Unleashing GPUs for Network Function Virtualization: an open architecture based on Vulkan and Kubernetes

Juuso Haavisto*‡, Thibault Cholez†, Jukka Riekki*
*Center for Ubiquitous Computing, University of Oulu, Oulu, Finland, {first.last}@oulu.fi
‡University of Oxford, Oxford, United Kingdom, {first.last}@cs.ox.ac.uk
†Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France, {first.last}@loria.fr

***Abstract*—General-purpose computing on graphics processing units (GPGPU) is a promising way to speed up computationally intensive network functions, such as performing traffic classification based on machine learning at line speed. Recent studies have focused on integrated graphics units and various performance optimizations to address bottlenecks such as latency. However, these approaches tend to produce architecture-specific binaries and lack the orchestration of functions. A complementary effort would be a GPGPU architecture based on standard and open components, which allows the creation of interoperable and orchestrable network functions.**

This study describes and evaluates such open architecture based on the cross-platform Vulkan API, in which we execute hand-written SPIR-V code as a network function. We also demonstrate a multi-node orchestration approach for our proposed architecture using Kubernetes. We validate our architecture by executing SPIR-V code performing traffic classification with random forest inference. We test this application both on discrete and integrated graphics cards and on x86 and ARM. We find that in all cases the GPUs are faster than the Cython code.

I. Introduction

The proliferation of programmable Software Defined Networking (SDN) and Network Function Virtualization (NFV) pipelines in networking has introduced commodity hardware such as Graphics Processing Units (GPUs) to parallel packet processing. However, several limitations currently hinder the use of GPU for networking tasks. First, there is a lack of NFV-compliant architecture to deploy GPU-accelerated Network Functions (NFs) in a cross-platform manner. Second, while GPUs are well-suited for parallel applications because of multiple cores, the devices have high latency when interfacing with CPUs. Recent studies have addressed this performance issue by using, e.g., integrated GPUs and optimization methods in work scheduling using continuous threads and multi-buffering. Related work detailed below has proven that GPUs can speed up many packet processing applications. However, solutions tend to be specific and reliant on proprietary Application Programming Interface (API) or frameworks that limit their adoption as orchestrated functions.

This study proposes an architecture based on the Vulkan API, which allows fine-grained control over GPU resources, but in a cross-compatible way. With Vulkan, it is possible to test different GPU-accelerated approaches for packet processing but keep compatibility over any setup capable of using Vulkan. Notably, Vulkan compatible setups include integrated and discrete AMD and Nvidia cards and host platforms on ARM and x86. The originality of this work is double. To our knowledge, we are the first to leverage the Vulkan API and SPIR-V (Standard Portable Intermediate Representation) with Kubernetes orchestration as a common ground to deploy GPU-accelerated network functions. We demonstrate the applicability and good performance of our architecture on a real network processing use case and on three different GPU architectures. We also open-source all the software components constituting our architecture to the community[1].

The study is organized as follows: background on general-purpose computing on graphics processing units (GPGPU) technologies is introduced in Section II and the related work is presented in Section III. We detail our system framework with Vulkan and SPIR-V in Section IV and how it can be deployed with Kubernetes microservice architecture in Section V. In Section VI, we benchmark different hardware on a machine learning application performing network traffic classification and discuss possible optimizations. Finally, Section VII concludes the paper.

II. Background on GPGPU technologies

The primary rationale to leverage GPUs is that parallel computation tasks better match the properties of GPUs and can be sped up when offloaded to one. This so-called GPGPU paradigm has recently found its foothold with machine learning applications, where, e.g., neural networks can be sped up. Hence, GPGPU follows the general direction of distributed state like container deployments (MapReduce-like state synchronization) and

[1]https://github.com/jhvst/haavisto2021vulkan/

thread-granular parallelism principles of serverless applications (slow uniform memory enforces parallelism), but within a single physical component. As such, GPUs can also be thought of as a microservice runtime, especially given support from an orchestrator such as Legate [1].

Yet, historically the GPGPU standards and approaches have been dispersed at best: it remains commonplace to use graphics shading languages like OpenGL Shading Language (GLSL), designed for display frame construction, for GPGPU by using no-display-producing graphics pipelines. An alternative is Open Computing Language (OpenCL) which is meant for computation workloads but concentrates on homogeneousness over hardware platforms, e.g., GPUs, CPUs, and Field Programmable Gate Array (FPGA). Another option is the GPU manufacturer Nvidia's proprietary Intermediate Representation (IR) called Parallel Thread Execution (PTX), which is produced by Nvidia's Compute Unified Device Architecture (CUDA) compiler. While performant and widely used, PTX only works on Nvidia GPUs.

