Fine-grained accelerator partitioning for Machine Learning and Scientific Computing in Function as a Service Platform

Aditya Dhakal
Hewlett Packard Labs
aditya.dhakal@hpe.com

Philipp Raith
Hewlett Packard Labs
philipp.raith@hpe.com

Logan Ward
Argonne National Laboratory
lward@anl.gov

Rolando P. Hong Enriquez
Hewlett Packard Labs
rhong@hpe.com

Gourav Rattihalli
Hewlett Packard Labs
gourav.rattihalli@hpe.com

Kyle Chard
University of Chicago
chard@uchicago.edu

Ian Foster
Argonne National Laboratory
foster@anl.gov

Dejan Milojicic
Hewlett Packard Labs
dejan.milojicic@hpe.com

ABSTRACT

Function-as-a-service (FaaS) is a promising execution environment for high-performance computing (HPC) and machine learning (ML) applications as it offers developers a simple way to write and deploy programs. Nowadays, GPUs and other accelerators are indispensable for HPC and ML workloads. These accelerators are expensive to acquire and operate; consequently, multiplexing them can increase their financial profitability. However, we have observed that state-of-the-art FaaS frameworks usually treat accelerator as a single device to run single workload and have little support for multiplexing accelerators.

In this work, we have presented techniques to multiplex GPUs with Parsl, a popular FaaS framework. We demonstrate why GPU multiplexing is beneficial for certain applications and how we have implemented GPU multiplexing in Parsl. With our enhancements, we show up to 60% lower task completion time and 250% improvement in the inference throughput of a large language model when multiplexing a GPU compared to running a single instance without multiplexing. We plan to extend the support for GPU multiplexing in FaaS platforms by tackling the challenges of changing compute resources in the partition and approximating how to right-size a GPU partition for a function.

ACM Reference Format:
Aditya Dhakal, Philipp Raith, Logan Ward, Rolando P. Hong Enriquez, Gourav Rattihalli, Kyle Chard, Ian Foster, and Dejan Milojicic. 2023. Fine-grained accelerator partitioning for Machine Learning and Scientific Computing in Function as a Service Platform. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3624062.3624238

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624238

1 INTRODUCTION

GPUs and other hardware accelerators are indispensable for speeding up machine learning (ML), deep learning (DL), and other scientific computing workloads [3, 21]. Popular ML/DL frameworks and scientific computing workflow managers, offer APIs to facilitate the use of GPUs and other accelerators. Usually, these frameworks also offer runtime abstractions (*e.g.,* , runtime graphs, data movement) to improve hardware utilization by mapping nodes from a computational graph to computational resources. To date, GPUs remain the most utilized accelerators in the market; they are particularly well-suited to accelerate core ML/DL tasks such as matrix multiplication and tensor operations.

Function-as-a-Service (FaaS) promises to provide an abstracted development and deployment framework. A user of a FaaS framework writes a compute function and the framework does the heavy lifting of providing an environment for seamless function execution. Furthermore, FaaS frameworks enable workloads to scale as functions may be executed repeatedly. Several open source FaaS platforms able to run core ML tasks such as inference, are currently under active research and development [2, 4, 23]. These computational workloads would greatly benefit from proper accelerator support, specially for GPUs. Currently, most FaaS frameworks do not offer such capabilities.

Various studies have shown that many ML workloads, such as inference, do not fully utilize GPUs [17]. Other experiments also have shown high idle time and low GPU utilization for some scientific computing tasks [6]. Several authors have added GPU-awareness capabilities to serverless environments in different ways [5, 13, 28]. However, to the best of our knowledge there are no universally accepted solutions to these optimization problems and consequently, these issues related to low GPU utilization systemically manifest themselves in HPC systems [18]. As a serverless framework operator, it is crucial to maximize the hardware utilization to support more concurrent tasks, and therefore, increase profitability. Multiplexing, *i.e.,* sharing the GPU to run multiple tasks that do not saturate the GPU concurrently, increases the utilization [12]. NVIDIA's Multi-process service (MPS) and Multi-Instance GPUs allow the partitioning of the NVIDIA GPU resources into isolated partitions [20]. AMD GPUs also have comparable partitioning techniques. Accelerator resources, both compute and memory, can be

provided individually to applications that concurrently use the GPU. However, many FaaS platforms do not support heterogeneity beyond CPU and are designed for launching homogeneous tasks. Especially because many FaaS platforms (*e.g.,* , KNative, Parsl) can run on the container orchestration service Kubernetes which only has limited GPU sharing support. Other platforms (*e.g.,* , Parsl) rely on local environment of the node where the function will be executed. These approaches often share the GPU temporally by default and do not utilize the latest development in spatial GPU multiplexing such as Multi-Process Service (MPS) and Multi-Instance GPUs (MIGs).

