectively sanitizing the input. With both combined, the output is demonstrated in Figure 4.

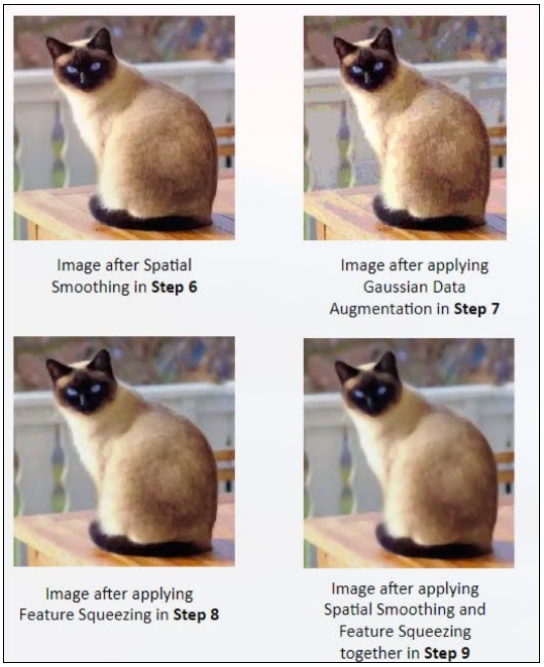

Fig. 4. Image changes throughout Spatial Smoothing and Feature Squeezing (Reprinted from [1]).

C. Projected Gradient Descent Reasoning

Projected Gradient Descent (PGD) is widely regarded as the "universal" first-order adversary, as noted in [3]. It performs an iterative sequence of gradient-based updates, each step pushing the input toward a misclassification while a projection operator keeps the perturbation within a specified ℓ_p-bounded region. This process makes PGD substantially stronger than single-step attacks such as FGSM, often producing perturbations that approach the worst-case adversarial examples permitted by the threat model.

Given its reputation as the strongest and most reliable first-order attack in the literature, evaluating our CUDA-based defense under PGD offers a meaningful and conservative stress test. Demonstrating robustness here suggests that the defense is likely to remain effective against weaker or less optimized adversaries as well.

D. Existing Implementations (Kornia)

The Kornia library is an open-source, differentiable computer vision library built specifically for PyTorch. Taking a deeper look at Kornia's median blur implementation, it currently carries out several independent GPU operations:

1) Constructing a binary kernel in Python and transferring it to the device,
2) Applying a full `conv2d` operation to extract local windows from the input, and
3) Computing the median via a reduction over the expanded window dimension.

Each stage triggers a separate kernel launch, resulting in additional global memory traffic, synchronization overhead, and loss of memory locality. A fused CUDA kernel would avoid these intermediate steps, keeping the computation local to the thread block and substantially reducing latency.

Another significant inefficiency arises from the creation of the sliding-window tensor. After the `conv2d` operation, the output is reshaped into a tensor of shape (B, C, K^2, H, W), effectively expanding the input by a factor of K^2. For a 3×3 kernel, this corresponds to a ninefold increase in data before the median is computed. Constructing and storing this intermediate tensor places a heavy burden on memory bandwidth, which is typically the dominant cost in filtering operations. In contrast, a specialized CUDA kernel would compute the median directly from registers or shared memory, avoiding the need to materialize the full window tensor in global memory.

III. Replication and Benchmarking Setup

A. Experimental Setup

- **Hardware:** Single Nvidia RTX A5000.
- **Model:** ResNet50V2.
- **Image Size:** 224×224
- **Blur:** Median blur is chosen as it's the more effective choice against adversarial attacks [2].
- **Edge Handling:** We use coordinate clamping rather than reflection padding in our Spatial Smoothing, as it's faster to implement in raw CUDA and avoids allocating larger padded images in memory.
- **Bit Depth:** 4-bit depth reduction is a common choice, although other options are certainly possible.

B. Benchmarking Methodology

To rigorously evaluate performance across all defense implementations (Defenseless, Kornia, and Custom Kernel), a

strict timing protocol was implemented to account for the asynchronous nature of GPU execution:

- **Warmup Phase:** A warmup loop of 10 iterations ensures the GPU caches are hot and the device is in a steady state.
- **Synchronization:** Calls to `torch.cuda.synchronize()` are placed immediately before the start timer and after the workload. This ensures the timer measures the actual GPU execution time, not just the CPU launch overhead.
- **Garbage Collection:** Python's garbage collector (`gc.disable()`) is disabled during the timing loop to prevent random CPU pauses from skewing the results, but this is mainly precautionary.
- **Metric:** We use Python's "time.perf_counter()" to benchmark the combined latency of applying the defense and performing model inference.