However, recent efforts for interoperability exist. The open GPU standards working group called Khronos, which is the organization behind GLSL and OpenCL, has released a new cross-platform graphics and compute API focused on performance, called the Vulkan API. Vulkan's new capabilities include a cross-platform and formally verified memory model called the Vulkan memory model and a class of cross-platform single instruction, multiple data (SIMD) operands for non-uniform group operations called subgroups. Still, while released in 2016, the Vulkan API has not seemingly yet become the leading API in the GPGPU space, plausibly affected by a design decision to only support a new open standard IR called SPIR-V. Thus, any application which wishes to adopt Vulkan's features would first need to update all the GPU code to SPIR-V. In practice, this is done via cross-compilers, but translations may not necessarily produce the most performant code nor be able to use the latest features. E.g., the Vulkan memory model was released as recently as September 2019 and adds new visibility semantics for variables use, paramount for deterministic computation results, but insofar languages that expose selection over these visibility levels seem non-existent. As such, evaluation of new Vulkan features tends to mean writing SPIR-V by hand, which serves as partial motivation for this study regarding to performance requirements.

III. Related work

Hardware acceleration of network processing workloads is a vast subject. Here, we will focus on previous initiatives that leverage GPGPU for specific network tasks in real-time. While the idea of using GPUs to perform packet processing tasks was introduced more than a decade ago with PacketShader [2] and its GPU-accelerated implementation of a software router, few initiatives have been proposed since. The approaches mainly divide into two categories. The first group of papers is focused on using GPGPU for a specific packet processing application. Among those applications, multi-field packet classification [3]–[6] and packet filtering [7] have been by far the most researched ones. Other security-oriented applications such as packet indexing [8], deep packet inspection [9], packet signature matching [10], or stateful packet processing for flow tracking [11] have also been proposed. Each paper proposes efficient data structures and algorithms for their application, optimized for execution on GPUs. Those papers demonstrate that many packet processing applications can benefit from GPU acceleration.

The second group of papers tackles the problem of making GPGPU more efficient when interacting with the network stack, in particular, by reducing the latency overhead. [12] presents the framework Snap, based on the Click modular router, that can offload computationally expensive tasks to the GPU. [13] develops a full network layer for GPUs that provides a socket abstraction and high-level networking APIs to GPU programs for efficient communication. [14] presents GPUrdma library to accelerate access to memory between GPUs across the network, and [15] optimizes the critical path for GPU to access the network. [16] evaluates more specifically integrated GPUs for packet processing and shows an interesting tradeoff between latency and parallelism. [17] states that many algorithms used in packet processing can benefit from GPUs and that APUs (Accelerated Processing Unit) can be a way to circumvent the latency added by discrete GPUs. [18] even proposes to execute a program written in P4 on GPUs, but the way GPUs kernel are written is not described.

Finally, those previous works are orthogonal to ours as they neither address the dynamic aspect of NFV nor propose an open architecture to deploy GPU-accelerated network functions as microservices. Further, they all rely on vendor-specific APIs (mostly CUDA) or homemade frameworks limiting their adoption. The closest related work is our paper [19] that studied the Vulkan and SPIR-V powered GPGPU paradigm in light of cold-start times and interoperability. This study shows that the time for starting GPGPU programs is on the millisecond scale, which makes Vulkan-powered GPGPU microservice cold-start latency similar to serverless applications and significantly faster than containers.

IV. Overview of the system framework

Our key contribution to the system framework is to leverage the Vulkan API and the latest SPIR-V versions. These technologies reduce the interoperability problems of past GPGPU approaches because Vulkan, as an open-source standard, facilitates integration. Regardless of the platform of choice, each device and architecture understands the same IR code in SPIR-V. In §V, we notice that SPIR-V is a valid IR for another reason: the produced binary files are only a few kilobytes in size and can hence be

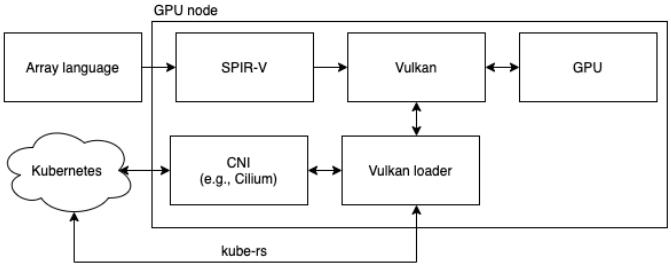

Fig. 1. Our proposed software stack of a GPU node.

inlined within configuration text files. We use this exciting property to propose an efficient way to orchestrate GPU Network Functions (NFs) on Kubernetes using Vulkan.

