The Programmability of Distributing Machine Learning on Heterogeneous Resources - A Preliminary Examination

Godley, Christopher
Department of Computer Science
University of Colorado Boulder
Boulder, CO
christopher.godley@colorado.edu

***Abstract*—Machine learning workloads benefit greatly from parallelization, but dividing work between computational resources can be difficult when those resources are decentralized and heterogeneous in nature. The problems inherent to training and inference revolve around data locality across the network, which rely on the end resources being effectively utilized as well. Utilizing co-processors and accelerators effectively can be challenging even at the node level, so applying novel strategies to address these challenges in a larger distributed resource pool should improve performance across the system. Lastly, if such a system isn't accessible to target users it will not be a useful tool. Programmability of a vastly parallelized machine learning application should not requisite in depth systems experience for data scientists working at a mathematically abstracted level. Such a system requires a straight-forward interface without syntactic bloat obfuscating its utility.**

***Index Terms*—Machine learning, edge computing, distributed systems, neural networks, FPGA, GPU, accelerator, API, oneAPI, coprocessor**

I. Introduction

Computational resources are becoming increasingly available while also more difficult to fully utilize at scale. For example, suppose a data scientist is just getting started in the field. This budding scientist may only have a limited amount of compute horsepower they can take advantage of, a budget laptop for example, but it may be sufficient for learning how to build their models. As they progress towards developing big data applications, more resources will be needed. Go-to places for these resources include cloud providers, such as Amazon's AWS or Microsoft's Azure, which can provide computational resources as a service to meet the scientist's growing need. This developer is still limited to the cloud provider's offerings, and there can still be system integration issues that our data scientist was never originally concerned with. If this individual is trying to run anything at scale, data parallelization and system management start to play a larger role in the task at hand. For this hypothetical data scientist who was primarily focused on machine learning techniques and mathematical operations, this could be a large enough system-side hurdle to demotivate the distribution or scaling of their model. At the least, it is in the way of a more desirable workflow, which would allow our scientist to more easily run their work on any system they need to. A hitch in workflow may become a roadblock, and simple tasks may then become impassible.

Distributing work isn't the only problem our subject faces. The code base they work in, let's say Python, may not be interpretable on every system, or at the least may leave desired performance on the table by not making full use of available hardware resources such as GPUs, FPGAs, and other accelerators. If this scientist decides to make use of Amazon's AWS, there are many single node instances that provide drastically different compute capabilities. The first type of accelerator that will likely be used is a GPU. This allows for parallelization, but the GPU may be more optimized for gaming, or other types of operations than what our application needs. TPUs were designed to address this very issue. By developing a dedicated ASIC, the Tensor Processing Units (TPUs) are designed for more specific tasks and therefore benefit greatly from this speciality. [13] FPGAs are another accelerator that go in the other direction by offering slightly lower performance for increased flexibility. This increased flexibility also comes at the cost of development time, and the programming interface is often quite obtuse and syntactically different. All of these accelerators and more are capable of performing our desired tasks, but do not present any homogeneous means of development for their disparate design philosophies. Whether it's CUDA, OpenCL, or HDL, our Python based scientist is now required to not just parallelize their problem, but their development process as well.

Let's suppose our data scientist is developing a distributed deep neural network for image classification. This model needs to take input data from any number of edge nodes that have varying degrees of compute power. Accelerators such as GPUs or FPGAs will be present and must be utilized to maximize performance. To meet our developer halfway, we will require them to work in C++ but will provide the distribution and aggregation of work without syntactic bloat. OpenCL and OpenAPI style syntax will be provided for easy parallelization, while allowing for a single code base that may run operations on any type of resource. A distributed model will be implemented to manage data flow and perform training and inference on the most effective resources for overall throughput. This model will need to take into account networking latencies, input data locality, and heterogeneous compute resources. The components of such a model are addressed in the following section.

