A Deterministic Approach to Inferencing on Real-Time Safety-Critical Systems

When topics of safety and artificial intelligence (AI) arise, the focus often rests on concerns around proving the intent of a neural network (NN). Although proving the intent of a network is an important problem to solve, it is not the only problem when it comes to the application of NN in real-time safety-critical systems (RTSCS). A key problem in this domain is proving that the platform execution is deterministic, meaning that the platform execution not only needs to provide results in a consistent and reliable fashion, but also needs to use a well-defined amount of memory. Existing AI platforms are built on technologies like Python that make use of techniques like runtime garbage collection. Time has shown that these existing platforms have not been a great fit for RTSCS. This paper explores how one would deterministically inference a NN, and how a standard like Khronos' OpenVX™ provides a platform for AI that will be applicable for real-time safety-critical systems.

INTRODUCTION

As neural networks become more commonplace in modern applications, concerns begin to arise when they become part of systems that we typically expect to be safe. We all step into cars, planes, and trains with the expectation that the engineers who built these vehicles built them with safety in mind. Several authors[1,2,3] have done significant research on how to provide mechanisms to prove that a given neural network is accurate and reliable, but there has not been an equal amount of research on how to provide a platform on which a given neural network can execute deterministically and safely. If we want to take advantage of neural networks in these safe systems, we also need to determine methods to deploy the neural networks in a safe manner.

Popular neural networks in literature such as the MobileNets[4] are built using software libraries like Tensorflow, Pytorch, and Caffe. These libraries provide a tremendous ability to developers and researchers to easily build and iterate their work with neural networks. These libraries, however, are not designed with real time safety-critical systems in mind. None of these libraries provide certification artifacts for any of the major safety-critical software standards like DO-178C, ISO 26262, and IEC 61508. In fact, many of these libraries rely on software platforms such as Python, which would provide a significant barrier for certification in these standards.

When building deterministic software for safety-critical applications, one must ensure that the software is deterministic in both space and time. This means that for a given operation the software needs to utilize a deterministic amount of memory and it needs to complete the operation in a deterministic amount of time. To properly explore how neural networks utilize time and space, we need to explore these topics in the context of specific implementations on specific hardware architectures. We will facilitate these specifics by analyzing the use of a rectified linear unit (ReLU) activation function on an artificial neural network, being inferenced on a GPU. In this paper we will explore how one could implement a deterministic inferencing engine for neural networks, as well as a path to migrate neural networks developed on unsafe platforms to a more certifiable solution.

DETERMINISTIC INFERENCING ENGINE

To better understand the space and time requirements of inferencing a neural network, first we must understand the operation of a neural network and how that directly applies to the hardware on which we are trying to execute the operations. The concept of a neural network is illustrated in Figure 1, where we can see that each node is represented by $f_{i,j}$ where i represents the layer in the network and j represents the j'th node in that layer.