Our approach to writing the GPU implementation (in SPIR-V) of network functions is not direct. We preferred to use another language called APL (A Programming Language) as a modeling language first. So, we first write the implementation in APL and derive the SPIR-V code from APL. This intermediate step is not mandatory, but we further motivate the benefit of using a domain-specific language for GPU-accelerated network processing in Section VII. Fig. 1 depicts our software architecture per a GPU node, and Fig. 2 shows how it situates and integrates within a Kubernetes cluster. To elaborate, in Fig. 1, the SPIR-V code translated manually from APL is loaded on any GPU thanks to our Vulkan loader program. In Fig. 2, we see how our approach processes the network flows captured by the host through the Container Network Interface (CNI) abstraction, preferably via Cilium or other kernel-passthrough abstractions such as Data Plane Development Kit (DPDK) to accelerate packet processing further.

A. Program Loading and Execution

Like previous NFV approaches, such as Netbricks [20], in this study, we run the NFs unvirtualized. Here, we leverage low-level Vulkan API bindings to allow us to refine the GPU computation pipeline to better accustom to the performance of each hardware. To elaborate, we declare static resources that exist along the complete lifetime of the program, with other parts, such as program loading, working dynamically.

Our Vulkan-based loader program uses a Rust-wrapper library called `ash`[2] to interface with the GPU. In the abstract, `ash` provides conveniences, e.g., wrapping returned structures in vectors and providing default values for all API call structures. `ash` is considered low-level in the sense that all operations are "unsafe" in Rust, which means that the programmer must consult the official Vulkan API documentation to avoid undefined behavior. In turn, the Rust compiler is less useful than usually on memory safety: unsafe calls are meant as code blocks where the programmer surpasses the type system, and everything in `ash` that interfaces with the GPU is unsafe.

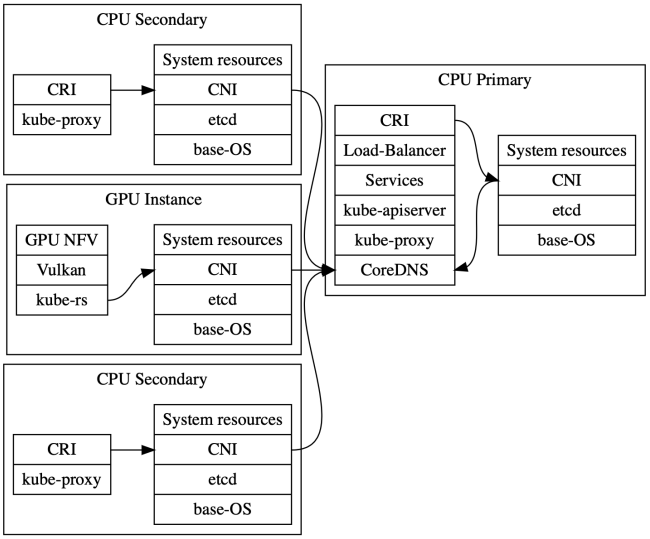

Fig. 2. Kubernetes integration. Here, four instances form the Kubernetes abstraction. The bottom parts of the records enable the abstraction on top, which, in turn, communicate with other nodes in the cluster.

The program flow of the loader is the following. We start by declaring static variables, i.e., ones that extend to the program's complete lifetime. Once initiated, the shader modules and pipeline layouts are retrieved from Kubernetes via the CNI using a Rust client library to Kubernetes called `kube-rs`[3]. This could be considered analogous to pulling a container image from a registry for deployment. After this initialization, we open the ports specified by the Kubernetes Services and start waiting for input to the functions. Once input is received, it goes through a standard Vulkan compute pipeline (detailed in our open-source code and illustrated step-by-step in Figure 3). In the final step, once the result is copied back to CPU memory, the result is written back to the network socket specified by the Kubernetes Service file. Here on, it is the job of Kubernetes CNI to forward the response. As such, it could be argued that this way, our approach yields itself well to the working principles of chained NFs. This allows such NFV chaining to be modeled in Kubernetes, spanning many Services with GPU NFs and traditional CPU containers complementing each other. However, we yield to the fact that our Vulkan pipeline is not state-of-the-art. Efficient use of Vulkan is a subject of many books, thus out-of-scope of what could have been prepared for this study. As such, further pipeline optimizations are left for future studies. Nonetheless, the Vulkan loader is the corner stone of our architecture and is highly technical. We decided to distribute it as an open source library[4].