II. Preliminary Design Philosophy

A. Machine Learning at Scale

There are many novel machine learning frameworks that have been and are being developed to improve model performance in almost any metric. Generally, the work in this field is trying to maximize model accuracy while minimizing training and inference times. To implement a machine learning model from scratch, very basic mathematical libraries are required. For example, the backbone of neural networks is matrix multiplication, which can be performed with a series of multiply and accumulate instructions. General computer hardware handles these operations with ease, but the increasing scale of big data has been requiring more powerful computational resources. The beauty of matrix multiplication is that it is an example of an embarrassingly parallelizable operation. Matrix multiplication is an operation that can be divided and distributed at the granularity of individual multiply and accumulate instructions. In high performance computing environments, a data parallel API such as OpenMP may be used to subdivide the work among compute nodes. A supercomputer in this environment is inherently a centralized resource. The server racks typically all sit in a climate controlled data center with high speed interconnects physically linking each smaller compute node to the greater whole. In supercomputer architectures, such as those developed by Cray Inc, individual nodes are traditionally hardware resources consisting of a CPU and RAM, but might include an accelerator like a GPU or FPGA as well. The system is managed by a workload manager and application scheduler that has complete knowledge of the resources available; this makes the distribution and execution of a neural network fairly straightforward in such systems.

For our deep neural network for image classification, training on test data must be done before the model may be used to provide a desired output or inference. These are the two main states our model may be in: training and inference. A training model must be able to perform a forward inference propagation, but also a back-propagation in order to better align the result with the desired output in a process called stochastic gradient decent (SGD). For this reason, executing and managing the training of a neural network is more difficult than simply performing an inference on a trained model with new input. Performing these operations at scale on something like a supercomputer is not entirely dissimilar to conventional multi-threaded paradigms, and problems such as data locality play a minimal role. With total control over the distribution of the problem space onto the resources, the workload manager and application schedulers can effectively manage these applications at scale. [13]–[15]

B. Distributed Computing Resources

Opposed to supercomputer architectures, which are very expensive to develop, build, and maintain, conventional distributed systems can attain comparable scale but must replace high speed interconnects for networking across the internet. Internet latency and reliability aside, this removes the need to have all compute resources in the same room physically connected, and instead provides our familiar cloud and edge computing platforms. It's worth mentioning that there does exist a spectrum of clusters that are somewhere in between these two ideas, where nodes may be close in locality, but may still rely on conventional networking and subsequently require alternative workload management solutions. Ideally, any heterogeneous resource should be utilizable by a truly decentralized system. The job of a distributed workload manager in this scheme would be to maintain information about each node or resource, the latency associated with that resource as well as it's compute capability.

Another big difference with any model that deviates from a supercomputer is that memory becomes decentralized. Instead of utilizing high speed data nodes or servers as input to the machine learning workloads, information needs to be managed and passed accordingly. The data set or input data also becomes decentralized, which means we may make compute decisions based on where data lives or is generated. [8], [10], [11]

C. Operating System API

Performance sacrifices are often made in the data science world. For example, Python is very popular in the field and provides an ease of use that allows its users to focus on the math at hand rather than for example: memory management. When users start parallelizing computation to improve performance, programming in a language like Python leaves some of that desired performance on the table. It is for this reason that many learning models are written in languages like Java, or C++, to claim back some of that lost performance. What these heavier typed languages provide is access to the hardware. In an extension of this granularity of control, operating system resources such as threading APIs and loadable kernel modules provide direct access to hardware resources for user applications. For example, if our hypothetical data scientist was programming on an Nvidia GPU, they would be using the CUDA API for access to the GPU hardware. More generally, we can delineate the need to program the accelerator, the CPU, and communication between the two.