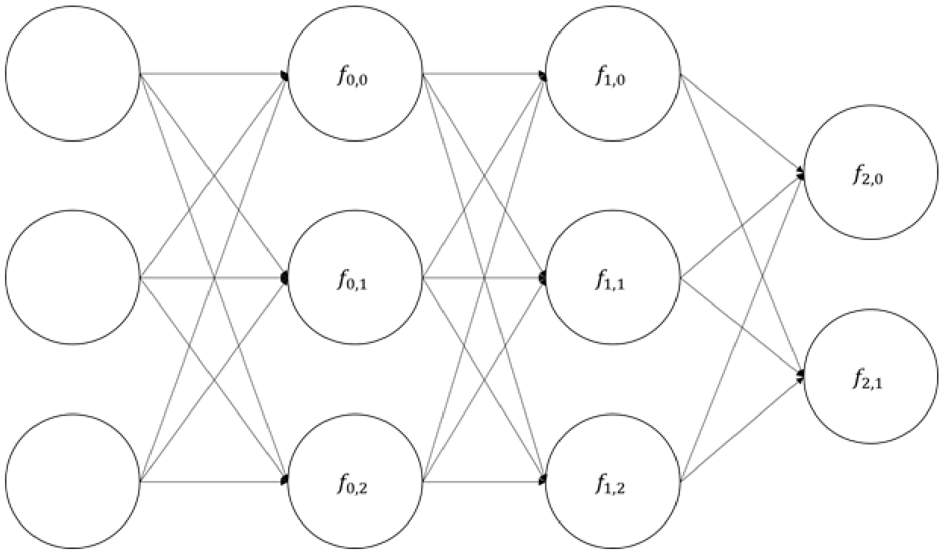

Figure 1: Example of a Neural Network

To inference with a neural network, we need to apply a very basic operation to each node in the network. This operation consists of taking a weighted sum of all the inputs to a node in the network and then applying an activation function σ to this weight sum. We can represent that operation with the equation in Figure 2.

$$f_{i,j} = \sigma\left(b_{i,j} + \sum_{k=0}^{n} w_{i,j,k} a_{i,j,k}\right)$$

Figure 2: Neural Network Evaluation Equation

Where $f_{i,j}$ represents the result of the operation for j'th node on the i'th layer in the network (which we will refer to as the activation of that node):

- $b_{i,j}$ represents the bias for j'th node on the i'th layer in the network;
- n represents the number of connections to the i'th node;
- $w_{i,j,k}$ represents the weight the k'th connection to the j'th node on the i'th layer in the network, and;
- $a_{i,j,k}$ represents the activation of the k'th connection to the j'th node on the i'th layer.

There is another approach we can take to represent an entire layer of the neural network as a series of linear equations that can be represented by matrix and vector operations, which would be written as shown in the equation in Figure 3.

$$f_i = \sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}\right)$$

Figure 3: Neural Network Linear Equation

Activation function σ is applied to each component of the resulting vector, and f_i is a vector that represents the activation value of each node in the i'th layer of the network. To execute this neural network, we simply need to compute f_i for every layer in the network, and the last layer of the network will represent our result.

With the understanding of how neural networks operate we can begin to investigate their usage of spatial resources. Using the explanation from above, we can see that all parameters in a neural network can be represented by either a fixed-size matrix or a fixed-sized vector. The fact that these primitives are fixed in size gives us a very important quality we can exploit for deterministic memory usage. As these matrix and vector primitives are fixed in their size and need to exist for the lifetime of the neural network, we can take advantage of a safety-critical memory manager to statically allocate all the required resources for this network. The use of static allocated memory with known sizes means we can deterministically compute the required amount of memory to represent and operate on our neural network.

We also need to ensure the execution of the neural network is deterministic in the time domain. For a given node there are multiplication and addition operations represented by the matrix vector operations, and there is also the activation function, which we will explore with a few examples.

First, it is trivial to see that the mathematical matrix and vector operations are deterministic. Although the mathematical operations are deterministic, we will also need to investigate how they are executed on GPU hardware to determine if the operation is deterministic. In this section, we will explore the determinism of the mathematics. To show that we can determine the mathematical operations, there are n multiplications to multiply the weight against the input activation, there are $n-1$ additions to sum all the weight-input multiplications, and one more addition to add the bias to the result. Additions and multiplications are basic operations, and because n is a fixed number based on the input, we can deterministically compute the number of operations we will do and amount of time it will take to complete all those operations.