[2]https://github.com/MaikKlein/ash

[3]https://github.com/clux/kube-rs

[4]https://github.com/periferia-labs/rivi-loader

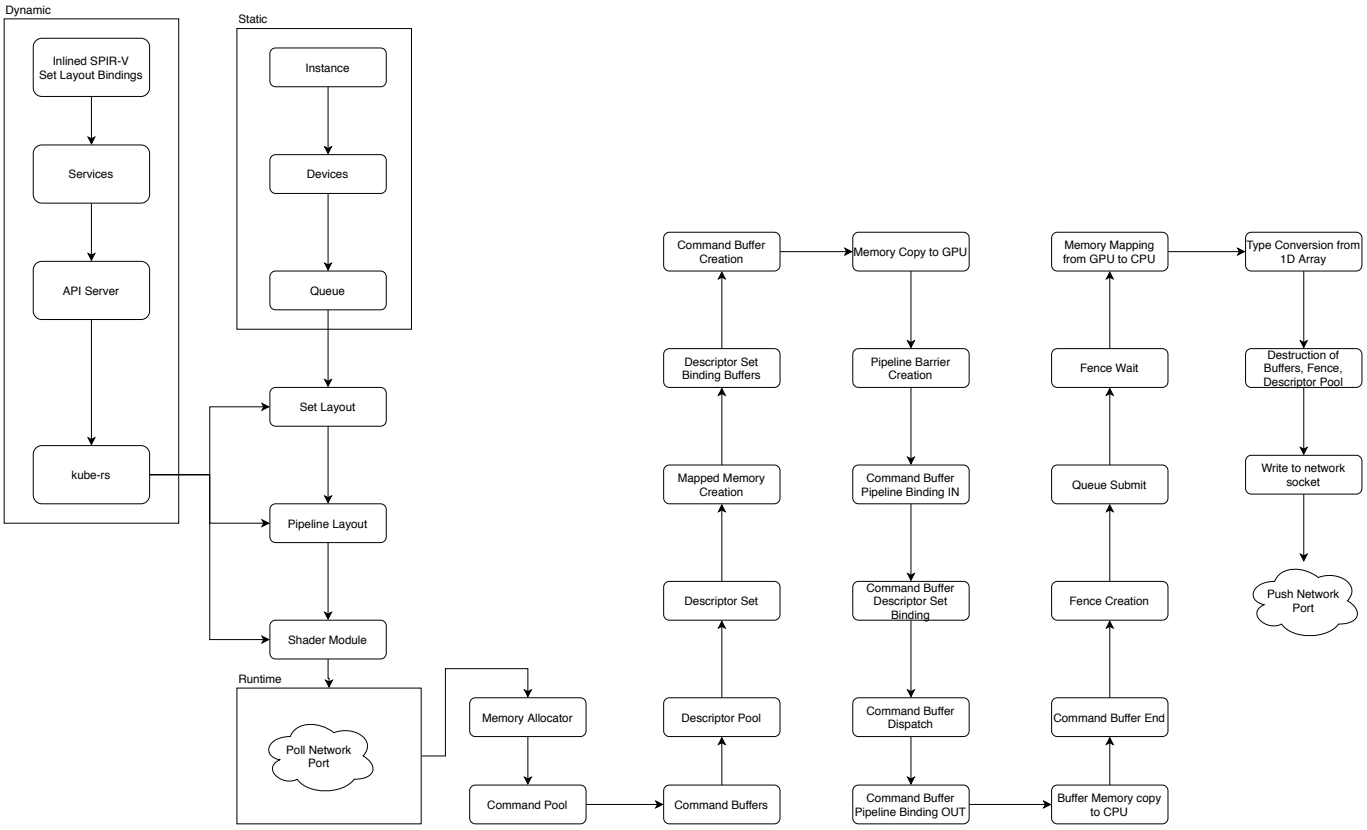

Fig. 3. Vulkan loader step-by-step

B. Orchestration

As mentioned, Kubernetes [21] is an orchestrator system for containers and can be considered as a system that conglomerates many physical servers into a single abstracted computing unit. Kubernetes is designed to schedule software packaged into containers, but it can be retrofitted to serve other purposes due to its modular design. We propose a way to retrofit Kubernetes to orchestrate GPU microservices in a container cluster. The GPU instances use the networking infrastructure to communicate SPIR-V binaries. This way, the GPU programs are managed as Kubernetes Services, and further, the SPIR-V binaries correspond to GPU containers. We may consider our approach "non-invasive," as we do not limit the functionality of the standard Kubernetes.

