Supercharging with GPU Acceleration: The Use of Quadrotor Flight Dynamics

INTRODUCTION

Hardware acceleration enables software to execute faster. Faster than what? Well, faster than some benchmark. Technically, hardware acceleration can be achieved simply by upgrading the CPU in a system for a newer/faster version. But this isn't really what we mean when we speak of hardware acceleration. Instead, what we mean is writing software to explicitly take advantage of specific hardware features to accelerate the software execution. Modern processors are predominantly multi-core. The availability of multiple execution cores means that software developers are expected to write their algorithms in such a way that execution performance is maximized by parallelizing the work over as many cores as possible, instead of relying on powerful single-thread performance from the processor.

We have a multi-core CPU and we write our software to take advantage of every core the CPU has to offer. Are we done? Is this the best we can do to accelerate the execution of our software? Purely in terms of acceleration, we can certainly do better. The CPU is executing our precious software algorithm, but it is also executing everything else in the system, including the operating system. A true measure of acceleration would be to move as much as possible off the CPU and into dedicated hardware. This was the intuition that birthed the compute industry we have today. The explosion in machine learning (ML) and artificial intelligence (AI) research we see now is not due to fundamentally new computer science theories. Indeed, most of the math involved in these algorithms is decades old. The truly revolutionary thing in AI and ML today is the realization that we can accelerate the vast number of computations involved—which were once considered computationally intractable—using graphics processor units (GPUs). GPUs offer large arrays of computational cores. Provided that we write our algorithms in such a way that they can take advantage of the GPU cores to perform our computations, we can offload much of our algorithms to the GPU and leave the CPU free to continue running the system. This is true acceleration.

Now that we have established what acceleration looks like, at least at a high level, the next obvious question is why is it important? Is hardware acceleration a nice-to-have, or is it essential? The answer of course depends on the use case. If we can afford to wait for the results of our computations, then acceleration may not be all that necessary. But there are plenty of use cases in which the execution speed is so important that a slow execution means the system is unusable. These examples are usually found in systems that must respond to environmental changes in real time. In the next few sections of this document, we will look at the need for acceleration through the lens of a quadrotor example. The purpose of this document is to discuss the calculations involved in modeling the flight dynamics of a quadrotor, discuss which parts of the calculations are candidates for GPU acceleration, and demonstrate the importance of determinism to the execution time of these calculations.

ORGANIZATION OF THIS PAPER

What is a quadrotor?

In this section we give a brief description of a generic Quadrotor aircraft, its main parts, and the motion around an inertial and body axis.

Understanding the motion dynamics of a quadrotor

In this section we discuss equations presented in [1] and [2] for describing a quadrotor's motion that can be used to model the flight dynamics of the aircraft. We describe how using these equations a quadrotor's flight can be predicted based on its current state and controller input.

Evaluation of a quadrotor's flight dynamic equations and suitability for GPU acceleration

In this section we discuss how the equations presented for modeling the quadrotor's flight dynamics can be accelerated through an onboard GPU, and how we can ensure that the software will execute deterministically. We discuss available industry standard APIs such as Vulkan® SC™, which are key not just for acceleration, but for deterministic execution.

Conclusion

In the concluding section we revisit the need for acceleration and the methodologies available for achieving it. We also briefly discuss the importance of industry standard APIs for once again decoupling software from hardware solutions.

WHAT IS A QUADROTOR?

A quadrotor is a helicopter-like vehicle, but instead of a single motor, it has four. The typical quadrotor has two arms in a cross configuration with a pair of rotors at the ends of each arm (see Figure 1). Flight control systems for autonomous quadrotors is a very active area of research today. This research includes the development of algorithms that can enable a quadrotor to autonomously learn to follow a pre-determined trajectory from point A to point B. As we will see, the mathematics involved in designing a controller algorithm that can fly the quadrotor in the desired trajectory are computationally non-trivial. Yet, the calculations must be performed in the expected amount of time, or the quadrotor will quickly veer off course. Autonomous flying machines are textbook examples of systems that require not just acceleration, but also deterministic execution. That is, it is crucial that the software execute predictably fast, every single time. Before we get into acceleration and determinism of quadrotor software, let's first understand how quadrotors fly.