For the activation function, more analysis is required to prove that the operation is deterministic. A common activation function used in many neural networks is the "rectified linear unit" or ReLU[5]. This is typically represented by the piecewise mathematical function in Figure 4.

$$\sigma(x) = \begin{cases} 0 \; if \; x \leq 0 \\ x \; if \; x > 0 \end{cases}$$

Figure 4: ReLU Equation

A naïve implementation of this function would be to translate the conditions on the variable x as branches in code. This is represented in pseudo code in Figure 5.

```
function RELU(x):
    if(x <= 0):
        return 0
    else:
        return x
```

Figure 5: NaÏve ReLU Implementation

At first, this simple implementation of the function appears quite basic and likely deterministic. When we analyze this function in the context of executing on a GPU, the branching behavior makes it difficult to prove the determinism of the execution. On most modern GPUs, the hardware will try to execute many instances of the same operation in parallel[6]. Most implementations achieve this by having an execution unit complete the exact same operations with different data, sharing the hardware that is used to decode the instructions and interpret the instructions, while having their own copy of data for the specific instance of the operation they are doing. This means that if the data—the variable x in our case—causes one instance to need to branch, every single instance in that execution unit now needs to branch, as the logic used to decode instructions is shared. Because of this hardware behavior, it becomes difficult to reason about the execution time of a program running on a GPU that makes use of branches that rely on dynamic input. Another more robust way we can implement this operation is presented in Figure 6.

```
function RELU(x):
    ge_than_zero = (x > 0)
    return ge_than_zero * x
```

Figure 6: More Robust ReLU Implementation

In this implementation we represent the branching logic as a multiplication by a Boolean value. The Boolean value can be either 0 or 1, so when the comparison result is false, we get 0 multiplied by x which will also be 0. When the comparison result is true, we get 1 multiplied by x which simplifies to just x. This exactly represents the ReLU operation without the use of branching, eliminating the portion of the algorithm that was hard to reason. With this implementation we will always execute the same number of operations regardless if one instance in an execution unit is taking a different "branch" than the other instances in the execution unit. With the more GPU-friendly implementation, there is one comparison and one multiplication operation. Therefore, if a network comprises k, we will need to perform k comparisons and k multiplications, showing that we can deterministically calculate the number of operations we would need to do for the activation function, which would allow us to determine the time it would take to complete these operations for a given network.

As a caveat to above information, we will briefly explore how a typical GPU would implement these operations, and why it is critical to investigate the actual GPU ISA that implements these operations. For example, to know that the pseudo code in Figure 6 is deterministic we would need to confirm that the GPU ISA has an instruction to do the "less than" operation without relying on branching. We would also need to confirm that the compiler we are using converts that "less than" operation into machine code does not use a branch and instead uses the "less than" instruction supported by the ISA. If we investigate the AMD GCN ISA used in Polaris GPUs[7] we can see that it supports a "V_CMP" instruction, which can be used for the "less than" comparison without relying on branch operations. If we are working to certify the program running on the GPU, we would also want to verify the actual instructions generated by the shader compiler we are using. Usually, a safety-critical driver vendor would generate the ISA into a human readable text file so that a developer would be able to compare the generated machine code to the source shader program and verify that the instructions the GPU would execute are deterministic.

PLATFORM FOR NEURAL NETWORKS

The last topic to explore is the platform that enables the execution of the neural network. What is meant by the platform, and what is the supporting code running on a CPU that allows the neural network to run on an acceleration device like the GPU? As mentioned during the introduction, popular frameworks like Tensorflow or Pytorch are built upon languages like Python. These tools are built to enable rapid development and iteration and do not concern themselves with the requirements for safety-critical systems. Python, for example, is a dynamic programming language that depends heavily on dynamic memory allocation and automatic memory management through a garbage collector. Some of these platforms like Tensorflow also provide C/C++ interfaces to eliminate the dependency on python but still do not provide any certification of safety.

Although these tools are not designed with safety in mind, they provide an enormous amount of support when it comes to developing and training neural networks in a lab environment. For safety-critical applications, the ideal situation would be to continue developing neural networks with these robust tools. An application developer would only need to worry about migrating to a safety-critical tool for the deployment of the neural network on the safety-critical system. The open standards for "OpenVX"[8] & "NNEF"[9] from the Khronos organization provide an approach that would allow developers to do exactly that.

