Validating Expression and Execution in Neural Networks: The Future of Safe Autonomy

INTRODUCTION

In today's increasingly automated world, decisions are made by algorithms interpreting the environment in which they operate. This is the plight of a car driving down a busy highway without human intervention. The car must understand its environment. It must be able to perceive road signs, lane markings, humans, animals, and other vehicles in its surroundings. The artificial intelligence system at the helm must decide how to steer the vehicle, when to accelerate, and when to apply the brakes. In the process of making these decisions there is a string of algorithms—often artificial neural networks—analyzing sensor data. Too often, when we discuss the role of algorithms in areas involving human-machine interactions, the focus of the conversation is placed on the safety of the algorithm's predictive capabilities. That is, how likely is the algorithm to make a mistake. Is it possible for the car to confuse a bridge overhead with an obstacle to be avoided? This 'facet of analysis' ensures the safety of the algorithm's expression. An algorithm's expression is its prediction or output. For algorithms that scan images and output a classification of the objects in the images—for example, truck vs. bicycle–the output of the algorithm is a value that indicates if the object it has identified is a car, a truck, pedestrian, or any of the other possible classes of objects it was trained to identify.

Instinctively, when we think of the safety of an autonomous system the questions that we ask are: How can we guarantee that the neural networks analyzing the input data coming from the sensors are making the right decisions? How do we know that when a neural network detects a human, there is indeed a human there, and can we know that it has not missed or misidentified other objects? Consider the implications of a self-driving car mistaking a shiny vehicle for the sky or mistaking a pedestrian at an intersection for a light post. Similarly, consider a mistake in a neural network decision concerning how to steer the car to keep to its lane. Clearly, errors involving humans and machines can be costly not just financially, but especially in human lives. Considering the stakes, it is no wonder that our attention immediately goes to the predictions of a neural network. But beware! The expression of the neural network is not the only problem that must be solved to prove that a system is safe. We must also show that the whole system behaves deterministically. We must document undefined behavior and prove the software stack has been rigorously tested. Finally, execution time for each piece of software must be reliably predictable.

PROBLEM STATEMENT

Safe AI necessitates validating the expression of the decision-making system and ensuring the deterministic execution of the same system. Too often we forget about execution safety and rely on simulation to prove an AI system safe. As we will argue, simulation can solve part of the problem, but it cannot guarantee deterministic execution.

ORGANIZATION OF THIS PAPER

Expression problem of neural networks
In this section we introduce the two aspects of AI algorithms that must be safe, expression and execution, and focus on the expression of neural networks.

Introduction to neural networks
In this section we describe how convolutional neural networks (CNN) work, by explaining the mathematical operations involved in convolutions. We also take a brief look at a common neural network architecture (MobilenetV2).

GPU acceleration is not free
In this section we discuss how a graphics processing unit (GPU) is used to accelerate the computations taking place inside neural networks. We show how an implementation chooses to use the GPU resources to accelerate computations can have a big impact on performance as well as determinism.

Deterministic execution with respect to results
In this section we discuss what it means for the results of an algorithm to be deterministic, and why execution can affect the determinism of certain mathematical calculations.

Vulkan SC as a conduit to deterministic general purpose GPU (GPGPU) software
In this section we discuss the importance of a safety-critical (SC) API in functional safety designs. We use Vulkan SC as an example of a safety-critical API that enables the implementation of deterministic software.

Predictable operations per second
In this section we introduce a new unit of performance measurement, Predictable Operations per Second, to convey the difference between raw system performance and relevant performance, considering that safety often means a tradeoff between performance and determinism.

Using simulation to prove the safety of execution
In this section we argue that simulation must not be used to prove that an algorithm executes deterministically. We base our argument on scalability, cost, and necessity; proving determinism through simulation is not scalable, is expensive, and is unnecessary considering there are well defined ways to achieve this.

A look to the future
In this section we close the article with a look to where autonomous systems, specifically self-driving cars, are moving to in the future. We propose that the time to set the foundations for safe, deterministic software based on functional safety design principles is now.

GLOSSARY OF TERMS

Deterministic execution means the worst-case execution time of a piece of software must be predictable.

Expert systems are artificial intelligent systems that emulate tasks traditionally performed by humans with expert knowledge in a specified domain.

Shaders are software programs that execute in a GPU.

Shader compilation is the process of converting (or compiling) a shader program from a high-level programming language such as GLSL, into GPU-specific instructions.