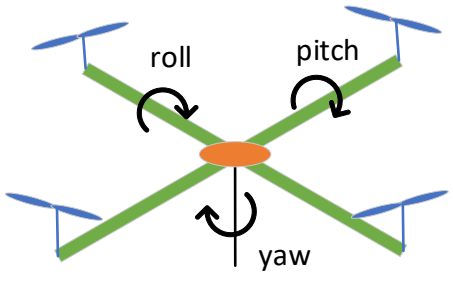

Figure 1: Quadrotor diagram showing four motors in a cross configuration.

UNDERSTANDING THE MOTION DYNAMICS OF A QUADROTOR

The motion of quadrotors has six degrees of freedom. Quadrotors can move in space in three dimensions along the x,y,z axis. And they can rotate about their own body-frame by rolling, pitching, or yawing. They can move in all these directions using only four motors. Therefore, quadrotors are called *underactuated systems*. They have only four actuators—the motors—but six degrees of freedom. Each pair of motors rotates in opposite directions. This ensures that forces generated by the rotation are balanced and the quadrotor can control its attitude. If all motors were to spin in the same direction, the quadrotor would spin about its axis and quickly fly out of control. There are different approaches to controlling the quadrotor, but whatever the approach, the method involves applying different voltages to the four motors. If all four motors spin at the same rate, the aircraft hovers. If the two motors at the back spin faster than the ones at the front, the aircraft will pitch down. If the two motors on the left spin faster than the ones on the right the aircraft will roll and bank to the right, and so on. The rate that each motor spins, and how we configure the spin-rate among the four motors determines the direction of motion in the inertial frame and in the body frame of the aircraft (see Figure 2).

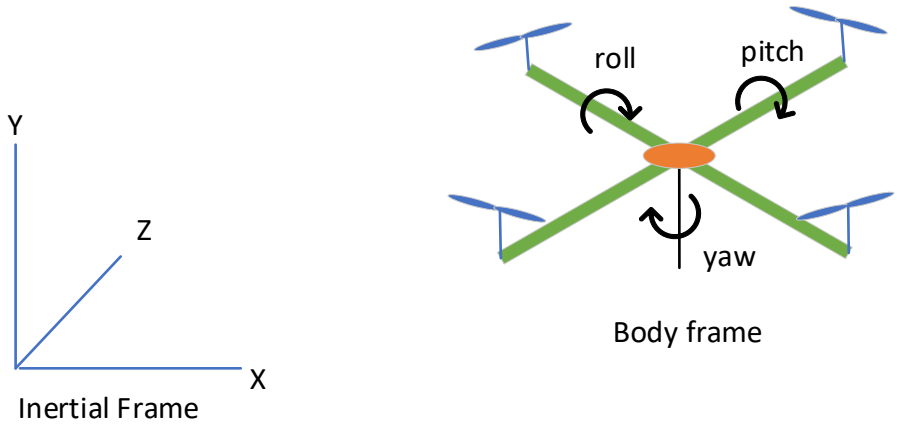

Figure 2: Diagram showing the difference between the inertial frame and the body of the quadrotor. The body-frame defines the quadrotor's motion about its own axis.

A QUADROTOR'S AUTONOMOUS FLIGHT

Now that we have a rudimentary understanding of how quadrotors fly, let's take a look at what is involved in the process of enabling quadrotors to fly autonomously along a prescribed path, and then let's try to identify areas which might be accelerated through a GPU. There are different strategies that may be used to enable quadrotors to fly autonomously, but at the most basic level we must: design a controller that generates the correct set of inputs (signals to each motor, for example) at each timestep to enable the quadrotor to remain in the desired path, and have a method for understanding the motion-dynamics of the aircraft. If we do not have a good model of the flight dynamics, then it will be impossible to design a controller as we will not be able to accurately predict the effect that each set of inputs will have on the aircraft (see Figure 3). In this paper we will look at accelerating the process of modeling the flight dynamics of the aircraft by using the mechanisms described in papers [1] and [2].