Next, to allow GPU NFs to be orchestrated, we integrate our Rust-loader application with Kubernetes Service abstractions (see: Fig. 4). In particular, the novelty of our approach here comes from the fact that SPIR-V programs can be inlined within Kubernetes Service abstraction as string-valued metadata. This is possible because SPIR-V kernels are small in size: our example of random forest prediction algorithm weights 2kb without compression. Furthermore, we achieve a non-virtualized and non-container approach while still managing to leverage Kubernetes APIs by defining the GPU nodes as non-schedulable (by not having Container Runtime Interface (CRI) installed, see: Fig. 2) and the Service abstractions as services without selectors. This way, we achieve two things: 1) Kubernetes does not try to schedule any existing worker nodes to spawn containers for the GPU Services as the Service declarations lack selectors, and 2) the Services are still exposing as a cluster-wide Domain Name System (DNS) entry via CNI. This keeps our proposed approach non-invasive to existing Kubernetes installations. Hence, our proposal to orchestrate GPU NFs this way can be viable for existing Kubernetes setups. The primary benefit of integrating with the standard Kubernetes scheduling workflow, oriented around the Service abstraction, is that the GPU Services in our proposed architecture are visible, routed, and exposed within the cluster as any other standard container-based Service. This way, we can simplify our loader program: CNI handles the networking to the primary node where CoreDNS handles Service discovery. As mentioned above, for better networking performance, CNI integrations like Cilium can leverage kernel-passthrough technologies like Berkeley Packet Filter (BPF) and DPDK. Such an approach might be helpful when running the GPU NFs on the network's edge, providing even lower latency to data inference and higher packet throughput.

V. Practical deployment and orchestration of GPU compute resources with Kubernetes

Due to Kubernetes' modular design, the architecture of each installation may vary. Here, we propose a barebone installation without many assumptions on the underlying modules. To illustrate our idea, we consider a network of a four-node system with one primary node and three secondaries (pictured in Fig. 2). Here, of the three secondaries, two are container hosts, whereas one is a GPU host. We propose that the GPU host remains non-schedulable for containers, per arguments on NF performance (e.g., see [20]). On the cluster level, we do not cordon the GPU hosts, but instead, never install a CRI on the host. This is a possible solution when the Kubernetes cluster installation is done modularly, e.g., per instructions on [22]. I.e., even though the GPU instance cannot schedule containers, it remains cluster-accessible when the CNI is installed adequately. The CNI is just another module on the Kubernetes stack and can be, e.g., `flannel`. The CNIs then rely on `etcd`, a distributed kv-storage and one of the most basic requirements for a Kubernetes instance. The hierarchy in Fig. 2 is bottom-up: the bottom part enables the use of higher abstractions, situated higher on the stack, which then interface cluster-wide. We insist that the CNI (networking), CRI (container runtime), and the GPU host operating system may be whatever in our proposed architecture. The independence is achieved by relying on the already modular interfaces provided by Kubernetes.

Once these essential services are installed on each node, a DNS layer for adequately addressing the cluster is needed. Usually, this is done via CoreDNS as it has a Kubernetes integration, but it may also be some other DNS server. In our proposal, the DNS server is required to route traffic to GPU-based Kubernetes Services. To quote the Kubernetes documentation, a Service is "an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service)." In other words, Service is the Kubernetes-native way of defining a microservice. With the proper abstractions detailed above, this promotes the GPU-based NFs to microservices. As such, it is logical for our proposal on the orchestration of GPU NFs as microservices to interface with the Service abstraction.

However, we do not use the standard declaration of Service because we do not want to run the GPU NF microservices inside a container. To elaborate, using containers for GPU applications is tricky and opinionated. For reference, the Kubernetes documentation on using GPUs[5] lists that to use, e.g., Nvidia GPUs, the cluster has to: 1) have Nvidia drivers pre-installed, 2) have nvidia-docker installed (which only works on Linux), 3) use Docker as the CRI. The steps include similar tweaks for AMD, including allowing the nodes to run in a privileged mode. In essence, these approaches limit the flexibility of the GPU node installations and require elevated execution modes for containers, which are usually meant to run unprivileged. Instead, we prefer declaring the Service abstraction without a node selector. To compare these declarations, consider Fig. 4. To avoid the Service becoming an orphan, we must declare an Endpoint abstraction. This can be done in a single command by separating the configuration declarations with three dashes, as shown in Fig. 4. We note that in our proposal, drivers still have to be installed to support Vulkan, but our proposal allows the GPU nodes to remain operating system agnostic while not relying on containers.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  spirv: AwIjBwAFAQA...
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
---