EXPRESSION PROBLEM OF NEURAL NETWORKS

Neural networks are not the only algorithms making decisions. Robust automation pipelines consist of a suite of algorithms working together to tackle different stages of problem-solving from initial input to the physical system taking an action in the environment. In this paper we will focus on neural networks for two reasons:

1) They are the current state-of-the-art solution for artificial perception, natural language processing, and most AI-related use cases.
2) The outputs of neural networks are difficult to validate.

When a neural network outputs a decision for a given input, it is impossible to trace the process that led from the input to the output and decode the rules that led to a given output. Classically, we think of software as a series of discreet steps, like a decision tree. A decision tree consists of a structure of “if” and “else” clauses branching out from the root of the tree. At each step we evaluate our progress against rules. If it is raining, then take an umbrella, else if it is sunny take a hat. Otherwise, no hat is necessary. With a decision tree we can walk the decision process back from the final decision to the rules that resulted in the decision. Unfortunately, with neural networks it is impossible to understand which condition caused the neural network to arrive at a given prediction. The decision of the neural network is encoded in the many millions of parameters making up the network connections, and those parameters were adjusted gradually during the training process so that there is no single explanation for why a given output was encountered. Instead, the explanation is that the training process and the training data is wholly responsible for the network’s decision. This is what we refer to as the expression problem of neural networks. We can interpret the output of a neural network, but we cannot explain it. Explaining the decisions of neural networks is at the forefront of current Computer Science research. In the meantime, to still employ these algorithms in use cases where risk must be quantified and understood, companies resort to simulation.

Simulation plays an important role in increasing the confidence level for a neural network by testing the accuracy of a neural network’s response using millions of test data samples. Considering the complexity of urban environments where autonomous cars must operate, the data set of events required to train a neural network using simulated events must be rich. The training data set must contain instances of cars traveling in clear conditions, in different weather conditions. There must be instances of the car driving under bridges, over bridges; instances of other cars merging onto the test car’s lane; instances of bicycles merging, and pedestrians crossing at an intersection, or j-walking. In short, there must be millions of diverse samples of simulated events to train the neural network, and then there must be a similarly rich data set of sample events to test the neural network. It might be argued that if the data set is rich enough, simulation can be used to show the output of the neural network is empirically safe enough to use in critical domains. This is often argued by showing a simulated car driving for millions of miles without an accident. Proving the expression of the neural network is safe by use of simulation is not ideal. Clearly, being able to explain the algorithms’ decisions, and debug failure cases by stepping through a decision process would be preferable. Unfortunately, the best tool that we have for making sense of data today is the neural network, and we have stated that at least for the moment, explaining their outputs is slightly out of reach of modern computer science. In this case simulation might be our only way to solve the expression problem. But, as I will argue in this series of papers, showing that the expression of a neural network is safe is only part of the problem. We must still prove that the execution of the neural network is safe by showing that it is deterministic. This part must not be done through simulation.

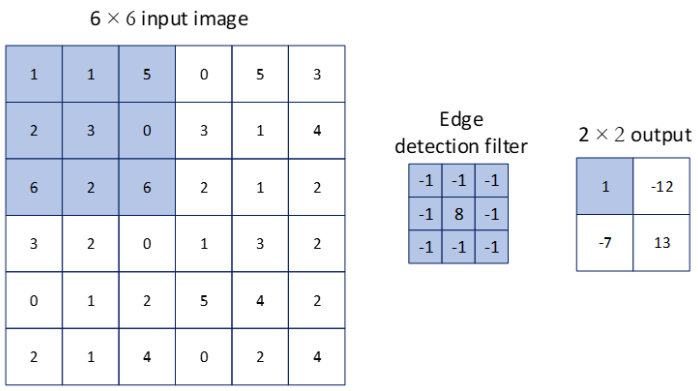
INTRODUCTION TO NEURAL NETWORKS

Besides the imposing name, an artificial neural network is simply a set of mathematical operations implemented in software using instructions. These instructions are executed by a CPU and an accelerator device, such as a GPU, in a manner no different from regular software. An autonomous system interacting in human operating environments in real-time must ensure that these instructions execute deterministically. The instructions must be validated and verified, and undefined behavior must be documented. These are the guiding principles of most certification standards for functional safety.

We will get to deterministic execution, its importance, and how we can achieve it in the following sections. But first, let's spend a few minutes describing what a neural network looks like, the operations taking place at each step, and the volume of operations that must be accelerated. To understand the implications that execution has on safety, it is important that we understand exactly what operations are involved in neural network execution. For our example we will use the MobilenetV2 architecture introduced by Google research in "MobileNetV2: Inverted Residuals and Linear Bottlenecks."

