Towards Accelerating Generic Machine Learning Prediction Pipelines

Alberto Scolari*,Yunseong Lee†, Markus Weimer‡, Matteo Interlandi‡
* Politecnico di Milano alberto.scolari@polimi.it
† Seoul National University yunseong@snu.ac.kr,
‡ Microsoft Corporation {mweimer,mainterl}@microsoft.com,

Abstract—Machine Learning models are often composed by sequences of transformations. While this design makes easy to decompose and accelerate single model components at training time, predictions requires low latency and high performance predictability whereby end-to-end runtime optimizations and acceleration is needed to meet such goals. This paper shed some light on the problem by using a production-like model, and showing how by redesigning model pipelines for efficient execution over CPUs and FPGAs performance improvements of several folds can be achieved.

Index Terms—Machine Learning, Model Scoring, Prediction Pipelines, FPGA.

1 Introduction

Machine Learning frameworks, such as Google TensorFlow [3], Scikit-learn [4], H2O [5] or Microsoft's Internal ML toolkit (IMLT), often structure the sequence of operations for training into pipelines of transformations, which typically operate on large, high-dimensional input data. Trained models are then served for prediction whereby a derived pipeline of data transformations is used to featurize raw input data points before model scoring.

While prior research focused on accelerating specific ML kernels [1], [2] or predictions for complex neural networks models [6], to our knowledge no research exists on the integration of accelerators within general ML prediction pipelines. Indeed, most of the input pre-processing, which is often a computationally demanding phase, still occurs on standard CPUs. However, some of these initial steps (for example string tokenization and hashing) have characteristics that make them amenable for acceleration.

Training and prediction pipelines have generally different system characteristics, where the latter are surfaced for direct users access and therefore require low latency, high throughput and high predictability. Accelerating predictions with specialized hardware such as FPGAs often requires redesigning these pipelines to efficiently utilize the accelerator's capabilities. This work sheds some light on the problem, showing how some common operations of prediction pipelines can benefit from hardware acceleration and from redesign.

Like increasingly many of the off-the-shelf ML libraries, the internal ML toolkit we used in this work expresses ML pipelines as a Direct Acyclic Graph (DAG) of *transformations*, which are mostly implemented in C#. Figure 1 shows an example of a model pipeline used for sentiment analysis (a more detailed description of each single transformation composing the pipeline will be introduced in Section 2). IMLT employs a pull-based execution model that lazily materializes input vectors, and tries to reuse existing data buffers for intermediate transformations. This largely decreases the memory footprint and the pressure on garbage collection, and avoids relying on external dependencies (like NumPY [7] for Scikit) for efficient memory management. Conversely, this design forces memory allocation along the data path, therefore making model scoring time hard to predict. This in turn results in difficulties in providing meaningful SLAs by ML-as-a-service providers.

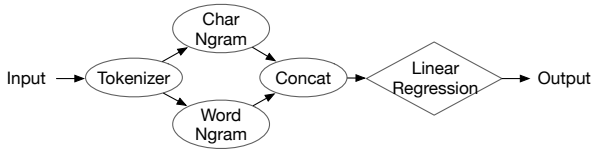

Fig. 1. A model pipeline for sentiment analysis.

Starting from models authored using IMLT, this work decouples the high-level user-facing API of the tool from the physical (execution) plane, with the goal of keeping data materialization minimal. This decoupling allows us to (1) optimize how transformations are executed, for instance by compiling several transformations together into one efficient execution unit; and (2) distribute different parts of the computation to heterogeneous devices such as CPUs and FPGAs (and, in the future, possibly to different machines) while optimizing the execution for each target device.

Using a production-like model we show that, compared to IMLT, our design improves the latency by several orders of magnitude and that generic ML prediction pipelines can benefit from acceleration via FPGAs. Interestingly, we find that a tuned CPU implementation outperformed the FPGA implementation; to fully exploit hardware acceleration, a redesign of prediction pipelines is not enough inasmuch as more hardware-friendly operations are not used at training.

Work done while Alberto Scolari and Yunseong Lee were at Microsoft.

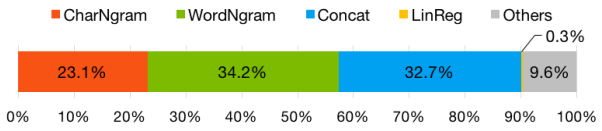

Fig. 2. Execution breakdown of the example model

2 The Case for Accelerating Generic Prediction Pipelines

Nowadays, several "intelligent" services such as Microsoft Cortana speech recognition, Netflix movie recommender or Gmail spam detector depend on ML model scoring capabilities, and are currently experiencing a growing demand that in turn fosters the research on their acceleration.