So, what does the process of modeling the flight dynamics of a quadrotor look like? We can think of the quadrotor's motion as being influenced by the signals sent to the motors and the forces in the environment acting on the aircraft—gravity, drag, inertia, and so on. A set of inputs will result in the aircraft being translated or rotated to a new position. One way to describe this is to say that the signals sent to the motors are a set of inputs that result in the model being spatially transformed to a new location. As a quick exercise, suppose that our quadrotor exists in a virtual world without physics, similar to the spaceship in the old Space Invaders game. In this case, predicting the new location of the aircraft given a set of inputs is trivial. The new location depends only on the input and some function designed by the engineer that determines how many 'units' the aircraft moves in the selected direction. For example, if the left arrow was pressed twice the aircraft will move two units to the left. But in the real world where physics determines the motion of objects, the function that describes the transformation of the object to the new position depends on the forces acting on the object as much as it does on the inputs. Correctly describing this function is far from trivial.

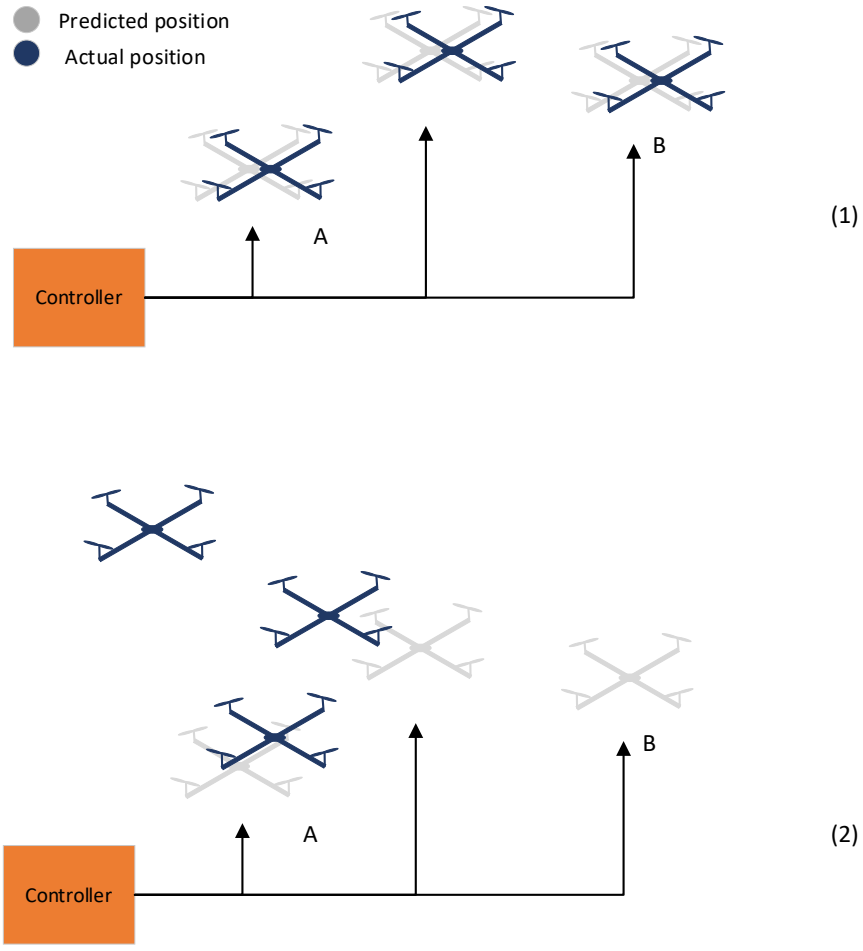

Figure 3: In this figure we see the importance of a good flight model for learning a quadcoptor controller. In panel (1) an accurate flight dynamics model allows the controller to learn the set of inputs necessary to get a quadcoptor from point A to point B. In panel (2) an inaccurate flight dynamics model means that the controller's understanding of the effect of a set of inputs on the quadcoptor's flight path and the quadcoptor's actual flight path greatly differ.

To restate this in mathematical terms we can say that there exists a function f that takes state $\mathbf{s}$ at time $\mathbf{t}$ with input parameters $\mathbf{u}$ and generates a new state at time $\mathbf{t+1}$.

$$s(t+1) = f\big(s(t), u(t)\big) \tag{1}$$

In this example, the state $\mathbf{s}$ is simply a vector of values consisting of the least amount of information necessary to accurately describe the aircraft. From equation (1) we can see that if we can figure out what f looks like, then we can predict the future states of the quadrotor. That is, we can predict its motion based on the provided inputs and changing state.

There are different strategies for defining or approximating f: supervised learning, reinforcement learning, standard physics.