MobilenetV2 is a common architecture used as the backbone of many computer vision applications in embedded systems. Its popularity is the result of achieving performance comparable to many state-of-the-art architectures, at a fraction of the computation cost. MobilenetV2 consists of 18 convolutional layers, where each layer is made up of several convolution operations.

Let's recall a simple convolution to understand the types of operations taking place inside a convolutional neural network (CNN).



6 × 6 input image

1	1	5	0	5	3
2	3	0	3	1	4
6	2	6	2	1	2
3	2	0	1	3	2
0	1	2	5	4	2
2	1	4	0	2	4

Edge detection filter

-1	-1	-1
-1	8	-1
-1	-1	-1

2 × 2 output

1	-12
-7	13

Convolution operation:

Top left output value:
(-1 x 1) + (-1 x 1) + (-1 x 5) + (-1 x 2) + (8 x 3) + (-1 x 0) + (-1 x 6) + (-1 x 2) + (-1 x 6) = 1
Top right output value:
(-1 x 0) + (-1 x 5) + (-1 x 3) + (-1 x 3) + (8 x 1) + (-1 x 4) + (-1 x 2) + (-1 x 1) + (-1 x 2) = -12
Bottom left output value:
(-1 x 3) + (-1 x 2) + (-1 x 0) + (-1 x 0) + (8 x 1) + (-1 x 2) + (-1 x 2) + (-1 x 1) + (-1 x 4) = -7
Bottom right output value:
(-1 x 1) + (-1 x 3) + (-1 x 2) + (-1 x 5) + (8 x 4) + (-1 x 2) + (-1 x 0) + (-1 x 2) + (-1 x 4) = 13

Figure 1: Example of a convolution operation. A convolutional neural network is based on thousands of operations like these.

A convolution is an operation involving an input signal and a filter matrix. In computer vision the input signal is typically an image. In the illustrated example we have a 6x6 pixel image and we are using a 3x3 filter matrix to convolve the image. The operation involves overlaying the filter matrix over the input, as shown, and multiplying the value in each filter cell with the corresponding value coming from the image. The results of the multiplications are summed, and this becomes the value of the new pixel in the output image. When considering the work that a GPU has to do to process an input signal through a neural network, we typically consider the number of multiplications as an estimate for the complexity of the algorithm. Additions also affect performance, but multiplications are by far the most ‘expensive’ operations. In this simple example, see Figure 1, where we are processing a 6x6 input image by a single 3x3 filter, we must perform 27 multiplications. A convolutional neural network (CNN) consists of many convolution operations just like these.

In the case of MobilenetV2, the classic size of the input image is 224x224x3 pixels. That is 224x224 pixels per RGB channel. The first layer contains 32 filters. Each filter must process the entire input volume. That is, as opposed to our simple example where the input and filter consisted of a single channel (and the filter was a single 3x3 matrix), here the 32 filters are in effect 32 3x3x3 matrices. Notice that we add an extra dimension to the filter matrix because the input volume has three dimensions thanks to the color channels. Each filter matrix will process the input volume by performing a convolution operation over each channel (the input’s red channel is processed by the filter’s corresponding channel, and the same goes for the green and blue channels), and the results of the convolution for each channel are summed element-wise to provide a single channel output for this filter (see Figure 2).

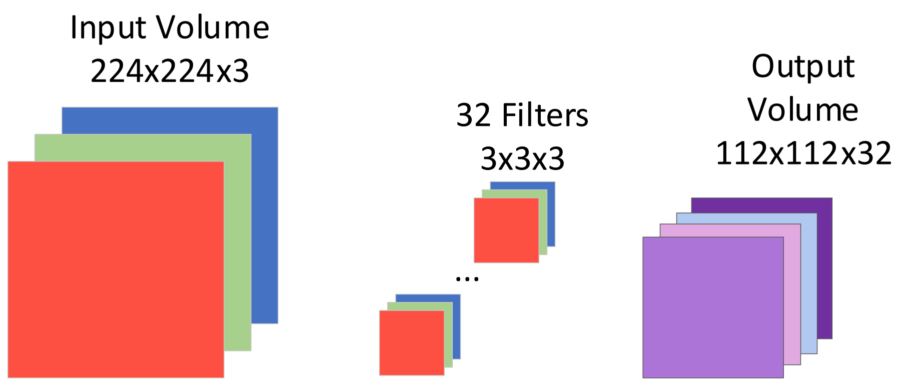


Figure 2: The first layer of the MobilenetV2 neural network consists of 32 3x3x3 filter matrices. That is one 3x3 filter matrix per channel in the input volume. The input volume consists of three (RGB) channels of a 224x224 pixel image.