While efforts exist that try to accelerate specific types of models (e.g., Brainwave for deep neural networks) many models we actually see in production are often generic and composed of several (tens of) different transformations. Many workloads run in a prediction-serving, cloud-like setting, where prediction models coming from data science experts are operationalized. While data scientists prefer to use high-level declarative tools such as IMLT for better productivity, operationalized models require low latency, high throughput, and highly predictable performance. By separating the platform where models are authored (and trained) from the one where models are operationalized, we can make simplifying assumptions on the latter:

- models are pre-trained and do not change during execution; [1]
- models from the same user likely share several transformations, or, in case such transformations are stateful (and immutable), they likely share part of the state; in some cases (for example a tokenization process on a common English vocabulary) the state sharing may hold even beyond single users

These assumptions together motivate the research in accelerating large portions of generic ML training pipelines: indeed, computationally-heavy transformations with immutable state can be thoroughly optimized and offloaded to dedicated hardware with only a one-time, setup cost for setting the state. Once the most common sequences of transformations are identified in a workload, multiple prediction pipelines can benefit from their acceleration.

To make our claims more concrete, throughout this paper we will use a sentiment analysis prediction model as a motivating example. This model starts from raw sentences and predicts a classification label for each sentence. The model approximately works as follow: the input sentences (strings) are tokenized into words and chars after an initial normalization step. Then two feature vectors are generated as bag-of-words composed by the n-ngrams extracted from word and chars, respectively. The two vectors are then normalized and concatenated into a unique feature vector which is then run through a simple linear regression model. Figure 1 contains the DAG of transformations composing the sentiment analysis example. Figure 2 shows the execution breakdown of this model when scored in IMLT, where the prediction (the LinReg transformation) takes a very small amount of time compared to other components (0.3% of the execution time). Instead, the n-gram transformations take up almost 60% of the execution time, while another 33% is spent in concatenating the feature vectors, which consists in allocating memory and copying data. In summary, most of the execution time is lost in preparing the data for scoring, not in computing the prediction. In our workloads, we observed that this situation is common to many models, because many tasks employ simple prediction models like linear regression functions, both because of their prediction latency, and because of their simplicity and understandability. In these scenarios, we can conclude that the acceleration of an ML pipeline cannot focus solely on specific transformations, as in the case of neural networks where the bulk of computation is spent in scoring, but needs a more systematic approach.

1. We ignore the online learning case here for ease of presentation.

3 Considerations and System Implementation

Following the observations of Section 2, we argue that acceleration of generic ML prediction pipelines is possible if we optimize models end-to-end instead of single transformations, i.e., while data scientist focus on high-level "logical" transformations, the operationalization of the model for scoring should focus on execution units making optimal decisions on how data is processed through model pipelines. We call such execution units *stages*, borrowing this term from the database and system communities. Here, a stage is a sequence of one or more transformations that are executed together on a device (either a CPU or an FPGA, in this work), and represents the "atomic" unit of the computation. No data movement from the computing device to another occurs within a stage, while communication may occur between two stages running on different devices, like the PCIe communication when offloading to FPGA. Within a stage, multiple optimizations are possible, from high-level optimizations (like fusing operators together through data pipelining) to low-level, hardware-related ones (like vectorization for CPUs or task pipelining for FPGAs). The notion of stage is particularly important with regards to FPGA acceleration because it allows accounting for the cost of communication: grouping multiple transformations into a single stage allows trading off the data movement overhead, and makes it possible to devise models of the communication and the execution to be used for scheduling.

In the example of Figure 2 we distinguish three stages:

- In the first stage the input is tokenized and the number of occurrence of the char n-grams found in the sentence is returned as a sparse vector together with its cartesian norm.
- In the second stage, another feature vector is computed out of the bag-of-words representation of the input tokens (coming from the previous stage).
- In the third stage, weighting and normalization occur, and finally the exponentiation is computed; within this stage, it is possible to vectorize the weighting operation and to change the representation of the vector (from sparse to dense) if this makes the computation more efficient

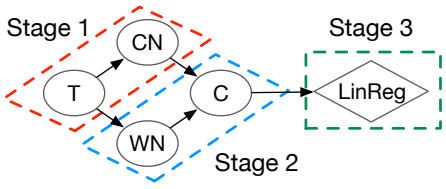

Fig. 3. Stages for the sentiment analysis example pipeline

We optimized these three stages both for CPU execution and for FPGA execution, and give more details on the specific implementations in sec. 3.1 and sec. 3.2, respectively. The model DAG with related stage subdivision is pictorially described in Figure 3.

3.1 CPU Implementation

As a framework written in C# language, IMLT heavily depends on reflection, virtual buffers, and virtual function calls. Users do not need to worry about data types, memory layout, and how the virtual functions are implemented. In this way, users can build their custom pipelines easily, however, we observed that IMLT has to sacrifice performance at scoring time in order to provide such level of generality.