Supervised Learning

Using supervised learning we could train a deep neural network to approximate f. To do this we would need a training dataset relating a set of inputs $\mathbf{u}$ and states $\mathbf{s}$ at time $\mathbf{t}$ and the corresponding, expected states at $\mathbf{t+1}$. The training process would then adjust each trainable parameter in f such that the correct transformation to $\mathbf{s(t+1)}$ is learned for any input in $\mathbf{s(t)}$ and $\mathbf{u(t)}$. In this manner the training process forces the neural network to discover the physics driving the flight dynamics of the system autonomously.

The problem with this approach, however, is that generating a dataset capable of training a neural network in this manner is extremely difficult. We would need to log thousands of hours of flight time where states $\mathbf{s(t)}$ and $\mathbf{s(t+1)}$ are captured along with the corresponding $\mathbf{u(t)}$. Latencies in sensor processing and communication between a controller and the aircraft means that aligning the set of inputs $\mathbf{u(t)}$ that produced $\mathbf{s(t+1)}$ from $\mathbf{s(t)}$ can be error prone and result in a training dataset that is not optimal. Consider also that to learn the physics affecting the aircraft's flight in this manner means that the dataset must contain samples of flight and maneuvers that translates the aircraft over every possible angle, rotation, acceleration. Otherwise, the learned model may only be accurate in a subset of flight scenarios.

Reinforcement Learning

There are other approaches, like Reinforcement Learning, in which the problem of the training dataset is avoided. In this case we don't need a training dataset to train a neural network (NN) model to approximate f Instead, the function is discovered as part of a set of episodes where the quadrotor flies around maximizing some kind of reward. The reward may be defined as minimizing the time it takes to fly from point A to point B. To maximize the reward, the system will be required to stabilize the flight of the aircraft, which will involve discovering the laws of physics that govern the motion of the system. While this process also involves discovering f in an autonomous manner and does not require creating and curating a training dataset, it involves flying the quadrotor thousands of times and going through the process of crashing the system several times. But the real problem with this approach is that defining a reward system that results in function f being approximated by the training process is also extremely difficult.

Reinforcement Learning is a fascinating area of artificial intelligence but among the most difficult frameworks to set up correctly, in a way that leads to generalization.

EVALUATION OF A QUADROTOR'S FLIGHT DYNAMIC EQUATIONS AND SUITABILITY FOR GPU ACCELERATION

Standard Physics

In the above examples we turned to artificial intelligence to try and discover f in some automated process. In other words, we aimed to find an automated process to learn the physics governing the flight dynamics of our aircraft. But is it not possible to propose a function f using standard physics? If we were to manually define f using our current understanding of the fundamental laws of physics, we could start with Newton's and Euler's laws of motion; that is what [1] does in its description of the motion dynamics of a quadrotor using first principles physics.

$$
\begin{aligned}
\dot{s}_1 &= s_4 + s_5 Sin(s_1) Tan(s_2) + s_6 Cos(s_1) Tan(s_2) \\
\dot{s}_2 &= s_5 Cos(s_1) - s_6 Sin(s_1) \\
\dot{s}_3 &= \sec(s_2)(s_5 Sin(s_1) + s_6 Cos(s_1)) \\
\dot{s}_4 &= -\left(\left(I_z - I_y\right) / I_x\right) s_5 s_6 - \left(k_r s_4 / I_x\right) + \left(1 / I_x\right) u_2 \\
\dot{s}_5 &= -\left(\left(I_x - I_z\right) / I_y\right) s_4 s_6 - \left(k_r s_5 / I_y\right) + \left(1 / I_y\right) u_3 \\
\dot{s}_6 &= -\left(\left(I_y - I_x\right) / I_z\right) s_4 s_5 - \left(k_r s_6 / I_z\right) + \left(1 / I_z\right) u_4 \\
\dot{s}_7 &= s_{10} \\
\dot{s}_8 &= s_{11} \\
\dot{s}_9 &= s_{12} \\
\dot{s}_{10} &= \left(-k_t s_{10}/m\right) + (1/m)(Cos(s_1) Sin(s_2) Cos(s_3) + Sin(s_1) Sin(s_3))\, u_1 \\
\dot{s}_{11} &= \left(-k_t s_{11}/m\right) + (1/m)(Cos(s_1) Sin(s_2) Sin(s_3) - Sin(s_1) Cos(s_3))\, u_1 \\
\dot{s}_{12} &= \left(-k_t s_{12}/m\right) + (1/m)(Cos(s_1) Cos(s_2)) u_1 - g
\end{aligned}
\tag{2}
$$