The most challenging commonly available accelerator to program is the FPGA. The compute fabric of FPGAs can be configured to run any kind of operation. These are the devices often used to develop proprietary ASICs. Once an operation is determined fit for the FPGA, the fabric takes time to be configured for that task. The programming interface to manage this is referred to as a hardware description language, or HDL. HDL can be extracted away at the C++ level via synthesis tools, or high level synthesis (HLS). Regardless, these tools take time to convert a code base as well as time to reconfigure the hardware fabric for different types of code. Systems like pvFPGA [1] and AmorphOS [2] try to address these challenges by generalizing the interface for interacting with FPGAs. For pvFPGA, Wang et al. decided to utilize a VMM to communicate with the FPGA. This is an example of a compromising system configuration complexity for performance. A more compelling model is the one presented by Khawaja et al. in the AmorphOS paper. This utilizes the more standard configuration of an FPGA on the PCIe bus, same as a GPU would be installed. The AmorphOS model makes efficient use of the FPGA fabric by controlling the pre-synthesized bit streams from the code base at a "morphlet" level, and uses these to quickly drive new functionality to the FPGA fabric. This is a compelling solution because with a known problem space, the synthesis time is removed and FPGA fabric can be shared by morphlets. In our image classification problem space, any FPGA resource can then be utilized by driving synthesized components of the segmented problem to any and all available fabric. This is analogous to sending threaded jobs to any available CPU, where the morphlet is the FPGA's "threaded" task.

D. Universal API

To maintain our data scientist's single code base while having it function across our heterogeneous resources, an API is needed. OneAPI [6] is being developed by Intel to address this very concern. This API provides C++ interfaces for programming across architectures such as Intel CPUs, GPUs, and FPGAs. There are currently some excellent examples on the Intel DevCloud regarding the data parallel programming of machine learning in OpenAPI. The selling point is that the model can be implemented in a single code base and then run on any type of Intel resource. The limitig factor is that this API only applies to Intel products and currently does not work on AMD, Nvidia, or Xilinx devices. Due to the open specification and the fact that their DPC++, or data parallel C++, utilizes SYCL as the heterogeneous computing model, this could be extended to any type of device. In a recent email to DevCloud users, Intel themselves stated "OpenMP 5.0 offers many of the same features as SYCL/DPC++ but supports the ISO language triumvirate of C++, C, and Fortran. If you want to program CPUs and GPUs using Fortran, C, or pre-modern C++ (i.e., before C++11) using an open industry standard, try OpenMP." This insinuates that for developing such an API for our distributed model, we may need to go to the source and utilize OpenMP or OpenCL for our data scientist's distributed image classification. OpenCL is more verbose than SYCL, and therefore offers better control for the heterogeneous environment outside of Intel's product lineup.

If our envrionment is Intel exclusive, OneAPI is very appealing. More realistically we will need to provide our users with interfaces implemented directly in SYCL, or even OpenMP or OpenCL.

III. Related Works

A. Distributed Deep Learning

1) Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices: Distributing deep neural networks over cloud, edge and end devices present many unique challenges. Teerapittayanon et al. present a DDNN model that allows for fast and localized inference at the edge/end devices while maintaining support for deep inference across the larger network. The idea is that for something like image classification, a deep network may be needed for a complex inference, but there are likely many smaller units of inference necessary that could reach intermediate conclusions. The authors address common approaches and their pitfalls, particularly the straightforward choices of either offloading large amounts of data to the cloud or performing limited classifications at the end device. The first suffers from latency and privacy issues, whereas the second is limiting in the types of machine learning that can take place. A CNN will not necessarily run on a refigerator, so simpler models like support vector machines (SVMs) may be chosen at the expense of accuracy. The authors then address implementing small neural networks at the edge and combine results as input space of a larger newtork. There were many challenges addressed for this approach, such as partitioning NN models over heterogeneous resources with low memory or power budgets, suferring from painfully high latency and/or complexity. Also, for DNN applications the local inferences may provide no real utility on their own.