We must perform these operations 32 times—once for each filter in the layer. This results in ten million multiplications just for the first layer of the neural network. The output of this layer is a volume of 112x112x32 feature maps. That is, 32 images of 112x112 pixels—one for each of the 32 filters.

The second layer of MobilenetV2 is an Inverted Residual block, which contains a series of convolution filters. This layer takes the output volume (112x112x32) of the previous layer as input, and performs a series of convolutions using its filter matrices. The operations in layer two result in 26 million multiplications, and the output is a volume of 112x112x16 feature maps (see Figure 3).

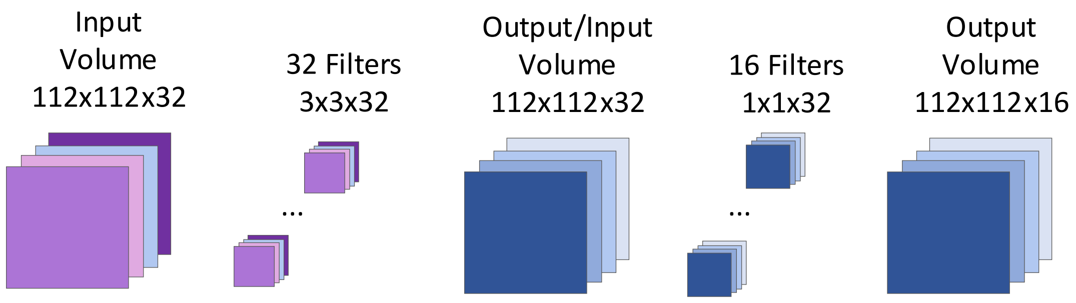


Figure 3: In the second layer of the MobilenetV2 architecture we have several convolutional blocks. The first block produces an output volume of 32 112x112 pixel images. This volume becomes the input into the next block, which produces an output volume for the layer that is 16 112x112 pixel images.

There are 18 such layers in Mobilenetv2. If we continue to account for all multiplications that take place as a result of the convolution operations processing a single image, we will see that this neural network requires approximately 300 million multiplications to process a 224x224x3 input image.

Although the Mobilenet architecture is known as an efficient class of neural networks, 300 million multiplications per inference is no easy task for a CPU to perform. This is why we use GPU accelerators.

GPU ACCELERATION DOES NOT COME FOR FREE

Let’s consider how a GPU might be used to accelerate the execution of a neural network. The GPU contains thousands of computational cores, but this alone does not guarantee that software will execute fast. The algorithm must be designed in such a way that computations can be parallelized to take advantage of the large array of cores a GPU provides.

Let’s illustrate this with an example:
Suppose that we write a very complex shader that iterates over each 3x3xc block in the input volume of a layer. In this example a single shader instance executing in a single GPU core can perform all the calculations necessary to inference the neural network. But doing this means that we are using only a single core of the GPU, and all processing is serialized. We might as well be using the CPU.

A better way to use the GPU might be to realize that you have to perform a convolution operation for each 3x3xc block of the input volume, so why not write a simpler shader that performs a convolution over a 3x3xc signal? Then, execute this shader across multiple GPU cores, where each core samples a different 3x3xc block of the input volume (see Figure 4).

In this approach we are parallelizing the processing of the input volume by a filter. But recall that a layer has multiple filters. In this algorithm, presumably the processing of the filters is serialized. We parallelize the processing of the input volume by executing multiple instances of the same filter across all GPU cores, but the next filter in the layer must wait until the GPU’s compute units are done processing to move on to the next filter.