In this paper, we use Parsl [2], the parallel runtime system behind Globus Compute (previously known as FuncX[4]). Specifically, we enhance Parsl with GPU partitioning capabilities and we demonstrate the use of GPU partitioning in Parsl by running a large language model (LLaMa2), a scientific computing application (Molecular-design), and well known deep neural networks (*e.g.,* , ResNet). In section 5 we report our findings and discuss the remaining challenges regarding accelerator partitioning for FaaS platforms. In our discussion, we also address the potential overheads that come with the use of accelerators.

2 BACKGROUND AND RELATED WORK

In this section, we introduce the Function-as-a-Service (FaaS) paradigm and the frameworks that we use in this paper (FunX/Parsl). We also describe GPU partitioning, multiplexing, and the workloads suitable for fine-grained accelerator partitioning.

2.1 Function-as-a-Service

Function-as-a-Service (FaaS) is a serverless computing model pioneered by cloud providers. In serverless computing, the cloud service provider manages the compute environment, execution of the code as well as the underlying resource management. In the FaaS model, users register programming functions alongside dependencies needed to execute these functions. The FaaS platform can then automatically create (and remove) the environment needed to execute the function. Deploying fine-grained and self-contained functions makes it simple for users to focus on the business logic and abstracts the underlying infrastructure. Moreover, FaaS enables the rapid spin up and down of function instances, making them suitable for stateless parallel applications commonly observed in scientific computing [25]. The scalability can increase performance and reduce resource usage while users pay only for the resources used over time. FaaS frameworks claim high scalability assuming homogeneity in the underlying hardware architecture. However, modern computing systems are often heterogeneous and the resources required by a single workload can also be complex. Therefore, FaaS frameworks should evolve to accommodate these challenges. For instance, a key enabler for FaaS is the ability to share resources and spawn multiple functions on a single node. However, sharing hardware accelerators – such as GPUs – in the same way, still poses challenges [9, 11, 32]. Addressing some of those challenges is the focus of the present work.

2.2 Globus Compute/Parsl

Globus Compute (Previously FuncX [4]) is a federated FaaS platform able to exploit HPC resources and therefore suitable for running scientific computing applications. Globus compute builds upon the open source parallel programming library, Parsl[2]. In Parsl, decorated Python functions (called "apps" in Parsl) are dispatched to Workers that are dynamically deployed on remote compute nodes. Globus Compute uses Parsl to manage the execution of functions once they are dispatched from the Globus Compute cloud service to a remote – and user-deployed – computing endpoint (*e.g.,* , a supercomputer).

2.2.1 Parsl Execution Provider and Executor. Parsl uses concepts like Execution providers and Executors to deploy and run a workload on different types of hardware such as laptops, multi-core servers or even supercomputers. Execution providers enable users to define configuration parameters. *e.g.,* Parsl execution provider for SLURM can use SLURM for resource management in a large hardware cluster. Here, we conduct experiments on a small scale testbed of 24 CPU cores and two GPU machines. Therefore, we used the *LocalProvider*, which provisions execution resources from the local system (*e.g.,* workstation, laptop) to deploy tasks in Parsl.

Parsl's Executor model builds upon Python's *concurrent.futures* executor API and allows for different Executors to be used in different situations. Parsl supports Executors designed to support different use cases; from extreme-scale to low latency. In this work we use the *HighThroughputExecutor* which implements a pilot job model and operates by deploying Python worker processes. Parsl also supports existing libraries such as Python's *ThreadPoolExecutor* to schedule and scale a given function to multiple CPU cores. In Listing 1 we present an example configuration of the *HighThroughputExecutor*, that must be defined before launching any tasks. Here, the configuration defines two executors; one with instructions for CPU use, and another one for the accelerators. Users define the number of CPU workers (*max_workers*) and the maximum available accelerators. Alternatively, we can also pick desired GPUs by providing the list of GPU-IDs. *HighThroughputExecutor* allows the function to access the number of CPUs and GPUs defined in the configuration. However, until now, *HighThroughputExecutor* could not partition the GPU to spatially share the GPU across multiple functions. In this paper, we enhance the *HighThroughputExecutor* to enable fine-grained GPU partitioning and spatial multiplexing.