The DDNN model expands on the authors previous BranchyNet [16] model, which allows for early exit points within a DNN. Training of a DDNN happens end-to-end so that the network and available resources may be optimized. While training, losses are combined from the many exit points and allow for the entire network to train while providing respectable accuracy depending on the depth of exit. The inference stage of DDNN relies on a predetermined exit threshold value to instigate early exits once a desired accuracy has been attained. End devices will send individual summaries to a local aggregator. The aggregator will determine whether the early exit should be taken, if more detailed summaries are needed, or if it should forward work to the cloud for final classification. Speaking to vertical and horizontal scaling, the DDNN model provides good performance at scale when both are considered. Communication is kept to a minimum and inferences are made in a kind of greedy fashion that allow for lower processing demand at the end devices. [7]

B. Compute Node Optimization

1) pvFPGA: paravirtualising an FPGA-based hardware accelerator towards general purpose computing: To handle the difficulties of optimizing FPGA resources, Wang et al. presented this model of virtualizing FPGA hardware via the Xen virtual machine monitor (VMM). The authors discuss the idea of horizontal and vertical solutions for big data problems, where MapReduce and Hadoop are provided as examples of horizontal solutions and compute node optimization is vertical. They drive home the message that both types of solutions need to play a role is large scale cloud applications. The paravirtualization of there system allows for OS drivers to assist in the traditional trap-and-emulate tasks of fully virtualized environments, allowing for better vertical solutions. This style of virtualization allows for better control of FPGA resources, where the Xen VMM can multiplex acceleration requests from multiple domains onto the available resource. The general idea is that a data pool can be shared over DMA between the processor and accelerator, and work can be presented and scheduled on the FPGA at the kernel level. Multiple applications can share FPGA fabric as well as DMA access. This system relies heavily on a streaming pipeline for efficient data transfer between CPU and accelerator; hazards and solutions are presented to maintain this pipeline. Hyper-requesting enables portions of multiple app requests to be simultaneously processed on the FPGA fabric via DMA context switches. A major downside is the lack of partial reconfiguration to accommodate these app requests. The pvFPGA system requires unique bitstreams for every combination of jobs that might want to share time on the accelerator. This results in slow response for new problems, as generating bitstreams is time consuming whereas using pregenerated bitsreams is fast. As the problem space grows, more bitsreams will be required, and it would not perform as well to provide these resources as part of a larger, general purpose service. It is desirable to support partial reconfiguration to maintain vertical performance across a large horizontal solution. [1]

2) Sharing, protection, and com-patibility for reconfigurable fabric with amorphos: AmorphOS is a system design philosophy built around sharing FPGA resources in a dynamic way by utilizing partial reconfiguration to keep bitstreams smaller. This model is designed to support demand sharing of the available FPGA fabric by segmenting work into morphlets. Traditionally, all of the available fabric on an FPGA is utilized as needed, and if an application's needs change an entirely new bitstream is generated. With a few optimizations to bitstream generation, particularly on the place-and-route stages, a task can be limited to a subset of available resources. These smaller morphlets can then be driven in whatever numbers they are needed. If there are too many for a single accelerator to run, they can easily extend to another. If an application consisting of a set of morphlets leaves fabric available, another application may then take advantage of the resource by utilizing partial reconfiguration to write its morphlets to the available space. Isolation and protection principles are applied to protect the security of individual morphlets, and the bitstream optimizations mentioned earlier are important for this. In reality, the AmorphOS system runs in either a low-latency or high-throughput scheduling mode. In low-latency mode, a known set of morphlets are combined into a cache of bitstreams which falls back on the problem of requiring timely generation of bitsreams for all morphlet combinations. High throughput mode is exclusively for high demand jobs in order to better utilize fabric, without the place-and-route optimizations detracting from the desired work. [2]

C. Programmability of Accelerator Platforms

High Level Synthesis Xilinx offers their HLS tools to allow C++ code to be directly compiled into bitstreams for their FPGAs. This happens through a synthesis toolchain that ports the C++ into an HDL and ultimately results in a bitstream that you can write to an FPGA. This is an important component in modern FPGA programming, especially for applications like machine learning. A limiting factor is that HLS is only supported on Xilinx FPGAs, but it is available to utilize as part of a larger toolchain if other devices also need to be utilized. This is an important tool on the search for accelerated single code base applications.