Listing 1. Accurate GPU timing implementation.

```
# Synchronize BEFORE start time ensures
# previous GPU work is done
torch.cuda.synchronize()
start = time.perf_counter()

# The workload
defended = apply_defense(batch,
                         defense_type)
with torch.inference_mode():
    output = model(defended)

# Synchronize AFTER end time ensures
# GPU work is actually done
torch.cuda.synchronize()
end = time.perf_counter()
```

C. Performance and Robustness Without Defense

For the defenseless case, the classification results were as follows:

- **Confidence:** The clean image was classified with an initial confidence of `87.48%`.
- **Average Inference Time:** The average inference latency was measured at `2.2089 ms`.
- **Standard Deviation:** The standard deviation for the inference time was `0.0172 ms`.

Applying the adversarial example to this baseline model successfully **destroyed its robustness**. The model misclassified the image with a final prediction of `paper towel` and a destructive confidence of `100%`.

D. Performance and Robustness With Kornia

For the baseline defense mechanism, the initial performance metrics show a predictable overhead compared to the undefended model:

- **Clean Confidence:** The model classified the clean image with an initial confidence of `84.92%` (a slight drop from the undefended baseline).
- **Average Inference Time (Overhead):** The average inference latency was measured at `3.7239 ms`.
- **Standard Deviation:** The standard deviation for the inference time was significantly higher at `0.1393 ms`.

When the adversarial example was applied to the model with this baseline defense, the robustness was significantly improved, as measured by the post-attack confidence:

- **Post-Attack Average Correct Confidence:** The average confidence for the correct class after the attack was `70.11%`.

This result demonstrates that the defense mechanism successfully maintained a high level of classification confidence, contrasting sharply with the 100% confidence misclassification observed in the undefended baseline.

IV. System Design: Fused Spatial Smoothing + Feature Squeezing Kernel

A. Kernel Design Overview

The core of the optimization strategy is a custom C++ CUDA kernel designed to fuse multiple operations into a single efficient pass. The kernel assigns each thread the job of calculating the new value for a single pixel. This process involves two distinct stages executed sequentially within the same thread:

1) **In-Register Median Sorting:** Unlike standard library implementations that rely on separate convolution and reduction steps, this kernel implements a sorting algorithm directly. For a given pixel, the thread retrieves the 3×3 neighborhood window, stores the values in a local array, and sorts them to identify the median. This minimizes global memory usage by keeping the operation largely within the thread's registers.
2) **Fused Bit-Depth Reduction:** Immediately after the median is found, the kernel applies feature squeezing. It uses `floorf` operations to quantize the pixel value:

$$\text{val} = \text{floorf}(\text{median} \times \text{levels})/\text{levels} \tag{1}$$

By performing this immediately after sorting, the kernel avoids writing intermediate results back to global memory, preserving memory locality.

Note that another reason median blur was chosen was to show that the CUDA kernel can still efficiently execute the more computationally expensive sorting option compared to simple averaging.

B. Implementation Details

The implementation leverages PyTorch's `load_inline` utility to integrate the custom CUDA kernel into the Python-based pipeline.

1) Runtime Compilation: The CUDA source code is defined as a C++ string within the Python script. The `load_inline` function compiles this string at runtime, creating a callable PyTorch extension. This allows the entire defense logic to remain self-contained while taking advantage of the raw performance of compiled C++.

Listing 2. Compiling the CUDA kernel at runtime.

```
my_ext = load_inline(
    name='defense_ext_v1',
    cpp_sources=cpp_source,
    cuda_sources=cuda_source,
    functions=['launch_fuse'],
    verbose=False
)
```

2) Kernel Launch Configuration: For the benchmarking experiments, the kernel is launched with a fixed configuration to ensure consistent occupancy. We utilize a thread block dimension of 16×16 (256 threads per block). The grid dimension is explicitly set to $14 \times 14 \times 3$ to cover the input spatial dimensions and processing planes.

V. Evaluation

A. Latency Comparison

To evaluate the computational efficiency of the proposed defense, we measured the processing time of the Custom Kernel against a standard library implementation (Kornia) and a baseline of no defense. As shown in Figure 5, the Kornia Median Blur introduced significant overhead, increasing processing time by 68.3%.

In contrast, our Custom Kernel implementation demonstrated superior performance, executing in 2.24 ms. This represents an overhead of only 1.4% compared to the "No Defense" baseline, making it a highly efficient solution for real-time applications.

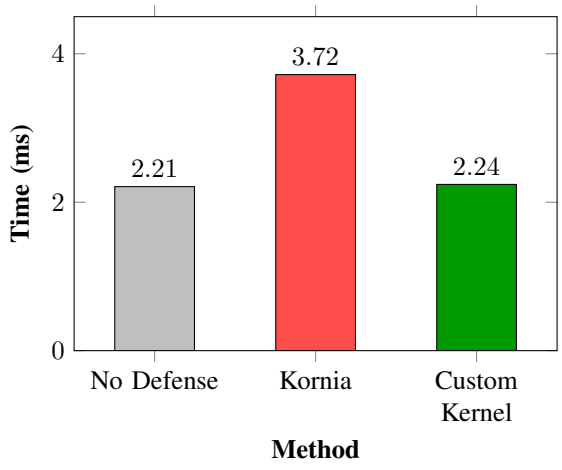

Fig. 5. Latency comparison between standard Kornia implementation and the optimized Custom Kernel.