Source Code 1: *HighThroughputExecutor* Configuration

```
# A configuration for a HighThroughputExecutor
def hsc(log_dir: str) -> Config:
    """Configuration to run ML work on the local node.

    Args:
        log_dir: Path to store monitoring DB
                and parsl logs
    Returns:
        (Config) Parsl configuration
    """
    config = Config(
        run_dir=log_dir,
```

```
        retries=1,
        executors=[
            HighThroughputExecutor(
                label='cpu',
                max_workers=16,
            ),
            HighThroughputExecutor(
                address='localhost',
                label="gpu",
                available_accelerators=1,
            ),
        ]
    )

    return config
```

2.3 GPU Multiplexing

GPU multiplexing, or running multiple workloads on a GPU simultaneously, can increase GPU utilization. Ideally, multiple GPU kernels would run concurrently, leaving little to no idle GPU streaming multiprocessors (SMs). However, in practice, there are multiple methods to multiplex a GPU. Table 1 summarizes different ways of multiplexing *NVIDIA GPUs* as well as their drawbacks and benefits. From the Table we can infer that there is no one-size-fits-all solution for GPU multiplexing; the final choice will ultimately depend on application and user requirements. Among the different multiplexing choices, CUDA MPS and Multi-Instance GPU provide both higher GPU utilization and ease of use while running multiple applications together. We will explore these two options in combination with the Globus Compute/Parsl framework described later in the paper.

3 APPLICATIONS

We profiled a few applications from different domains to understand if they would benefit from multiplexing the accelerators. We describe our findings below.

3.1 Molecular-Design

We used a molecular-design application [1] that implements an active ML strategy following sequence of tasks: (1) generate an initial pool of molecules from the MOSES dataset [22]; (2) use quantum chemistry simulations to calculate molecular properties of the generated molecules, *e.g.,* ionization potentials (IPs); (3) use the data from the previous steps to train an ML model that acts as an emulator of the simulations; (4) use the trained emulator to estimate the IPs of a larger pool of newly generated molecules; (5) perform quantum simulations on those molecules with higher emulator-estimated IPs; (6) enrich the ML training dataset with the results of the last simulations; (7) train a new emulator and so on. The output of this workflow are molecules with optimized physical properties. These calculations were performed using the Colmena framework [31] in an implementation backed by Globus Compute and Parsl.

3.2 LLaMa2

LLaMa2 is a recent Large Language Model (LLM) that is used for creating text-based AI assistants [29]. LLaMa2's basic use is simple; users submit textual prompts (questions, instructions, etc.) and the model generates a corresponding response. The model's ability to produce a sensible reply primarily depends on the training data, as well as the number of parameters and architecture of the model. LLaMa2 provides six different models that range in their intended use case (text or chat) and the number of parameters (7, 13 and 70 billion). LLaMa2's architecture is mostly based on the predecessor LLaMa1 which is based on the transformer architecture [29, 30]. LLaMa2's use cases depend on the deployed model where the LLaMa text is used for processing a single request-response situations and LLaMa2-Chat is a fine-tuned version targeting dialogues. The difference is crucial to the expected runtime behavior (*i.e.,* execution latency and resource usage) due to the expected varying length of interaction time and input. Having two separate models covering different use cases already can ease the deployment in large scale settings, however, it is yet unclear how they behave in scenarios where large GPUs are shared for different applications.

3.3 Imagenet Models

The creation and enrichment of the Imagenet dataset was instrumental for the surge and democratization of deep learning research [8]. Specifically, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) has provided a benchmark of labeled images organized in 1000 categories [27]. These images are used by competing research teams during these challenges. Indeed in 2015, a well known breakthrough was achieved when Imagenet models finally exceeded human capabilities [15]. In this paper, we use ResNet-50 and ResNet-101 for evaluation [14]. These models employ an architecture known as deep residual networks that uses convolution layers extensively.

3.4 GPU utilization of the applications

Current data center GPUs (*e.g.,* NVIDIA A100, H100, AMD MI210, MI250, etc.) host a large amount of computing power. A100 features 108 streaming multiprocessors (SMs) with theoretical 19.5 teraflops, while the comparable AMD MI210 with its 104 compute unit (CU) attains up to 22.6 teraflops for single precision (32-bit) computation. The newer generation of GPUs has an even higher theoretical peak. To saturate the GPU SMs/CUs, a large amount of optimized calculations like the multiplication of large matrices or the training of a deep neural network using large data batches, is usually needed. Indeed, many workloads are not able to fully utilize the GPUs and are therefore, wasting these resources. Here we briefly describe two scenarios where an ML or scientific computing application makes a poor use of GPU resources.

Difference in compute requirements within one workload: Deep neural network often present a software architecture where an input is processed by numerous *layers*, each processing the output of the previous layer. In convolution neural networks (CNNs) the dimension of output of each layer depends on the input as well as the filter size of each layer. Here, we present the floating points operations required in each layer of CNN and see how the compute requirement vary.