apiVersion: v1
kind: Endpoints
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 192.0.2.42
    ports:
      - port: 9376
```

Fig. 4. Kubernetes Service declarations. On the left, one with selectors, which would spawn container image MyApp. On the right, a Service without a selector, which would not spawn any containers. Our proposal uses the right-hand side version to spawn GPU NFs as microservices.

As can be seen in Fig. 4, our proposal for declaring GPU microservices within Kubernetes requires the SPIR-V binary file to be inlined within the metadata description. The binaries are encoded in base64. After the creation of the Service file, CoreDNS triggers a CNAME entry creation for the Service. We clarify that this is a standard procedure in Kubernetes, triggered by the orchestrator on Service creation. By default, this would expose an endpoint by name `my-service.default.svc.cluster.local` in each of the cluster's nodes' CNI routing table. In our proposal, what follows is that the Service creation events must be listened to by the GPU nodes using a Kubernetes client-wrapper, e.g., `kube-rs`. This means that each GPU node listens to the primary Kubernetes node to announce changes in the cluster Service entries. One such is found, the GPU nodes would pull the declarations using the kube-apiserver API. This would communicate the SPIR-V binaries required to load the particular GPU microservice into memory. Finally, if the Endpoints match the GPU node's local IP address, the microservice is provisioned

[5]https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/

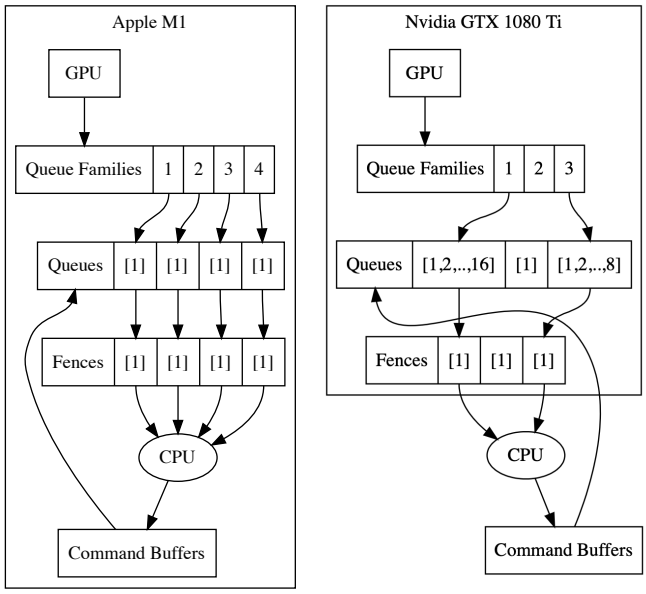

Fig. 5. Example of two GPU architectures layout leading to different optimizations

using Vulkan. Once the microservice is initialized, the corresponding port found in the Service declaration is opened on the node by CNI. When the port receives packets, the contents are unmarshaled in Rust and passed to the GPU. Once done, the result is written back to the connection from which it came. As such, it is the responsibility of the Kubernetes CNI to route data in and out. Such reliance cuts two ways: on the one hand, our proposal only works with Kubernetes. But, we do not make assumptions about what the Kubernetes installation has to be like. I.e., it is possible to leverage Kubernetes abstractions on top of the GPU NFs, such as LoadBalancer, to balance the load among many GPU nodes or any other networking constructs. For example, to create a function chain of NFs, we would encourage the chaining to be declared on the Kubernetes level. This way, the function-chain may mix both GPU NFs and CPU NFs by interfacing via CNAME entries. By this reasoning, we consider our proposal capable of introducing GPU NFs as part of a heterogeneous system consisting of CPU and GPU nodes.

VI. Evaluation

A. Use case description

The empirical part of the work is a development to [23], in which Random Forest (RF) prediction is used to label encrypted hypertext data. We use the paper's application in this study by extracting the prediction algorithm as a GPU microservice, using the well optimized CPU implementation of scikit-learn RF (written in Cython) execution time as the baseline. We choose this application also because the binary decision tree traversed by RF algorithms can be parallelized as each three is data-independent. Further, the number of trees to traverse is usually well above the usual logical threads that a CPU may have. As a thesis, such workload should see performance increase on GPUs, as GPUs can have up to thousands of cores. To elaborate, the physical processor of GPU is less likely to get throttled by its physical capabilities compared to its CPU counterpart, assuming that the program's execution time takes long enough time.

For the sake of reproducibility and to allow the community to further build upon our proposed architecture, all software components have been released in open-source on GitHub[6]. This includes:

- the Vulkan bootloader library written in Rust and used to load the SPIR-V code[7];
- Kubernetes configuration files to orchestrate a set of GPU-compute nodes;
- the SPIR-V assembly code of our network flow classifier to serve as an example (and the related but optional APL model).

B. Results

The benchmarks' (Table I) baseline was run on an Intel Core i7-9700 processor (8 cores). As introduced before, the application was a RF prediction over 150x6000x300 dimensional trees. As shown in Table I, the Cython code on OpenMP resulted in the baseline of 380ms. The results are gathered independently of the Kubernetes workflow as neither did the baseline run within Kubernetes. Yet, we do not expect any particular bottleneck to be introduced by Kubernetes itself. On the GPU side, all devices executed the identical SPIR-V code, which is possible thanks to the Vulkan-based architecture. To evaluate this aspect, we used different GPU manufacturers (Nvidia, AMD, Apple) and different processor architectures (x86, ARM). And while not evaluated at this time, the same architecture also allows different GPUs to be mixed on a single computer to achieve manufacturer-independent multi-GPU support.