We ran an experiment where the model described in the previous section is first loaded in memory and then scored multiple times. Even when using the same input record, we noticed that large variability in the execution time exists: the first scoring is regularly around 540 times slower than the remaining executions. We refer to each case as *cold* and *hot*, respectively. In both situations, we identified that major bottlenecks were created by (1) the just-in-time (JIT) compiler, (2) the module inferring data type with reflection, (3) virtual function calls; and (4) allocation of memory buffers on demand. To address the first two problem, in the CPU implementation we removed this overhead by rewriting the pipeline code: since the model pipelines are pre-trained, all the type information and data path information can be resolved at compile time. This removes the overhead of dynamic type binding and code compilation. Moreover, we pre-compiled the pipeline into bytecode which reduces the cost of reflection and virtual function call. Furthermore, we created a native image of the pre-compiled code and its dependencies. In this way we avoided all unpredictable performance due to just-in-time compilation.

Finally, we can benefit from information extracted at training time to improve memory management at scoring time, i.e., since the data types in the pipelines are fixed after training, we can estimate the amount of required memory before execution, and create pools of memory buffer which are then accessed when stages are executed.

3.2 FPGA Implementation

In the FPGA implementation, the first stage can greatly benefit from the hardware characteristics. Indeed, the whole stage can be easily parallelized and its sub-phases can be pipelined with each other. In particular, the text normalization can be done immediately before the tokenization in a pipelined fashion, and the tokenization of a long string can be split into multiple parallel tokenizations by allowing some overlap of input characters between consecutive tokenizer units. A tokenizer unit should recognize common punctuation marks, the beginning and end of words and emit a corresponding Murmur hash [8] for each token. To achieve an efficient implementation, we devised a Finite State Machine (FSM) recognizer automaton that is able to take in input one character per clock cycle, recognize the tokens observed and record the corresponding hashes. The design of the tokenizer allows trading between performance and area: if area is limited, a tokenizer can run over more characters and record more tokens at the expense of a higher latency; otherwise, it is possible to limit this latency by using many tokenizers in parallel, with at most one tokenizer per character that can emit at most one hash.

Once the hashes are available (output of stage 1), stage 2 associates each word to a unique number for n-gram extraction, with a dictionary lookup that goes to the off-chip memory. Although this operation is surely expensive, we argue it is no more expensive than on CPU because of the similar hardware for off-chip RAM memory access, and because of the low locality of this transformation that makes caches ineffective. To make this operation as efficient as possible, we increased the number of buckets of the model dictionary in order to limit collisions (which are handled by serializing the data in the buffer) to a pre-defined number, so that the FPGA logic can fetch a fixed amount of data in a burst fashion for every lookup. In pipelines with dictionary lookup, we insert each n-gram identifier in an array, with a parallel lookup to avoid double insertions and increment the counts. This whole phase could be simplified and sped up by using the hash itself as the n-gram identifier and allowing collisions, but this requires retraining the model; as from the assumptions in Section 2, we assume the model is fixed and we have no control on it, and leave this work for the future.

Regarding the FPGA implementation of the third stage, for the weights lookup we used the n-gram identifier to access the value in the off-chip memory, again at the cost of an off-chip memory access. This is due to the size of the weights vector, which cannot stay in the on-chip memory. As before, the sparsity of this computation requires memory accesses on both the FPGA and the CPU.

4 Experimental Evaluation

In this Section we experimentally evaluate the performance gain of our implementations described in Section 3 wrt baseline IMLT. For the CPU implementation, the experiments were run over a Windows Pro machine with an Intel Xeon CPU E5-2620 with 2 processors at 2.10GHz, and 32 GB of RAM. Regarding the FPGA implementation, we used the SDAccel prototyping platform by Xilinx, which abstracts the communication, and we used Xilinx' High Level Synthesis (HLS) tools to generate the accelerator logic. As a device, we used an ADM-PCIE-KU3 board by Alpha Data, with 2 DDR3 memory channels and PCIe x8 connection to the CPU. While the area did not cause major limitations to the design, the number of RAM channels limited the number of parallel dictionary and weights lookups, despite still

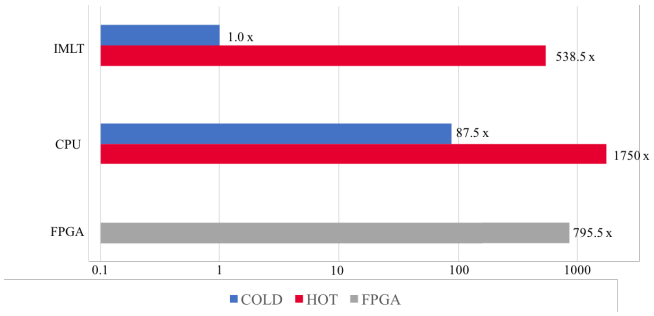

Fig. 4. Performance improvement achieved by CPU and FPGA implementations

providing comparable bandwidth compared to a CPU for reading the input sentence and for the dictionary bursts.