The system of equations in (2), as described in [1][1], is a set of first order differential equations that can describe the motion of a quadrotor under some simplifying assumptions. [1] presents these equations as a starting point for understanding the motion dynamics of a quadrotor, with the understanding that these equations do not fully capture the conditions under flight, and more must still be done. If our goal is to accelerate the computations required to control a quadrotor, then we must understand the equations that control the quadrotor. So what is (2) telling us? (2) is a system of equations that represents f in (1). The state vector describing the quadrotor at each timestep is represented by $\mathbf{S}$ and the set of inputs into the aircraft is represented by u, where $\mathrm{u} = [\mathrm{u}_1, \mathrm{u}_2, \mathrm{u}_3, \mathrm{u}_4]$ = [thrust, rolling moment, pitching moment, yawing moments].

[1]Some terms of the equation in (2) have been modified from [1] to use the naming convention typical of control systems equations, which uses S and U as the vectors for state and inputs. This is also the convention used in [2].

$$S = [S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}, S_{11}, S_{12}]$$

Where S_1, S_2, S_3 represent the quadrotor's orientation as the roll, pitch, and yaw (or Euler angles), S_4, S_5, S_6 represent the angular velocities in the quadrotor's frame of reference. S_7, S_8, S_9 represent the position of the quadrotor in the inertial frame, and S_{10}, S_{11}, S_{12} represent the linear velocities of the quadrotor in the inertial frame. Together, all the components of S describe the state of the quadrotor. Now what we need are equations or functions that enable us to calculate how the state evolves over time. (2) is a standard method for describing the motion of quadrotors using first-principle physics. That is, it is a standard method for defining the mathematical functions that describe the state evolving over time. Let's see how it does that.

$[\dot{s}_1, \dot{s}_2, \dot{s}_3]$ forms the angular velocity vector of the quadrotor in the inertial frame. If the pitch, roll, and yaw describe the current attitude of the quadrotor, the angular velocity vector describes how the orientation will change over time. $[\dot{s}_4, \dot{s}_5, \dot{s}_6]$ describes the rotational acceleration that the quadrotor is subjected to. Acceleration describes the rate of change of velocity, therefore, $\dot{s}_4, \dot{s}_5, \dot{s}_6$ describe the rate of change of S_4, S_5, S_6.

Similarly, $\dot{s}_{10}, \dot{s}_{11}, \dot{s}_{12}$ refer to the linear acceleration of the quadrotor, and $\dot{s}_7, \dot{s}_8, \dot{s}_9$ refer to the linear velocities of the quadrotor in the inertial frame. To understand what each element in (2) means, please refer to [1]. For the purpose of our discussion, we need to understand the intuition behind the equations and understand their limitations. Of particular interest are components k_t and k_r, which represent the drag coefficients for translational and rotational acceleration and are assumed to be constant. Assumptions like these and missing considerations for propeller force and torque functions which affect the motion of the aircraft mean that (2) is at best an approximation of the forces acting on the quadrotor. How to go from (2) to a more accurate description of the flight dynamics of the quadrotor is an active area of research, but in all cases the effort is concentrated on finding better functions for describing the evolution of each component of the state vector. That is, the left side of (2) is fixed. But the question is, how do we improve on the right side by finding better functions to describe each $\dot{s}$?

Combining Standard Physics and Deep Learning

[2] attempts to improve on (2) by appealing to deep learning techniques to approximate the linear and rotational acceleration vectors ($[s_{10}, s_{11}, s_{12}]$ and $[s_4, s_5, s_6]$) using fully connected neural networks. This is a compromise between the strategies of fully approximating using supervised learning, or reinforcement learning and solving the problem using first-principle physics, which we know is limited by the difficulty in manually accounting for every possible force acting on the quadrotor. Thus [2] proposes two neural networks f_v and f_w with the goal of learning a subset f, the dynamics of the linear and rotational accelerations of the quadrotor. The learning process minimizes the mean squared difference (MSE) between the acceleration predicted by the neural network, and the one observed by sensors in-flight. f_v and f_w are each two-layered neural networks represented by (3) as follows:

$$f(s, u; \alpha) := w^T \phi(W^T(s, u) + B) + b \qquad (3)$$

Where w and W represent the weight matrices of layers 1 and 2, B and b represent the bias terms, and ϕ represents the activation function, which in this case is ReLU.