B. Robustness Comparison

We evaluated the effectiveness of the defenses against adversarial examples. Table I summarizes the prediction results, confidence scores, and success status for each strategy.

Without defense, the model failed completely, misclassifying the input as "Paper Towel / Remote" with 100% confidence. Both the Baseline (Kornia) and the Custom Kernel successfully recovered the true class ("Siamese Cat"). While the Custom Kernel resulted in a slightly lower confidence score (∼66.17%) compared to the Kornia Baseline (∼70.11%), it maintained the correct classification while offering the significant latency improvements noted in the previous section.

Defense Strategy	Prediction	Confidence	Status
No Defense	Paper Towel / Remote	100.00%	FAILED
Baseline (Kornia)	Siamese Cat	∼70.11%	RECOVERED
Custom Kernel	**Siamese Cat**	**∼66.17%**	**RECOVERED**

TABLE I
ROBUSTNESS COMPARISON SHOWING SUCCESSFUL CLASSIFICATION RECOVERY BY BOTH DEFENSE METHODS.

VI. Discussion

A. Extending to Other Defenses

While this work focused on replicating the spatial smoothing and feature squeezing defense proposed in [1], the core contribution—reducing memory bandwidth pressure through kernel fusion—extends naturally to a wide range of preprocessing defenses. Many gradient-masking or input-transformation techniques share similar bottlenecks, including repeated kernel launches, unnecessary global memory traffic, and Python-level overhead. Because these inefficiencies are structural rather than algorithm-specific, the same CUDA optimization strategy could be applied to defenses such as JPEG compression or randomized preprocessing layers.

B. Limitations

Although the Custom Kernel demonstrates substantial speedups, several limitations must be considered when interpreting the results. One key factor is the impact of the underlying machine learning libraries. The original work by Muthalagu et al. used TensorFlow and the Adversarial Robustness Toolbox (ART), whereas this replication was conducted in PyTorch. This mismatch is not ideal for direct comparison, as each framework differs in memory allocation behavior, operator implementations, and scheduling.

C. Timing Fairness and Kornia Overhead

When comparing the Custom Kernel to Kornia, it is important to consider what each tool is optimized for. Kornia is a fully differentiable CV library designed for training pipelines, meaning it must maintain autograd traces and relies on generic PyTorch primitives that prioritize flexibility over minimal memory movement. In contrast, the Custom Kernel is specialized purely for inference and removes all unnecessary overhead. Thus, the performance difference reflects a trade-off between the flexibility of a general-purpose differentiable library and the efficiency of a purpose-built CUDA kernel, rather than a deficiency in Kornia's implementation.

D. CUDA Efficiency

At the low level, the Custom Kernel presented here is not theoretically optimal. The implementation favors clarity and simplicity, relying on per-thread register sorting. A more advanced version could use shared-memory tiling to reduce redundant global memory reads for overlapping windows,

or using a more GPU-Optimized sorting algorithm. As a result, the measured 1.4% overhead should be viewed as a conservative upper bound; tighter optimization could push the overhead toward zero.

VII. Conclusion

This paper presented a CUDA-optimized implementation of a spatial smoothing and feature squeezing defense for PyTorch. By fusing operations into a single kernel, we reduced inference overhead from 68.3% to a negligible 1.4%, making the defense viable for real-time systems. While high-level libraries like Kornia provide essential flexibility for research and training, this work demonstrates that custom kernel fusion is a necessary step for deploying adversarial defenses in production environments where latency is critical. We successfully demonstrated that the defensive principles established in prior TensorFlow research can be efficiently adapted to PyTorch, providing a robust and high-speed protection against PGD attacks.

References

[1] R. Muthalagu, J. Malik, and P. M. Pawar, "Detection and prevention of evasion attacks on machine learning models," Expert Systems with Applications, vol. 266, Art. no. 126044, 2025. doi: 10.1016/j.eswa.2024.126044.

[2] W. Xu, D. Evans, and Y. Qi, "Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks," in Proceedings 2018 Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 2018. doi: 10.14722/ndss.2018.23198.

[3] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," arXiv preprint arXiv:1706.06083, 2019.

[4] Udacity, "Smoothing Process Over an Image Using Average," YouTube, Feb. 23, 2015. Accessed: Dec. 8, 2025. [Online]. Available: http://www.youtube.com/watch?v=ZoaEDbivmOE.

[5] Blender Artists Community, "Reducing the number of colors (color depth)," Apr. 8, 2013. [Online]. Available: https://blenderartists.org/t/reducing-the-number-of-colors-color-depth/571154. [Accessed: Dec. 8, 2025].

[6] E. Riba, D. Mishkin, D. Ponsa, E. Rublee, and G. Bradski, "Kornia: an Open Source Differentiable Computer Vision Library for PyTorch," in IEEE Winter Conference on Applications of Computer Vision (WACV), 2020, pp. 3674–3683.