Fig. 1 shows the number of floating point multiplication and addition of each convolution layer of few popular DNNs available in Torchvision repository. With Fig. 1, we want to convey that

Table 1: Comparison of GPU multiplexing techniques for NVIDIA GPUs

Multiplexing Technique	Description	GPU Utilization	AMD Equivalent	GPU Resource reconfiguration	Software required	Drawbacks
Time-sharing	Every kernel gets an exclusive access to the GPU for a time.	Low	None	No	None	Low hardware utilization when an application cannot saturate the GPU.
Default CUDA MPS (Multi-process Service)	Kernels from different applications run concurrently when possible.	Highest	Default Multiplexing method in AMD ROCM	No	nvidia-cuda-mps-control	Some applications can be resource starved due to contention.
CUDA MPS with GPU Percentage	Applications are restricted to the maximum numbers of SMs they can utilize	High	Compute unit (CU) masking	App. process restart to reconfigure GPU resources	nvidia-cuda-mps-control	Application restart for GPU resource reallocation. No memory isolation.
Multi-Instance GPUs (MIGs)	GPUs are divided into multiple smaller instances. Compute and memory isolation.	High (lower than CUDA MPS)	None	Requires GPU reset	nvidia-smi	Requires GPU reset and application restart to change resource allocation.
vGPU	Designed for sharing GPU via VMs	High (But multiplexes in VM level rather than process level)	MxGPU	Requires restarting a VM	NVIDIA vGPU driver	Homogeneous resource division. Requires proprietary drivers to run.

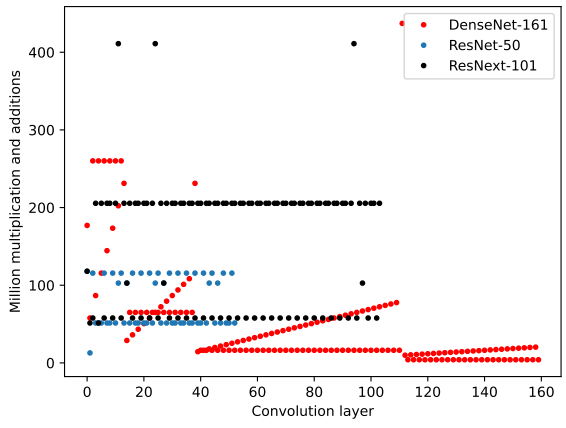

Figure 1: Variation of compute requirement per image for few convolution neural network performing image classification

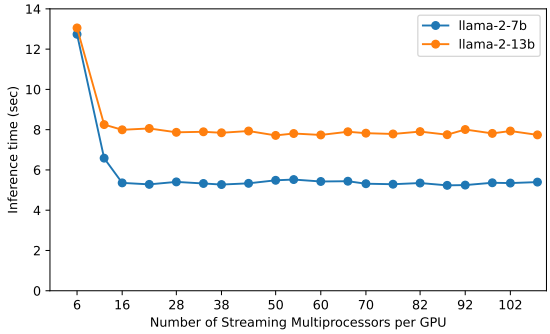

Figure 2: Inference run-time (seconds) of llama2 7 billion and 13 billion parameters using A100 GPUs. For llama 2 7b parameters, only 1 A100 GPU was used while for llama2 13 billion parameters 2 A100 GPUs were used for inference. Inference was conducted using 32 bit floating point parameters

different models have widely varying compute requirements at different points of their execution. Different layers' computation changes very rapidly. Even with different batch sizes, this variability remains.

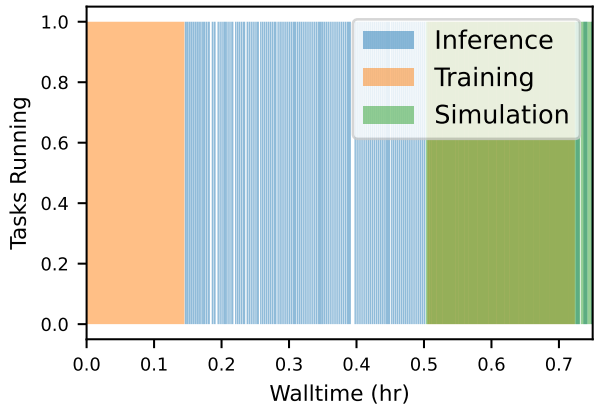

Figure 3: Time Spent on simulation, training and inference tasks during molecular-design workload

Small compute requirement: Many applications use GPU for acceleration, but they do not require a large amount of GPU. For instance, the large language model LLaMa-2 has both large memory requirements and low GPU utilization. We evaluated LLaMa-2 inference performing text completion tasks for 20-word sentences. Specifically, we used LLaMa-2 models with 7 and 13 billion parameters on a single NVIDIA A100 GPUs and 2 NVIDIA A100s, respectively. We limited the number of GPU SMs by using CUDA MultiProcess service (MPS). We present the inference time vs. number of SMs in Fig. 2. LLaMa-2 greatly benefits from GPU acceleration. Indeed, the same inference with CPU takes 180 and 360 seconds for the models with 7 and 13 billion of parameters, respectively; this is approximately 40 times slower than the GPU calculations.