We have just described the equations that govern the motion of a quadrotor. Next we must solve those equations to fly the quadrotor. For the period when the control system is learning how to fly the quadrotor, these equations must be solved at each timestep of the flight. This means the equations in (2) and the two neural networks described by (3) must be executed each timestep. Delays in solving parts of the equations mean that the control system will react to stale data, meaning the real position of the aircraft and the position that the control system is working on will not be aligned. This mismatch will prevent the neural networks from learning the true dynamics of the rotational and linear acceleration, and the aircraft will eventually fly out of control. Clearly, we want the system of equations in (2) and the neural networks in (3) to execute as close to real time as possible, and we want the execution to be deterministic. That is, we want to predictably estimate the amount of time required to solve those equations to know if our processors can keep up with the onboard sensors measuring the aircraft's state in real time.

Computationally speaking, the calculations described by (3) are a set of matrix multiplications performed between the input vector and weight matrix of each layer. In the example proposed by [2] there are only two such layers, so two sets of matrix multiplications are required. However, deeper networks may be useful in different circumstances and the deeper the networks the more matrix multiplications required. Similarly, the wider the network—meaning the more neurons per layer—the more elements there are in the matrices, so the more multiplications required. [2] does not specify the number of neurons per layer used in their neural networks, but considering the size of the input vector, the number may be in the 32 to 64 neuron range. We can assume a rough estimate of 1000 multiplications required per neural network during inference. That still leaves the rest of the equations in (2), which may combine to another few thousand calculations. While we can use the CPU onboard the quadrotor to perform these computations, the CPUs in these quadrotors are typically low power and already busy handling myriad of other systems (for example, the arrays of sensors) and the OS, which means that keeping the expected execution rate is no small task.

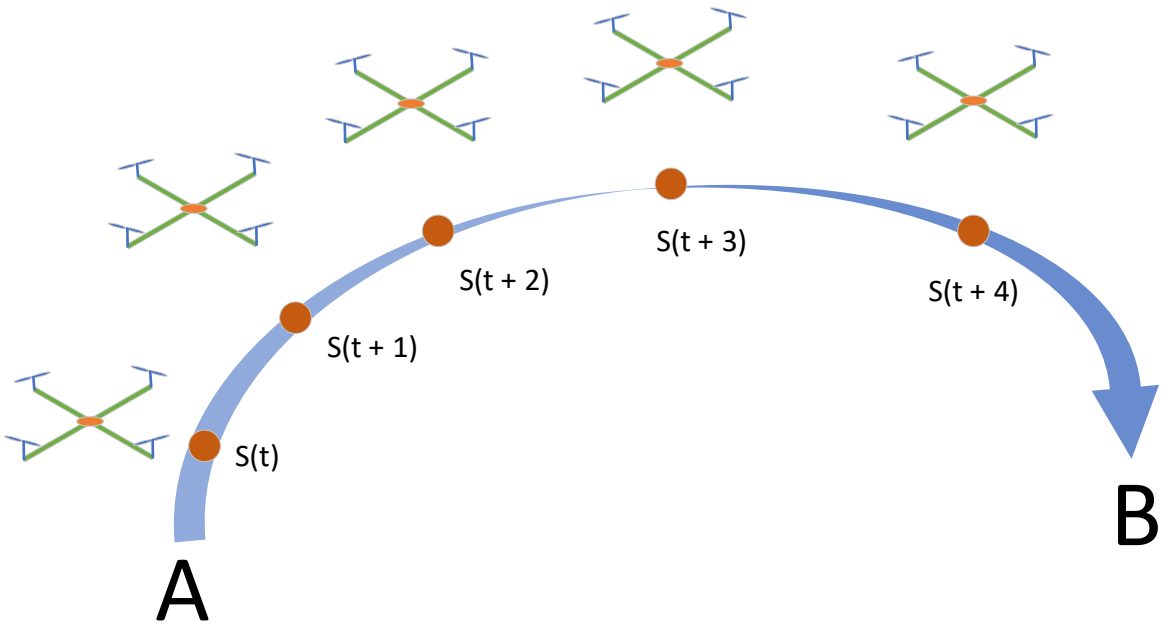

Figure 4: This figure depicts a quadrotor in different time-steps as it flies from point A to point B.