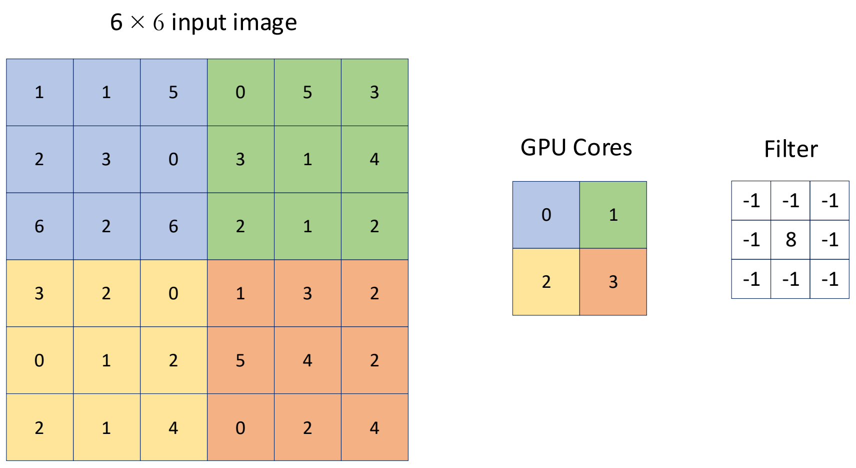


Figure 4: An input signal may be partitioned into blocks, where the same convolution filter processes different blocks in parallel through multiple cores of the GPU.

A different algorithm might suggest parallelizing the usage of all filters across a layer. For example, notice that the same 3x3xc block in the input volume will be processed by each filter in the layer. In some cases, it might be beneficial to write a shader that convolves the same block of the input volume for each shader instance, but in each instance (running in its own GPU core) the shader is convolving a different filter with the input volume (see Figure 5). This approach parallelizes the use of filters within a layer, but serializes the processing of the 3x3xc blocks across the input volume.

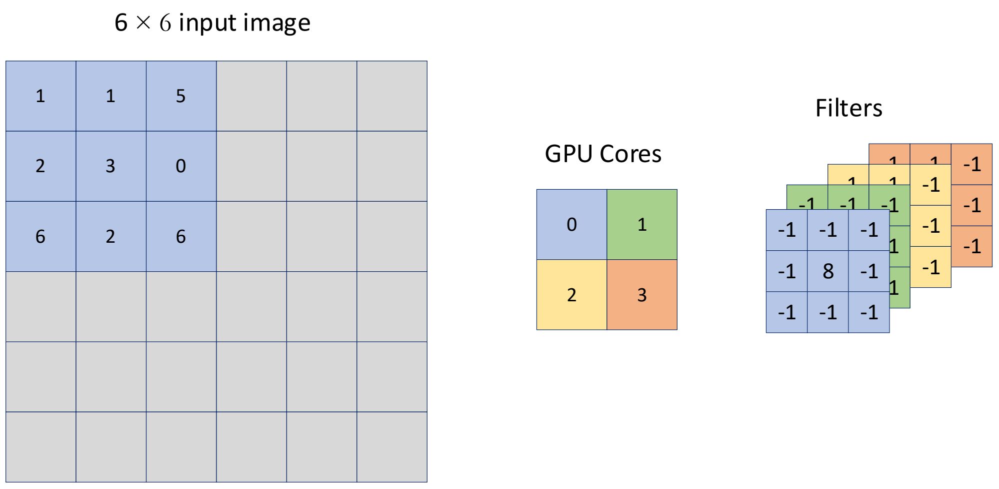


Figure 5: The same block of the input image is processed by multiple filters in parallel through multiple GPU cores.

The approach that we choose to extract as much performance out of a GPU as possible depends on many variables including the GPU architecture, the specific algorithm we are trying to optimize, the overall system load and performance, etc. The point is that GPU performance does not come for free. We have to write creative algorithms that can take advantage of the GPU resources to properly accelerate our execution performance.

These approaches can be quite complex and offer tradeoffs between performance and execution determinism. When it comes to execution, determinism can mean different things, and have broad implications on the safety of a system.

DETERMINISTIC EXECUTION WITH RESPECT TO RESULTS

We have thus far focused on multiplications in CNNs as a measure of computational complexity, but notice that there are also quite a few summations. The previous section showed that to squeeze as much performance as possible out of a GPU we might use different algorithms depending on what tasks we want to parallelize. Our inferencing algorithm might attempt to parallelize the summations as well as the multiplications. But this presents a problem.

In mathematics, additions are associative. Suppose that we want to add six values: A + B + C + D + E + F. And associative property means that A + B + C + D + E + F is the same as F + E + D + C + B + A. An acceleration algorithm might decide to exploit this associative property of mathematics to parallelize the work of adding a string of values. This might be accomplished by breaking up the sums into separate blocks and performing the sums in each block in parallel.