We can also see that increasing the GPU processing *i.e.,* number of SMs only decreases the inference latency quite a bit but the latency does not decrease when we use more than 20 SMs. This indicates that the neural network model can only properly utilize about 20 SMs and does not benefit from more SMs even if they are available. This indicates that although the model is large, the compute requirement for this task is small. This kind of application is well-suited to be used in multiplexing setting.

Large GPU idle time: We profiled the Molecular-design application to see the hardware utilization. This application identifies the molecules that have certain desirable properties. The progress of the applications displays three main phases of computation, *simulation*, *training*, and *inference*. Among these phases *simulation* only utilizes CPU. There are times when the GPUs are idle as they are waiting for simulation results. Fig. 3 shows the total amount of time simulation, training and inference were running. We can see from the plot that there are many white lines between inference instances. There, the GPU is idle. Pipe-lining this application will yield higher accelerator utilization.

4 TECHNICAL DESIGN

We have described the original Parsl/Globus Compute HighThroughput executor architecture in § 2.2. Here we describe how we have enhanced it to enable GPU multiplexing.

4.1 Multiplexing with Parsl and MPS

To partition a GPU with CUDA MPS, users must specify the GPU% *i.e.,* the maximum percentage of SMs that an application is allowed to use. This is done by setting the environment variable CUDA_MPS_ACTIVE_GPU_PERCENTAGE to an integer value between 0 and 100 before the process starts. For example, setting the environment variable to 50% for an A100 GPU allows the process to use half *i.e.,* 54 out of total 108 SMs.

In Parsl, we modified the executor configuration to accept GPU percentage for each GPU. Specifically, we added a GPU percentage option that accepts a list containing the GPU percentage for each accelerator in the system. We present an example code listing in Listing 2. This code shows that the first three GPUs (GPU 1, 2 and 4) listed in *available_accelerators* field are indeed available to the executor but are limited to only 50%, 25% and 30% of those GPUs. When we want to multiplex a GPU by making it available to 2 functions, we can list the GPU twice as seen in Listing 2 and provide a GPU% for each instance of the function.

Source Code 2: *HighThroughputExecutor* MPS Resources

```
# Highthroughput executor with GPU percentage example
HighThroughputExecutor(
                address='localhost',
                label="gpu",
                available_accelerators=[1,2,4,0,0],
                gpu_percentage=[50,25,30,40,40],
            ),
```

These GPU percentages are enforced by updating the environment variable CUDA_MPS _ACTIVE_THREAD_PERCENTAGE before a new process executing the function is started. In Parsl, we modified the executor to also update the aforementioned environment variable. We need to make sure that the application *nvidia-cuda-mps-control* is launched in the compute node before any function with GPU code runs. Parsl also allows launching *nvidia-cuda-mps-control* with bash operations.

4.2 Multiplexing with Parsl and MIG

Multi-Instance GPU (or MIG) are smaller instances created out of ampere or newer generation GPUs. To create a new MIG instance, the user must put the GPU in MIG mode, and specify how the GPU should be partitioned in terms of memory and process by choosing a pre-defined configuration. *e.g.,* we can specify a configuration of *1g.10gb*, which will provide a MIG instance with $\frac{1}{7}^{th}$ fraction of SMs and 10 GB of GPU memory. Other configurations such as 2g.20gb, 3g.40gb, 4g.40gb and 7g.80gb are also available.

Each MIG instance is given a UUID, which can be provided to the functions so they launch their GPU kernel in that particular instance. We can then modify the Parsl configuration by specifying the MIG ID instead of the GPU ID in place of an available accelerator. Our configuration while working with MIG is shown in Listing 3.

Source Code 3: *HighThroughputExecutor* with MIG

```
# Highthroughput executor with MIG Instances
HighThroughputExecutor(
```

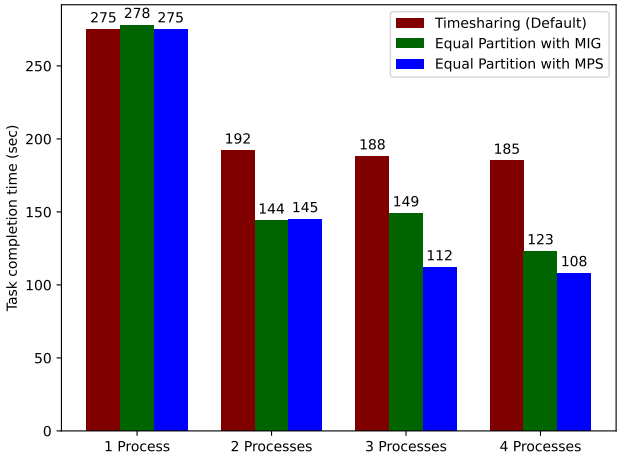

Figure 4: Time taken to complete a paragraph of text 100 times on LLaMa2. Work was divided equally across number of processes. *e.g.,* in 4 process experiment, each process will complete 25 text completion tasks.