1) OneAPI: OneAPI is an Intel, open standard API that provides access to accelerator hardware via a single code base. The OneAPI interface provides a data parallel C++ syntax that is built on top of the SYCL interface. The issue that Intel is trying to address is the programmability of systems that consist of many CPUs, GPUs, and FPGAs. All of these devices are available under Intel's product line, and only these products are currently supported. However, this is an open standard and support for other devices is not unwelcome. OneAPI allows for efficient scaling of applications accross these devices with straight forward C++ interfaces for distributing workloads. Currently OneAPI is best utilized on an environment like Intel's DevCloud, where Intel CPUs are available for testing purposes alongside Intel FPGAs and even their new GPUs. Intel is not shy about recognizing OpenCL and OpenMP still have their place in many applications, and for finer granularity of control they are still necessary. OneAPI is still relatively new, however, so as development progresses these use cases may become better integrated.

IV. Conclusion

As our compute space grows with increasingly powerful connected devices, cloud and edge computing platforms become more tenable. Scaling applications need to be intelligently designed to mitigate the problems inherent with distribution across networks and to disparate devices. Strategies like the early exits in distributed deep neural networks should be applied when possible, and lessons on distributed training and inference should be leveraged as well. Similarly, the need for easy programmability will be extended as hardware becomes more diversified. Porting and distributing a code base should not be painful in a world where it is expected. By using a single code base, interfaces like OneAPI may be utilized to run applications on any variety of CPU, GPU, FPGA, and other types of accelerators. Other interfaces, like openCL, may be utilized in similar fashion to help distribute the work to heterogeneous resources. When designing for efficient distribution of work, approachable and intuitive programming interfaces should allow data scientists who know little of system programming to be liberated from these problems when developing their machine learning models.

References

[1] Wang, Wei et al. “pvFPGA: paravirtualising an FPGA-based hardware accelerator towards general purpose computing.” IJHPCN 10 (2017): 179-193.

[2] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 107–127, Carlsbad, CA, 2018. USENIX Association.

[3] Venkatasubramanian Viswanathan et al. “A Parallel And Scalable Multi-FPGA Based Architecture for High Performance Applications (Abstract Only)”. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. FPGA ’15. Monterey, California, USA: ACM, 2015, pp. 266–266. ISBN: 9781-4503-3315-3. DOI: 10.1145/2684746.2689115. URL: http://doi.acm.org/10.1145/2684746.2689115.

[4] ”Deep Unified Model For Face Recognition Based on Convolution Neural Network and Edge Computing” https://ieeexplore.ieee.org/abstract/document/8721062

[5] ”Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing” https://ieeexplore.ieee.org/abstract/document/8270639

[6] ”Porting a Legacy CUDA Stencil Code to oneAPI” https://ieeexplore.ieee.org/abstract/document/9150323

[7] ”Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices” https://ieeexplore.ieee.org/abstract/document/7979979

[8] ”Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing” https://ieeexplore.ieee.org/abstract/document/8876870

[9] “An Effective Training Scheme for Deep Neural Network in Edge Computing Enabled Internet of Medical Things (IoMT) Systems” https://ieeexplore.ieee.org/abstract/document/9109259

[10] “The Emergence of Edge Computing” https://ieeexplore.ieee.org/abstract/document/7807196

[11] ”Online Proactive Caching in Mobile Edge Computing Using Bidirectional Deep Recurrent Neural Network” https://ieeexplore.ieee.org/abstract/document/8660445

[12] “Convolutional Neural Network Construction Method for Embedded FPGAs Oriented Edge Computing” http://crad.ict.ac.cn/EN/Y2018/V55/I3/551

[13] ”In-Datacenter Performance Analysis of a Tensor Processing Unit” Norman P. Jouppi https://arxiv.org/ftp/arxiv/papers/1704/1704.04760.pdf

[14] ”A DomainSpecific Supercomputer for Training Deep Neural Networks” https://dl.acm.org/doi/pdf/10.1145/3360307

[15] ”Image Classification at Supercomputer Scale” https://arxiv.org/abs/1811.06992

[16] S. Teerapittayanon, B. McDanel, and H. Kung, “Branchynet: Fast inference via early exiting from deep neural networks,” in International Conference on Pattern Recognition, 2016.