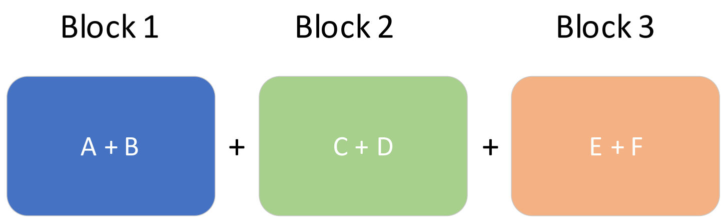


Figure 6: The processing of the input is split through different blocks in the GPU, where block 1 calculates A+B, block2 calculates C+D, and block3 calculates E+F. The results of each block calculation is summed for the final result.

As each compute unit completes the task of calculating the sums of each block, the final value can be calculated by adding up the results of each compute unit as they are completed. An important observation in this approach is that the order of operations is not deterministic. If this exercise is performed multiple times, it is possible that different blocks will complete execution at different times so that during run 1 the result is calculated as block1 + block2 + block3 (see Figure 6) whereas in run 2 the result might calculate as block2 + block1 + block3.

It turns out that while operations such as additions are associative in mathematics, in practice, where processors have a finite number of bits to store precision information for floating point values, these operations are not associative. The reason for this is that the precision error for the floating-point values being summed accumulates differently depending on the order of operations. In practice, this means that A + B + C + D = (B + C + D + A) + ϵ , where ϵ may vary depending on the order of operations.

In expert systems, neural networks are used for different purposes, from object detection and image classification to data processing and forecasting. While the fluctuations in calculated values due to precision errors are quite small, the effects on the overall application algorithm depend on how the application is interpreting these results. If the calculations are not deterministic, and the margin of error is not accounted for, the system designer cannot with any confidence decide whether the same input sample X could result in conflicting predictions on two separate runs.

DETERMINISTIC EXECUTION IN SPACE AND TIME

As we have seen, implementing efficient inference algorithms is not an easy task. Consider now that beyond the math, inferencing a neural network involves running software instructions through a CPU and a GPU. Even in the case where we are lucky enough to have an inference framework that can take advantage of a GPU to accelerate our millions of computations, we still need software running on the CPU to coordinate the operations taking place in the GPU. The GPU is there to accelerate the calculations, but the CPU is still in charge of the overall application pipeline. It is in charge of system controls, interpreting the results of the GPU calculations and making decisions based on those results. The CPU is even in charge of organizing the work that the GPU will do on its behalf. So, although we often suffer from tunnel vision when it comes to GPUs and machine learning, and think of the GPU as the center of the world, the CPU remains in charge of the whole operation.

The CPU is also in charge of running everything else in the system, from the operating system to mission critical software like that which is flying an aircraft or braking and steering a vehicle. How are we able to ensure that the CPU has enough time to execute the mission critical software, as well as our perception algorithms, and lesser important things like the infotainment system? In some circumstances you might decide to have multiple CPUs, with each CPU dedicated to a distinct set of applications, but chances are that even in these cases a CPU will be shared between multiple applications.

The role of a real-time operating system (RTOS) is to schedule slices of CPU time where a system integrator can decide when each application runs and how long each application can run on the CPU before it has to relinquish execution to another application. RTOSs enable applications to be categorized with different priorities, so that mission critical applications are allowed to preempt an executing lower priority application if a critical event is detected. Without an RTOS it is not possible to predict when each application in a system will execute or for how long. An overly egoistic application might decide to starve other applications by running tight or endless loops.

RTOSs are critical to ensuring that software can execute deterministically, that is, to ensure that we know when and for how long a specific application will execute. But RTOSs alone are not enough to ensure that our algorithms complete in a deterministic amount of time. Consider an application that dynamically allocates a memory buffer to perform a given operation. A system integrator may decide that this application must execute for 5 ms every 30 ms. The RTOS guarantees that every 30 ms our application will execute for 5 ms, but can we guarantee that 5 ms are enough for the application to perform its task every time? Because our application is dynamically allocating memory, the amount of time required to find a free block will depend on the state of the memory manager—which is shared by the entire system—at that point in time. It is possible that in some cases 5 ms will not be enough for the application to complete its duties, but worse, we have no way of knowing how long the application would need. This is why RTOSs alone are not enough. We must have other means of guaranteeing determinism in software execution.

For this, the application stack, including auxiliary libraries and frameworks such as inference engines, must be implemented following safety-critical standards. Vulkan® SC is a safety-critical API for accelerating graphics and compute operations through a GPU (or other accelerator devices). Vulkan SC was just recently ratified by the Khronos® Group—the same industry consortium that defines other successful APIs like Vulkan®, OpenCL™, OpenGL®, OpenVX™, and more—to meet functional safety requirements, where deterministic execution of software is required (for example, avionics, automotive, industrial, and autonomous markets).