Regarding the numerical results, the main point is that in all cases the GPUs are faster than the Cython code despite the delay of memory copies (especially for discrete GPUs), which proves the potential of our architecture. If we look more precisely at the different GPUs results, they are easily justifiable. The AMD card is the most powerful of the three and it also performs the best in our tests. On the other hand, the M1, while initially expected to lack in computation power in comparison, for example, to the GTX 1080 Ti, performs however significantly better than the latter. This can be explained because the chip's memory is integrated with the CPU thus what is lost in raw performance is gained in memory copy latency. We stipulate that with other more complex applications, such as neural networks, the M1 would start falling behind in performance as the memory copying becomes less relevant

[6]https://github.com/jhvst/haavisto2021vulkan/
[7]https://github.com/periferia-labs/rivi-loader

TABLE I
RUNTIME COMPARISON OF RANDOM FOREST MODEL OF 150X6000X300 TREES BETWEEN CYTHON AND SPIR-V.

	Device name	Runtime
CPU (Cython)	Intel Core i7-9700	380ms
GPU (SPIR-V)	NVIDIA GeForce GTX 1080 Ti	318ms
	AMD Radeon RX 6900 XT	136ms
	Apple M1	201ms

compared to core count and other physical properties. We remind that the purpose of this paper was not to optimize the memory copy delay but that this topic is already covered in related works.

C. Possible optimizations and discussion

Enhancing the GPU pipeline can be considered to consist of two factors: 1) Vulkan-based work scheduling and 2) SPIR-V code vectorization. For Vulkan, we stipulate the most significant area of improvement to depend on effective use of queues, as shown in Fig. 5 for Apple M1 and Nvidia GTX 1080 Ti. Other improvements on the Vulkan side include use of transfer-only queues, prevalent in discrete graphics cards. Transfer-only queues should improve memory copy times on discrete GPUs. Yet, proper application of these approaches is non-trivial: it requires concurrent software architecture and a high level of static information about the program.

On the SPIR-V side, SIMD instructions should be used where possible. Another approach, specific to our application, is refactoring the data structures. Here, the binary decision tree could be transformed into an indexed path matrix [24]. Such a parallel data structure would allow the GPU to also traverse the tree in parallel, avoiding the sequential while loop within the random forest tree traversal that currently limits GPUs performance in our use case. This comes at the cost of larger memory footprint of the dataset. Application of these approaches is also non-trivial, and a proper Domain-Specific Language (DSL) would be required. With a DSL, the language programming model could steer the programmer to avoid sequential or accumalative-style programming approaches in the first place. P4 did it for packet processing so we legitimately call for a new DSL that compile to SPIR-V to write efficient network functions that can benefit from GPU-accelerated NFV infrastructures thanks to the architecture we proposed in this paper.

We consider that producing highly parallelizable code with a low-level API still remains challenging. This challenge is two-fold: scheduling on the API side and efficient GPU code. It could be said that capturing the efficiency of the physical properties of GPUs is non-trivial because dynamic memory allocation is not possible. Thus an automatic system for static memory allocation is required. This way, more information about the program beforehand translates to wider applicability of performance tweaks. If achieved, such information could also provide novel yet practical features to NFV, notably, verifiable memory consumption policies.

That is why we leveraged rank polymorphic programming language APL as a parallel modeling language for SPIR-V IR GPU code in the first place. We identify a clear relation between restricted domain-specific languages and performance: the more we can statically know about programs, the better we can schedule them while leveraging hardware capabilities. This is especially true with GPUs, which have no shared stack memory and cannot allocate memory at runtime. As such, the property of *static memory management* as a language feature (for a review, see: [25]) is interesting to encourage using physical capabilities such as multiple queues. We remark that the GPGPU scene is approaching this paradigm already via languages such as NumPy and Julia, which, if integrated with novel type systems from dependent type theory, could capture static memory management via a property called *static rank polymoprhism*.

VII. CONCLUSION