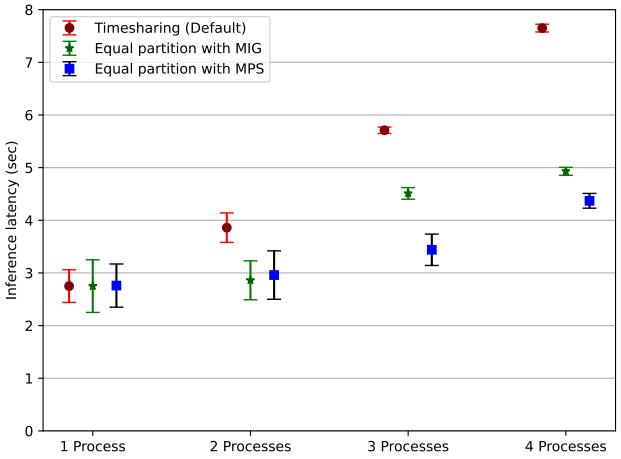

Figure 5: Average LLaMA2 inference latency with default timesharing, MPS, and MIG multiplexing

```
            address='localhost',
            label="gpu",
            available_accelerators=[MIG-1-UUID,
            MIG-2-UUID, MIG-3-UUID],
        ),
```

The Parsl executor launches a function with MIG-ID as CUDA_VISIBLE_DEVICES environmental variable, thus, causing the function to only run in the particular MIG.

5 EVALUATION

In this section we present the evaluation of GPU multiplexing with use of Parsl in our testbed.

5.1 Testbed

Our primary testbed is a virtual machine with 2 A100-SXM4 GPUs with 40 GB of memory each. Our testbed has 24 Intel Xeon CPUs with 2.20 GHz frequency. We are running Ubuntu 20.04 together with NVIDIA CUDA version 11.8 and Toolkit driver version 520.61.05. We have used PyTorch to run LLaMa2 and ResNet models while the molecular-design workload uses Tensorflow 2.8.0.

5.2 Multiplexed vs. Non-Multiplexed Execution

We ran experiments to understand the effect of GPU multiplexing on the memory-intensive application LLaMa2. We envision a scenario in which multiple LLaMa2 chatbots from different clients run in a serverless setting using Parsl/Globus Compute.

However, due to memory constraints, we could fit only four concurrent instances of LLaMa2 (7 billion parameter version) in an 80 GB NVIDIA A100 GPU.

We evaluated the time taken to complete 100 text completions and the latency of each completion when multiple chat models are running concurrently in the GPU. Fig. 4 shows the task completion time with LLaMa2 when running up to 4 models with timesharing, the default for NVIDIA GPUs. In another experiment, we partition the GPU equally with MPS, *i.e.,* when running 2 LLaMa2 processes we give each of them 50% GPU and so on. With MIG, we provide 3/7th of the GPU compute to each model when running two LLaMa2 models. We provide 2/7th of the GPU when running 3 models concurrently and 1/7th of the GPU to each model when running 4 models concurrently.

Fig. 4 shows that any form of multiplexing, even time sharing decreases total task completion time. However, we can clearly see that spatial multiplexing can yield even more parallelism from the GPU hardware and reduce the time to perform inference by up to 60% when compared to running only one inference process on the GPU, which is the default setting for several FaaS platforms. In terms of throughput, we can compute that spatially sharing 4 processes with MPS can lead to 2.5 times higher throughput than running 1 model at a time in GPU.

Spatial sharing with MPS or MIG leads to much higher GPU utilization and therefore gets a higher overall throughput and lower total task completion time compared to default time sharing. We should also see the difference between MPS and MIG. Both MPS and MIG take a similar time to infer 100 requests when 2 inferences processes share the GPU. However, MPS is much better when 3 processes are running on the GPU at the same time. This is due to the fact that MPS can divide GPU in a much more fine-grained way than MIG which must use pre-defined configuration. In the 3 process experiment, MPS gives about 1/3rd of GPU to each application while, MIG can only provide 2/7th of the GPU compute for each instance, thus, the difference in compute time. Similarly, in the case of 4 applications MIG can only provide 1/7th of a GPU for each application, while, MPS can provide 1/4th, therefore, running slightly faster.

We present the average inference latency in Fig. 5. Here also we see that increasing the number of processes in timesharing mode increases the latency rapidly. This is due to the fact that kernels from different models are interleaved in the timesharing mode, thus, adding more applications will increase latency for all running

applications. However, with MPS and MIG, we see a slower increase in latency. Moreover, MPS and MIG's inference latency is 44% lower compared to just timesharing when running 4 LLaMa processes in the GPU. MPS and MIG spatially share the GPU. An application running on one partition of MPS or MIG does not affect another application running in another partition.