VULKAN SC AS A CONDUIT TO DETERMINISTIC GPGPU SOFTWARE

Vulkan SC facilitates deterministic execution of software by simplifying the management of memory resources. In accordance with safety-critical best practices, Vulkan SC removes the ability to free allocated memory. When the Vulkan SC environment is initialized, the application must define all Vulkan SC resources (object counts) that the application will use in its lifetime. This allows a Vulkan SC implementation to maintain a memory manager that avoids memory fragmentation. This ensures that as a Vulkan SC application allocates device memory and creates Vulkan SC objects at runtime, the amount of time required to allocate the necessary resources is deterministic and can be properly accounted for when deciding on an RTOS scheduling strategy.

Vulkan SC also moves shader compilation offline. Removing shader compilation from the runtime has several benefits: now that the compilation process happens offline, the execution time required to compile shaders need not affect the runtime schedule of the applications sharing the CPU. From a certification perspective, offline compilation greatly reduces the amount of runtime code that must be certified. It also ensures that all shaders that can ever run on the GPU are accounted for, tested, and validated during the certification process. There is no worry that an untested combination of just-in-time shaders can be compiled at runtime and executed for the first time in a real-world scenario, possibly leading to fatal consequences.

Now we have an RTOS, and a safety-critical API to implement our software. Do we have a guarantee that our software is safe, and that it executes deterministically? Not yet. While a safety-critical API facilitates the implementation of deterministic software, determinism and safety are not an inevitable consequence of using such API. This is where functional safety standards and guidelines like ISO26262 and DO-178C come in to provide process oversight. These certification standards ensure that software is implemented according to strict requirements, and is tested against those requirements. This avoids potentially catastrophic situations where code, which was never executed during testing, is executed for the first time in a production environment.

RTOS, SC API, and functional safety standards all work together to provide a pathway for deterministic execution on the CPU, but what about the GPU? Most off-the-shelf GPUs in use today are designed for raw performance. The RTOS, which maintained a reliable scheduler on the CPU side, cannot save us here. The schedulers driving execution inside these GPUs are not concerned with determinism, and their design is a closely guarded secret of each GPU manufacturer. How does this affect our SC application? It means that we cannot deterministically calculate how long our shader will take to run in the GPU.

Consider an application that is using a neural network as part of a vision pipeline for a driver assisted self-driving car. The application runs on the CPU along a time slice scheduled by the RTOS. The application submits a video frame from a camera to the GPU to be processed using the neural network. The application needs to know the worst- case execution time of the neural network in the GPU to know if it will have enough time left in its CPU time slice to react to the result of the computations occurring on the GPU. Since we do not know much about how the GPU internally schedules its tasks, or the impact that memory accesses from the thousands of executing threads can have on the execution time of the shader, it is not possible to calculate the worst-case execution time of the neural network. For this to happen we need one last piece of the puzzle. The GPU itself needs to be designed with functional safety. The mechanism to schedule tasks on the GPU and the latency during memory reads and writes must be known. In the absence of a deterministic GPU, we must write defensive algorithms that employ best practices for writing shaders that lower the impact of non-determinism—for example, avoiding branches that depend on random variables and synchronization primitives inside the shader wherever possible. Frameworks that are designed for functional safety often trade performance for a less erratic execution time. These frameworks may also provide tools that help approximate the worst-case execution time of a shader even on non-deterministic GPUs as long as the shaders do their best to minimize complexity.

PREDICTABLE OPERATIONS PER SECOND

Considering the state of GPGPUs today, and the need to trade performance for determinism in functional safety designs, does it still make sense for hardware vendors to use Floating Point Operations per Second (FLOPS) as a measure of performance in these markets? Do we care what the peak performance of a system is if we can never reach it for fear of losing all control of the carefully choreographed sequence of execution happening on the CPU? It seems to me that we need a new unit of performance. Instead of discussing FLOPS or Tensor Operations per Second, the conversation should involve Predictable Operations per Second (PROPS). In accelerators that are designed with functional safety, and where execution time can be reliably predicted, PROPS will be equivalent to FLOPs. But, for hardware whose design is not deterministic, the PROPs will depend on the algorithm and the software stack implementing the algorithm. This is a more honest way of describing the tradeoffs between performance and safety that must be made for every shader that is implemented. These tradeoffs are per algorithm and not per GPU and have the power to immediately communicate whether advertised performance for a given system means anything to those of us pursuing Safe AI.