Conveniently, we know that GPUs are great at parallelizing tasks, and by offloading these operations from the CPU to a GPU we can free the CPU to attend to mission critical tasks like collecting sensor data, running the OS, and dispatching work to the GPU. The question is, are these operations good candidates for GPU acceleration?

Well, we know that neural networks are great candidates for GPU acceleration because GPUs are the fuel that ignited the current AI revolution. But fundamentally, the reason GPUs are great at accelerating neural networks is that GPUs are great at data parallel tasks. We have already said that neural networks consist mostly of matrix multiplications, and matrix multiplications consist of a set of dot product operations independent of each other. This means that each dot product operation can be performed by independent compute cores inside the GPU. In fact, GPUs are great at both data parallel and task parallel operations. Not only can we accelerate the execution of neural networks inside a GPU, we can also accelerate the execution of the entire set of equations in (2) by parallelizing the computations of each equation (see Figure 5).

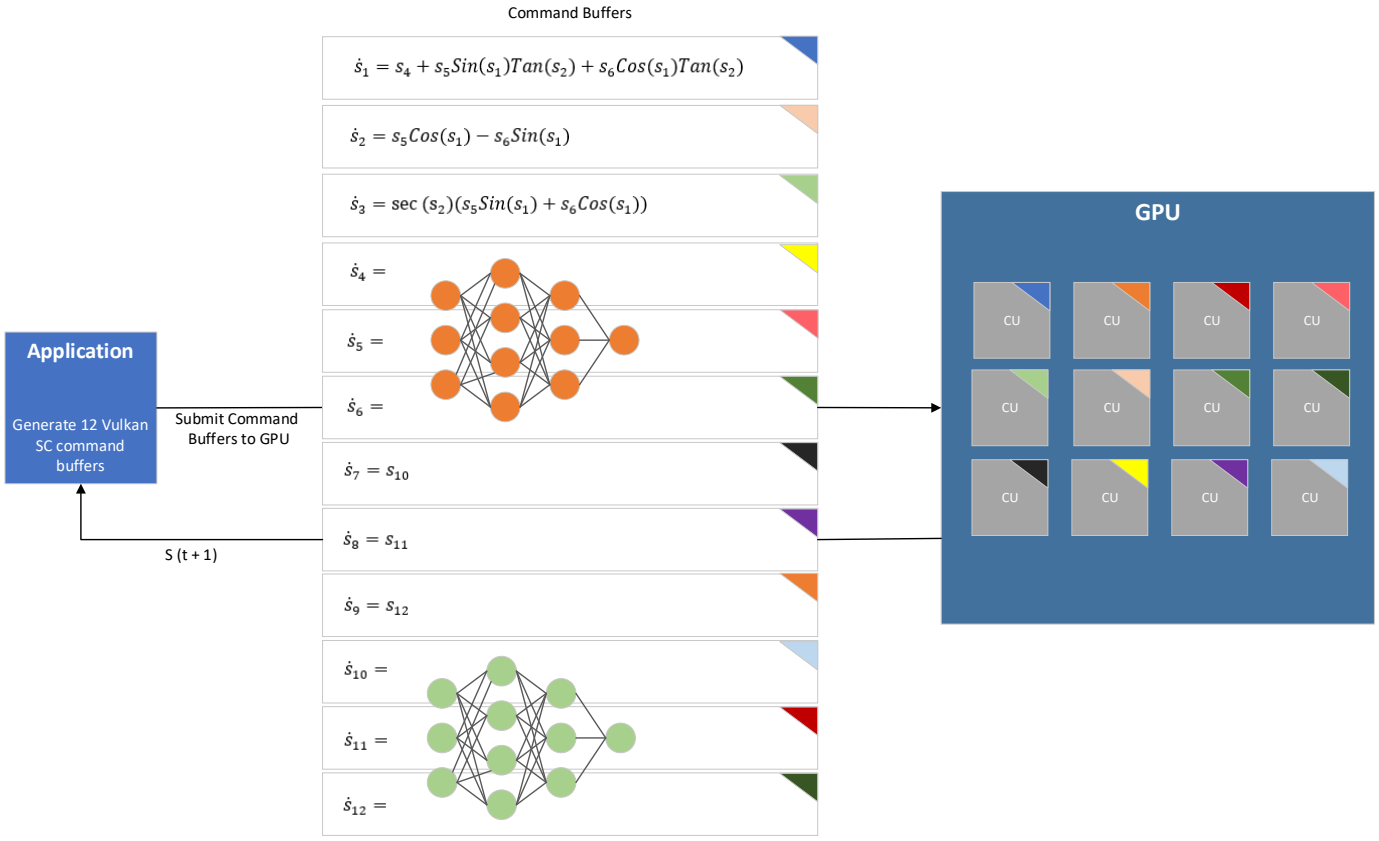

Figure 5: Function f may be accelerated through a GPU by an application using Vulkan SC to generate 12 command buffers each tasked with computing a different portion of the state vector S. The tasks in the command buffers may be distributed within a GPU's array of compute units. The resulting execution of the command buffers represents the evolution of state S to S(t+1) given u(t).

Standard compute APIs like OpenCL™ or Vulkan® from The Khronos Group, enable the acceleration of computational workloads using GPUs. But Vulkan SC (the safety critical version of Vulkan) is the only industry

standard API that can guarantee deterministic execution. This means that not only do we have a method for accelerating the execution of our equations, in Vulkan SC we also have a method for guaranteeing the execution time of those equations. Without deterministic execution, mean system performance is irrelevant. As previously discussed, if our algorithms run longer than expected even on just a few occasions, the synchronization between the quadrotor and the controller will spiral out of control. For autonomous flight it is imperative that execution be fast and deterministic. To achieve that, Vulkan SC is the only industry standard solution today.

CONCLUSION