Figure 4 shows the results of our experiments over the sentiment analysis model of Section 2 in terms of performance, reporting the scoring latency of IMLT, of the CPU implementation and of the FPGA implementation. We report the speedup over the prediction latency on both the hot and cold scenarios for the CPU implementation. For the cold scenario we scored one single record, while for the hot scenario we first use one record to worm up the model and then we average the scoring latency of a batch of 9 records. We repeated the experiment 5 times and we report the average speedup. All the results have been normalized with respect to the IMLT prediction latency in the hot scenario, which is our baseline. Figure 4 shows that our CPU implementation achieves 3.3 times improvement over IMLT in the hot scenario, while in the cold scenario it is still 20 times slower, but with an improvement of 87.5x over IMLT cold. These improvements are due to the up-front memory allocation and to the more optimized code, which avoids memory copy and data conversions among transformations within the same stage. Instead, the FPGA implementation (which does not suffer from warm-up delays and is thus shown in a single scenario) achieves a 1.48x speedup over IMLT hot but a 2.3x slowdown over the CPU implementation in the hot case. This result is due to the high penalty of off-chip RAM access, which is higher due the lower operating frequency and to design delays between memory bursts (due to HLS scheduling).

5 Related Work

Two systems for machine learning prediction have been introduced recently in the academia and the industry: Clipper [9] and TensorFlow Serving [10]. Clipper targets high performance online ML prediction while making model deployment easy. Clipper does not consider models as composed by complex DAG of transformations, but instead runs each pipeline as a single functional call in a separate container process. Users can deploy models learned by different frameworks, but this flexibility comes at the cost of losing the control over the execution inside the pipeline which instead relies on the target framework to run the model. Clipper's optimizations thereby focus on models as black boxes: Clipper caches results for popular queries and controls the batch size adaptively to achieve high throughput while achieving low latency SLA. Those optimizations lose the chance to utilize hardware acceleration.

Tensorflow (TF) serving is a library for serving ML models in Tensorflow framework. TF Serving batches multiple prediction requests as Clipper, however, the execution of pipelines is more flexible, allowing users to define custom *Servable* for the part of pipelines. While *Servable*s enable the use of hardware accelerators and make execution faster, there is no framework-level support for fine-grained control over the pipeline execution as introduced in our implementation.

6 Conclusions and Future Works

This work introduced a framework for the acceleration of generic ML prediction pipelines. To avoid runtime overheads and allow hardware-specific optimizations, we introduced the notion of stages, observing that ML transformations usually occur in common sequences that can be organized into atomic execution and scheduling units. Based on this, we implemented a case study in CPU and FPGA, showing noticeable speedups over the initial baseline.

This work paves the way to more research in optimizing ML prediction pipelines. The identification and generation of stages is a promising future work. For the CPU implementation, a given pipeline can be generated automatically starting from the code of its transformations: once the stage division is performed, the stage generator can inline the function calls and aggressively apply thorough optimizations.

In the FPGA case, similar techniques can be applied, possibly with the aid of HLS tools. While a careful design space exploration is fundamental to identify appropriate performance/area trade-offs, higher gains come from appropriately setting the transformations at training time, for example by avoiding the dictionary lookup and allowing the hash conflicts; similarly, caching the model weights on the on-chip memory can enable low-latency, parallel lookups. Modeling these characteristics and properly sizing those stages in order to match a common throughput allows maximizing the final performance.

References

[1] K. Kara, D. Alistarh, G. Alonso, O. Mutlu and C. Zhang, *FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-Off*, FCCM 2017, pp. 160-167.
[2] X. Lin, R.D. Shawn Blanton, and D. E. Thomas, *Random Forest Architectures on FPGA for Multiple Applications*, GLSVLSI 2017, pp. 415-418.
[3] M. Abadi and A. Agarwal et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, white paper, 2015.
[4] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel et al., *Scikit-learn: Machine Learning in Python*, JMLR, volume 12, 2011.
[5] H2O website, `https://www.h2o.ai`
[6] Microsoft Brainwave project announcement, `www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/`
[7] NumPY website, `www.numpy.org/`
[8] MurmerHash (Wikipedia), `https://en.wikipedia.org/wiki/MurmurHash/`
[9] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, Ion Stoica, *Clipper: A Low-Latency Online Prediction Serving System*, NSDI, 2017.
[10] TensorFlow Serving website, `https://www.tensorflow.org/serving/`