USING SIMULATION TO PROVE THE SAFETY OF EXECUTION

So far we have discussed the importance of deterministic execution for safe AI. In the introductory section we discussed the use of simulation to gain a level of confidence on model expression. Some companies have built massive infrastructure, equipped with state-of-the-art super computers and vast datasets to simulate real-world driving conditions to test and train the advanced driver assistance and self-driving systems. While we agree that simulation has its place in empirically showing that an algorithm can respond favorably in scenarios likely to arise in the real-world tests, simulation must not be used as a proxy for the proving safety of execution; that is, for proving that the algorithms execute deterministically, as we have defined it. The most important reason for this is that we do not have to rely on simulation to prove execution safety. There are proven, well understood ways of ensuring that software executes deterministically (SC APIs, RTOSs, functional safety standards). Secondly, using simulation to prove execution safety is unscalable and unnecessarily expensive.

Let’s use a relatively simple ADAS example. Suppose that you have an application capable of ensuring a vehicle keeps to its lane. There are sensors that the algorithm analyzes to detect lane markings, and the algorithm outputs a set of control settings to steer the vehicle so that it remains in its lane. Next, we put the algorithm through millions of miles of simulation to show that regardless of road configurations, quality of lane markings, and so on, the vehicle remains in its lane. What the simulation has done is empirically show that the car can handle lane keeping. Unfortunately, we often extend this result to also mean that the execution of the algorithm is deterministic enough to enable the car to move in real time in the environment. For example, some might argue if the execution is not consistently reliable, then you would expect the car to veer off the lanes (as perhaps lags and latency in processing might mean that although lanes are detected they are not consistently detected at the rate required to keep the car in its lane in real time).

The are problems with this argument. First, unless we are also simulating the ability of other critical applications to run on the same system uninterrupted, then we do not really know how well our ADAS application may work on deployment in conjunction with critical software sharing the CPU. Second, execution is clearly tightly dependent on the underlying hardware system. This means that even if we were able to empirically show that the execution of our application is reliable through simulation, that would only apply to the specific system in which the simulation occurred. The minute the hardware system changes, the simulation has to be rerun. Simulation is expensive and most independent hardware vendors and OEMS cannot afford to build the scale of computing and data collecting power that is required to run millions of miles of simulated driving. By building our applications on safety-critical frameworks based on the whole package of functional safety standards, safety-critical APIs, and safe-deterministic hardware designs, we can decouple the expression problem from the execution problem. We can imagine a scenario where some companies build perception models that are validated through simulation, which can then be exported and deployed to a myriad of platforms without requiring the expensive step of running simulation on every deployed system. Functional safety standards and processes enable scalability and lower costs by standardizing the safety of AI execution.

A LOOK TO THE FUTURE

There seems to be the opinion in certain automotive circles that while safety is important in advanced driver assistance systems, most applications in these systems need not reach the level of assurance of say, avionics. The thought is that as there is a human in the driver seat anyway, then most failures in the system could be corrected by the human driver taking over. In my view this misses an important point. In many ways safety in fully autonomous vehicles might be more important than in avionics, considering that cars traveling through busy intersections and highways have less time to recover from fatal failures than airplanes flying in relatively less congested airspace. This is obviously important in the context of security as well. Consider the effect a rogue application, or an attack involving the communications and control systems of a vehicle traveling at 100mph in a complex highway environment. Even if there is a driver in the driver seat, there may not be enough space for much troubleshooting when a problem arises.

As we look to the future, the autonomy industry must prepare to support densely packed highways and cities where real-time systems and reaction times will be tested to their limit. We must build an infrastructure based on sound principles of functional safety and deterministic execution. We must think in terms of PROPS. Standards like ISO26262, and safety-critical APIs such as Vulkan SC and OpenVX are paving the way to enabling that future. Now we need to agree that we must not compromise on safety, and that simulation only gets us so far. Beyond the prototype stage, we need rigor and we need standards.

AUTHOR

Ken Wenger

Senior Director, Research and Innovation

Ken Wenger is a Senior Director, Research and Innovation at CoreAVI. He has developed a number of safety critical products including graphics drivers and compute libraries. His current area of focus is research in the application of safety critical principles and guidelines in autonomous systems, using neural networks and other advanced machine learning algorithms. He is currently leading a research project into the use of semi-supervised learning algorithms for medical image diagnosis.