6 DISCUSSION: EXECUTION OVERHEAD

Cold startup times are a major challenge for serverless platforms—they significantly degrade response times and can possibly render responses unusable. For example, low latency applications, such as Real-Time Object Detection, may be expected to produce results in less than a hundred milliseconds [24]. Considering that serverless functions are created and removed on demand, cold starts may happen frequently. Therefore, reducing cold startup time is an important research topic and one that can make serverless computing a viable option for variable workloads and time-sensitive tasks. Specifically in GPU workloads, we identify three parts causing overhead: (1) function initialization (including download, decompression), (2) GPU context initialization, and (3) application loading (e.g., loading AI model into video memory). Our results have shown that the loading time of LLaMa 2 13B can take up to 10 seconds—a significant overhead to the execution.

Moreover, using MIG and MPS-like techniques to partition GPU also has drawbacks. Once the GPU% is allocated for a process with MPS, the GPU% cannot be changed while the process is still alive, necessitating process restart to change the allocation [9, 10]. For LLMs like LLaMa, it results in 10-20 seconds of setup time before the model is ready to be used for inference. To reallocate MIG, we must shut down all the applications that are running on the GPU and re-configure the MIG with higher resources. Re-configuring MIG adds even more (1-2 seconds) overhead than MPS as well as interferes with other applications running on the GPU. However, as MIG guarantees memory and compute isolation, MIG might be preferable for spatially multiplexing the GPU for many users.

7 FUTURE WORK

With our understanding from analysis and evaluation, we are investigating on the following future research directions.

Re-configuring GPU resources Faster: We are working on an apparatus to lower the time to reconfigure GPU resources with MPS. As we have noted before, changing the GPU% for a DNN application with MPS requires restarting the application process. Restart will cause the application to reload the large DNN models to GPU again. Future work revolves around the reduction of model loading time. Different approaches have been explored by using shared CPU memory to enable access to models across function instances [16, 19] or pre-warming function instances by proactively initializing them to avoid any delays [26]. Memory sharing has been mostly done on CPU memory and not considering the possibility of sharing GPU-loaded models [7]. Improving the loading time of AI models into GPUs can have a positive impact for serverless and at the same time can improve pre-warming approaches.

We are working on an approach to share the model weights across different function instances. When a new instance of the DNN model is needed, the model code can refer to cached weights in the GPU and proceed with inference. When we are able to reduce the model loading time, which is a very big slice of DNN inference process startup time, we will be also able to re-load the model with different GPU% quicker than today.

Understanding GPU resource requirement: Another future direction we are pursuing is to understand the amount of GPU resources necessary for an application to complete running at a certain time as well as the approximation of runtime based on GPU resources. This challenge becomes crucial as we multiplex the applications and aim to change GPU resources depending on demand. We aim to create a tool that will give hints on what the expected GPU compute resources would be based on static analysis of applications.

8 CONCLUSION

In this paper, we presented different techniques for multiplexing the GPU. We analyzed some DNN applications and saw that only some DNN applications fully use GPU during inference. These applications can be multiplexed in GPU, increasing GPU utilization and overall throughput. We picked a FaaS framework, Parsl, and extended it to take arguments for GPU partitions. We evaluated inference with a large language model LLaMa 2. We showed that multiplexing GPU with resource partitioning lets us run multiple instances of LLaMa 2 concurrently and decreases the time to complete 100 inferences by 60% compared to running only 1 LLaMa model per GPU. Similarly, LLaMa 2's inference throughput by 250% when 4 LLaMa 2 models are spatially multiplexed in a single GPU compared to the default 1 LLaMa model per GPU.

ACKNOWLEDGMENTS

LW and IF acknowledge support by the ExaLearn Co-design Center of Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. KC was supported by NSF Grant 1550588.

REFERENCES

[1] 2023. Multi-Site Active Learning for IP Optimization. https://github.com/exalearn/multi-site-campaigns/tree/main/molecular-design. Accessed: 16/08/2023.

[2] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/3307681.3325400

[3] Sathwika Bavikadi, Abhijitt Dhavlle, Amlan Ganguly, Anand Haridass, Hagar Hendy, Cory Merkel, Vijay Janapa Reddi, Purab Ranjan Sutradhar, Arun Joseph, and Sai Manoj Pudukotai Dinakarrao. 2022. A Survey on Machine Learning Accelerators and Evolutionary Hardware Platforms. *IEEE Design & Test* 39, 3 (2022), 91–116. https://doi.org/10.1109/MDAT.2022.3161126

[4] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. FuncX: A Federated Function Serving Fabric for Science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/3369583.3392683