We proposed in this paper a whole architecture to implement, deploy and orchestrate NFs on any GPGPU capable device using a combination of open and standardized technologies that are SPIR-V, Vulkan, and Kubernetes. We provided design guidelines to use this architecture and proved its applicability with an actual use case and our successful evaluation performed on different hardware. Our contribution is supported by software made available for the community, in particular, 1) our compute shader Vulkan loader library (in Rust), 2) Kubernetes configuration files to orchestrate a set of GPU-compute nodes, and 3) the APL code and its translation into SPIR-V assembly code of our network flow classifier example. Given the proof-of-concept work done in this study, we consider platform-independent GPGPU based network functions prosperous in the future.

Regarding future work, an integration of array programming and advanced type systems could address our undeveloped features on optimal scheduling by using dependent types to apply the most efficient GPU scheduling by exploiting the restrictiveness of the programming model of array languages.

References

[1] M. Bauer and M. Garland, "Legate numpy: Accelerated and distributed array computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–23.

[2] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: A gpu-accelerated software router," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 195–206, 2010.

[3] A. Nottingham and B. Irwin, "Parallel packet classification using GPU co-processors," in *SAICSIT 2010*, ACM, 2010, pp. 231–241.

[4] ——, "Towards a GPU accelerated virtual machine for massively parallel packet classification and filtering," in *SAICSIT '13*, ACM, 2013, pp. 27–36.

[5] C.-L. Hsieh and N. Weng, "Many-field packet classification for software-defined networking switches," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2016, pp. 13–24.

[6] S. Zhou, S. G. Singapura, and V. K. Prasanna, "High-performance packet classification on GPU," in *IEEE High Performance Extreme Computing Conference, HPEC 2014*, IEEE, 2014, pp. 1–6.

[7] C. Hung, C. Lin, and P. Wu, "An efficient gpu-based multiple pattern matching algorithm for packet filtering," *J. Signal Process. Syst.*, vol. 86, no. 2-3, pp. 347–358, 2017.

[8] F. Fusco, M. Vlachos, X. A. Dimitropoulos, and L. Deri, "Indexing million of packets per second using gpus," in *Proceedings of the 2013 Internet Measurement Conference, IMC 2013*, ACM, 2013, pp. 327–332.

[9] Q. Gong, W. Wu, and P. DeMar, "Goldeneye: Stream-based network packet inspection using gpus," in *43rd IEEE Conference on Local Computer Networks, LCN 2018*, IEEE, 2018, pp. 632–639.

[10] C. Hung, C. Lin, H. Wang, and C. Chang, "Efficient packet pattern matching for gigabit network intrusion detection using gpus," in *14th IEEE International Conference on High Performance Computing and Communication, HPCC-ICESS 2012*, IEEE, 2012, pp. 1612–1617.

[11] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "Design and implementation of a stateful network packet processing framework for gpus," *ACM Trans. Netw.*, vol. 25, no. 1, pp. 610–623, 2017.

[12] W. Sun and R. Ricci, "Fast and flexible: Parallel packet processing with gpus and click," in *Symposium on Architecture for Networking and Communications Systems, ANCS 2013*, IEEE, 2013, pp. 25–35.

[13] M. Silberstein, S. Kim, S. Huh, *et al.*, "Gpunet: Networking abstractions for GPU programs," *ACM Trans. Comput. Syst.*, vol. 34, no. 3, 9:1–9:31, 2016.

[14] F. Daoud, A. Wated, and M. Silberstein, "Gpurdma: Gpu-side library for high performance networking from GPU kernels," in *6th International Workshop on Runtime and Operating Systems for Supercomputers*, ACM, 2016, 6:1–6:8.

[15] M. LeBeane, K. Hamidouche, B. Benton, M. Breternitz, S. K. Reinhardt, and L. K. John, "GPU triggered networking for intra-kernel communications," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017*, ACM, 2017, 22:1–22:12.

[16] J. Tseng, R. Wang, J. Tsai, *et al.*, "Exploiting integrated gpus for network packet processing workloads," in *IEEE NetSoft Conference*, 2016, pp. 161–165.

[17] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "Apunet: Revitalizing GPU as packet processing accelerator," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX, Mar. 2017, pp. 83–96.

[18] P. Li and Y. Luo, "P4GPU: accelerate packet processing of a P4 program with a CPU-GPU heterogeneous architecture," in *Symposium on Architectures for Networking and Communications Systems, ANCS*, ACM, 2016, pp. 125–126.

[19] J. Haavisto and J. Riekki, "Interoperable gpu kernels as latency improver for mec," in *2020 2nd 6G Wireless Summit (6G SUMMIT)*, IEEE, 2020, pp. 1–5.

[20] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 203–216.

[21] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Queue*, vol. 14, no. 1, pp. 70–93, 2016.

[22] K. Hightower, *Kubernetes The Hard Way, Bootstrap Kubernetes the hard way on Google Cloud Platform. No scripts.* [Online]. Available: https://github.com/kelseyhightower/kubernetes-the-hard-way.

[23] P.-O. Brissaud, J. Francçis, I. Chrisment, T. Cholez, and O. Bettan, "Transparent and service-agnostic monitoring of encrypted web traffic," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 842–856, 2019.

[24] A. W. Hsu, "The key to a data parallel compiler," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 2016, pp. 32–40.

[25] R. L. Proust, "Asap: As static as possible memory management," University of Cambridge, Computer Laboratory, Tech. Rep., 2017.