The Neural Network Exchange Format (NNEF) standard provides an open and common exchange format where a neural network developed in one tool (such as Tensorflow) can be exported into the NNEF format and then imported into another tool. This allows us to leverage the significant amount of effort that has been invested in the popular neural network development platforms.

The OpenVX standard from the Khronos Group provides standardized APIs to accelerate computer vision applications through traditional computer vision methods and through neural networks. For our specific use case of neural networks and safety-critical applications, the standard provides three feature sets that are of significant interest. First, there is the "Neural Network" feature set that provides functions to build and execute neural networks. Second, there is the "NNEF Import" feature set that provides the ability to import NNEF formatted neural network models into an OpenVX application. Finally, there is the "Safety-Critical Deployment" feature set, which provides a subset of features in OpenVX meant to simplify the certification process of an OpenVX application. These specific feature sets provide a compelling reason to use OpenVX in the safety-critical application of neural networks.

The neural network feature set and NNEF import feature set provide the bulk of the requirements for a safety-critical deployment platform for neural networks. They also allow safety-critical implementations to exist more easily. Specifically, the NNEF requires the use of the “vx_khr_export_and_import”[10] extension for OpenVX. This extension allows an implementation vendor to define a vendor-specific format of an OpenVX object such a neural network. This is useful for a safety-critical implementation as it would allow the conversion of an abstract neural network model to a GPU accessible version to be done “offline”. This would mean that any code responsible for compiling and optimizing the model would not need to execute on the safety-critical system and instead could execute on a development machine and produce a binary that could be loaded by a vendor specific implementation. This would reduce the amount of complex code in a safety-critical OpenVX implementation, simplifying the certification process of that implementation.

CONCLUSION

When developing a GPU accelerated implementation of neural networks it is important to consider how the hardware will execute each of the individual operations. In this paper we explored how a modern GPU would execute branching operations and how that could affect the determinism of a given implementation. Other hardware devices that are being used to accelerate neural networks like Tensor Processing Units (TPUs) and Neural Processing Units (NPUs) likely have their own hardware idiosyncrasies that developers of safety-critical software will need to consider, ensuring that their implementations are deterministic. Additionally, it is important to consider that a neural network does not simply exists on its own; there is a lot of supporting software to enable just the deployment of a neural network. In a safety-critical system, all components need to be designed with determinism in mind.

REFERENCES

1. R. R. Zakrzewski, "Verification of a trained neural network accuracy," IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222), Washington, DC, USA, 2001, pp. 1657-1662 vol.3
2. Grégoire Montavon, et al. "Methods for interpreting and understanding deep neural networks". Digital Signal Processing 73, 2018, pp. 1 - 15.
3. C. Cheng *et al.*, "Neural networks for safety-critical applications — Challenges, experiments and perspectives," *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, 2018, pp. 1005-1006
4. Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, & Hartwig Adam, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.”, 2017.
5. Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks.", In Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, 2011, pp. 315–323, JMLR Workshop and Conference Proceedings.
6. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
7. AMD. "AMD GCN3 ISA." 2016. http://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf
8. https://www.khronos.org/openvx/
9. https://www.khronos.org/nnef
10. https://www.khronos.org/registry/OpenVX/extensions/vx_khr_import_kernel/1.0/vx_khr_import_kernel_1_0.html

AUTHOR

Lucas Fryzek
Field Application Engineer

Lucas Fryzek has been a software developer with CoreAVI for 5 years and has recently transitioned over to the role of Field Application Engineer (FAE). Lucas has extensive hands-on experience in developing embedded software systems deployable in DO-178C safety critical environments and has worked in tandem with many of CoreAVI's largest leading customers to deliver safety critical solutions, including those specializing in the division of graphics workloads across multiple CPU cores. Lucas' experience with OpenGL SC and Vulkan APIs allow him to provide expert technical support and guidance both to CoreAVI's customers and internal team.