[5] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. 2022. SLA-Driven ML INFERENCE FRAMEWORK FOR CLOUDS WITH HETEROGENEOUS ACCELERATORS. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 20–32. https://proceedings.mlsys.org/paper_files/paper/2022/file/bcf9bef61a534d0ce4a3c55f09dfcc29-Paper.pdf

[6] Gregor Daiß, Patrick Diehl, Dominic Marcello, Alireza Kheirkhahan, Hartmut Kaiser, and Dirk Pflüger. 2022. From Task-Based GPU Work Aggregation to Stellar Mergers: Turning Fine-Grained CPU Tasks into Portable GPU Kernels. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 89–99. https://doi.org/10.1109/P3HPC56579.2022.00014

[7] Abdul Dakkak, Cheng Li, Simon Garcia De Gonzalo, Jinjun Xiong, and Wen-mei Hwu. 2019. Trims: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 372–382.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[9] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 492–506.

[10] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2023. D-STACK: High Throughput DNN Inference by Effective Multiplexing and Spatio-Temporal Scheduling of GPUs. *arXiv preprint arXiv:2304.13541* (2023).

[11] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. ECML: Improving Efficiency of Machine Learning in Edge Clouds. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. 1–6. https://doi.org/10.1109/CloudNet51028.2020.9335804

[12] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. Machine Learning at the Edge: Efficient Utilization of Limited CPU/GPU Resources by Multiplexing. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*. 1–6. https://doi.org/10.1109/ICNP49622.2020.9259361

[13] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, and Emmett Witchel. [n. d.]. DGSF: Disaggregated GPUs for Serverless Functions. *IEEE International Parallel and Distributed Processing Symposium* ([n. d.]). https://doi.org/10.1109/IPDPS53621.2022.00077

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385* (2015).

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. arXiv:1502.01852 [cs.CV]

[16] Myung-Hyun Kim, Jaehak Lee, Heonchang Yu, and Eunyoung Lee. 2023. Improving Memory Utilization by Sharing DNN Models for Serverless Inference. In *2023 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 1–6.

[17] Cheng Li, Abdul Dakkak, Jinjun Xiong, Wei Wei, Lingjie Xu, and Wen-mei Hwu. 2020. XSP: Across-Stack Profiling and Analysis of Machine Learning Models on GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 326–327. https://doi.org/10.1109/IPDPS47924.2020.00042

[18] Jie Li, George Michelogiannakis, Brandon Cook, Dulanya Cooray, and Yong Chen. 2023. Analyzing Resource Utilization in an HPC System: A Case Study of NERSC's Perlmutter. In *High Performance Computing*, Abhinav Bhatele, Jeff Hammond, Marc Baboulin, and Carola Kruse (Eds.). Springer Nature Switzerland, Cham, 297–316.

[19] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*.

[20] NVIDIA. 2023. Multiprocess Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. Accessed: 15/08/2023.

[21] Biagio Peccerillo, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. 2022. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *Journal of Systems Architecture* 129 (2022), 102561. https://doi.org/10.1016/j.sysarc.2022.102561

[22] Daniil Polykovskiy, Alexander Zhebrak, Benjamin Sanchez-Lengeling, Sergey Golovanov, Oktai Tatanov, Stanislav Belyaev, Rauf Kurbanov, Aleksey Artamonov, Vladimir Aladinskiy, Mark Veselov, Artur Kadurin, Simon Johansson, Hongming Chen, Sergey Nikolenko, Alan Aspuru-Guzik, and Alex Zhavoronkov. 2020. Molecular Sets (MOSES): A Benchmarking Platform for Molecular Generation Models. *Frontiers in Pharmacology* (2020).

[23] Philipp Raith, Stefan Nastic, and Schahram Dustdar. 2023. Serverless Edge Computing—Where We Are and What Lies Ahead. *IEEE Internet Computing* 27, 3 (2023), 50–64.

[24] Philipp Raith, Thomas Rausch, Schahram Dustdar, Fabiana Rossi, Valeria Cardellini, and Rajiv Ranjan. 2022. Mobility-aware serverless function adaptations across the edge-cloud continuum. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 123–132.

[25] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. 2022. Mashup: making serverless computing useful for hpc workflows via hybrid execution. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 46–60.

[26] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 753–767.

[27] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[28] Lukas Tobler. 2022. *GPUless – Serverless GPU Functions*. Master's thesis. ETH.

[29] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[31] Logan Ward, Ganesh Sivaraman, J. Gregory Pauloski, Yadu Babuji, Ryan Chard, Naveen Dandu, Paul C. Redfern, Rajeev S. Assary, Kyle Chard, Larry A. Curtiss, Rajeev Thakur, and Ian Foster. 2021. Colmena: Scalable Machine-Learning-Based Steering of Ensemble Simulations for High Performance Computing. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. 9–20. https://doi.org/10.1109/MLHPC54614.2021.00007

[32] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent {GPU} Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 69–85.