We began our discussion by stating that modern silicon vendors emphasize parallelism over single threaded performance. We then explained that this introduces a coupling of software and hardware, in which software needs to be written to take full advantage of the execution engines offered by the hardware. The coupling of software and hardware solutions is great for performance, but terrible for scalability. As different hardware solutions become available in the industry, mature software stacks that are tightly coupled to hardware become stuck in deprecating ecosystems, unable to move with the evolving industry. We discussed Vulkan SC as a standard API capable of accelerating GPU compute in a deterministic way. Vulkan SC provides another advantage: as an industry standard API, Vulkan SC enables us to decouple software from hardware so that our software stacks are once again free to move to the industry-leading hardware solution of the moment, while still enabling us to tap into the compute resources of the underlying hardware.

In this paper we used a specific portion of a quadrotor flight control system as an example of a process that deserves GPU acceleration. We focused on the flight dynamics of a quadrotor, but this was just an example. In fact, every process in the flight control system, especially the controller itself should be considered for acceleration onboard the aircraft. Current research solutions often involve ground stations with GPUs accelerating the calculations for flight dynamics and controller. This requires constant and fast communication between the aircraft and the ground station which puts a lot of constraints on how useful the system can be. For example, the refresh rate of the sensors on the aircraft may be underutilized as they often run at a higher frequency than the communication link between the aircraft and the ground station. Increasingly available are computationally powerful and power-efficient GPUs that can process the computations necessary for control and flight dynamics on board the aircraft itself.

If current trends are an indication, research into autonomous flight will continue to rely on increasingly deeper and wider neural networks that demand deterministic acceleration. Thankfully, accelerating these calculations is a solved problem using Vulkan SC and GPGPUs.

REFERENCES

[1] N. Mohajerin, M. Mozifian and S. Waslander, "Deep Learning a Quadrotor Dynamic Model for Multi-Step Prediction," 2018.

[2] S. Bansal, A. K. Akametalu, F. J. Jiang, F. Laine and C. J. Tomlin, "Learning Quadrotor Dynamics Using Neural Network for Flight Control," 2016.

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.

The information contained in this document is for informational purposes only and is subject to change without notice. CoreAVI, the CoreAVI tracer logo, VkCore® SC, VkCoreGL® SC, ArgusCore™ SC, ComputeCore™, and combinations thereof are trademarks of CoreAVI. All other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

© 2022 Core Avionics & Industrial Inc. All rights reserved.