2021 31st International Conference on Field-Programmable Logic and Applications (FPL) | 978-1-6654-3759-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/FPL53798.2021.00067

MAFIA: Machine Learning Acceleration on FPGAs for IoT Applications

Nikhil P Ghanathe[1], Vivek Seshadri[2], Rahul Sharma[2], Steve Wilton[1], and Aayan Kumar[3]

[1]*University of British Columbia* [2]*Microsoft Research* [3]*University of California, Berkeley*

***Abstract*—Recent breakthroughs in ML have produced new classes of models that allow ML inference to run directly on milliwatt-powered IoT devices. On one hand, existing ML-to-FPGA compilers are designed for deep neural-networks on large FPGAs. On the other hand, general-purpose HLS tools fail to exploit properties specific to ML inference, thereby resulting in suboptimal performance. We propose MAFIA, a tool to compile ML inference on small form-factor FPGAs for IoT applications. MAFIA provides native support for linear algebra operations and can express a variety of ML algorithms, including state-of-the-art models. We show that MAFIA-generated programs outperform best-performing variant of a commercial HLS compiler by 2.5× on average.**

I. Introduction

Traditionally, IoT devices are used for data collection and the data analysis is performed in the cloud [1]–[3]. However, performing ML inference directly on IoT devices has recently gained attention, offering benefits such as real-time analysis, increased privacy, and reduced energy consumption. Recent breakthroughs have produced new classes of ML applications that have compute and memory requirements low enough to be run directly on milliwatt-scale IoT devices [4], [5] while still having high classification accuracy. While micro-controllers are often used to implement these applications [6], [7], reconfigurable devices like FPGAs may be better suited for such applications. Several reconfigurable architectures for edge computing have been proposed by prior works [8]–[10]. On one hand, as we can configure an FPGA to directly run a program, it can deliver superior performance and energy efficiency compared to general-purpose micro-controllers. On the other hand, unlike ASICs which do not allow updates to algorithms once deployed and incur a high non-recurring engineering (NRE) cost, an FPGA can be reprogrammed in the field using over-the-air updates [11], [12] to accommodate the constant evolution of ML models. Unfortunately, FPGAs are notoriously hard to program, even for experts.

To enable practical use of FPGAs in this domain, we need a compiler that can take high-level specifications of a ML model and directly compile it to an FPGA program. Unfortunately, existing solutions are insufficient in that they mainly focus on running DNN models on large FPGAs. However, state-of-the-art ML models [4], [5] for IoT applications are mostly based on classical ML algorithms [13], and not DNNs. In fact, existing DNN-to-FPGA compilers cannot even express the state-of-the-art ML models designed for IoT applications, and the techniques used by DNN-to-FPGA compilers do not extend to small form-factor FPGAs because they assume an abundance of FPGA resources that allows them to preconfigure the FPGA as, for instance, a sea of matrix multipliers. While general-purpose C-HLS tools can express our target models, they do not exploit specific properties of ML inference and thereby generate suboptimal programs. Section II describes the limitations of these two approaches in more detail.

We propose MAFIA, a compiler for Machine-learning Acceleration on FPGA for IoT Applications. MAFIA provides native support for linear algebra operations and can express classical ML inference algorithms [13] like decision trees [14] and k-nearest neighbors [15]. We currently support a subset of TensorFlow [16] and the framework of a recent prior work, SeeDot [17] that compiles ML inference code to IoT devices (Section III-A). We compare MAFIA with four different tools: A) Bambu HLS, B) Xilinx Vivado HLS, C) Vivado HLS with automatically generated compiler hints, and D) (C) with additional manual hints. We evaluate these mechanisms on two state-of-the-art ML models [4], [5], each on ten different data sets used as benchmark by prior works [4], [5], [17]. MAFIA-generated programs consistently outperform all prior approaches – 2.5× better on average compared to Vivado HLS with automatically generated hints.

II. Limitations of Existing HLS Tools

A. ML-HLS tools target the Neural-network family

Existing ML-HLS tools such as DNN Weaver [18], FP-DNN [19] and DeepBurning [20] specifically target deep neural network (DNN) workloads. There are several reasons these tools do not work well for our target models.

Building Blocks: These tools focus on DNN layers (e.g. conv, pooling, FC). As a result, currently, they cannot express classical algorithms such as decision tree, k-Nearest Neighbors, and kernels like RBF of SVM. Although, we can incorporate support for the classical algorithms in these frameworks, the optimization strategies for DNN's are at odds with classical algorithms (see below). MAFIA identifies that matrix operations are the fundamental building blocks of most classical ML algorithms and expresses the input program as a matrix data flow graph (DFG).

Parallelism: The logical view of a DNN model is serial. Therefore, ML-HLS tools only exploit intra-layer parallelism since opportunities for inter-layer parallelization are often

978-1-6654-3759-2/21/$31.00 ©2021 IEEE
DOI 10.1109/FPL53798.2021.00067

scarce. MAFIA exploits both intra-node and inter-node parallelism (Section IV-A).

Nature of Computation: Most DNN workloads are computation-bound. Therefore, the overhead of data shuffling across layers has minimal impact on latency [21]. However, classical ML algorithms are memory-bound, and data shuffling can significantly increase latency. MAFIA manages this by introducing intelligent design constraints (Section IV-A).

Resource-efficiency: Most ML-HLS tools focus primarily on optimizing matrix multiplication. However, in ML models for IoT applications, the dominant kernel can be any node in the matrix DFG. For example, in the output code of MAFIA, a matrix addition node may be more critical for latency than a matrix multiplication node. MAFIA iteratively optimizes the design based on the criticality of each operation in the program (Section IV-E). Furthermore, existing tools usually target high-end FPGAs and assume abundance of resources. As a result, the optimization techniques employed in these tools are impractical for resource-scarce devices.

B. Limited Scope for Optimizations of C-based HLS tools

Difficult to extract parallelism: Automatically extracting parallelism from a sequential C program is difficult. Most safe compiler transformations make conservative assumptions and yield suboptimal performance. To extract better performance, some HLS tools allow the programmer to annotate the C program with compiler hints. However, this requires expert knowledge, is restrictive in the kind of information that can be provided and is time-consuming.

Difficult to explore solution space: One approach explored by prior work is to automatically generate loop unrolling hints for the HLS compiler. To determine the best unrolling factor for each loop, the compiler must estimate how critical each loop is for the program and the impact of unrolling on resource consumption. This is hard because 1) finding the critical path in a C program requires knowledge of target domain 2) HLS compilers have poor accuracy in predicting (without requiring synthesis and simulation) the resource consumption and critical path latency (Section VI-B).

III. Overview of MAFIA

We propose MAFIA, a compiler that generates high-performance Verilog program from a high-level specification of an ML inference algorithm for a milliwatt-scale FPGA. The heart of the compiler is an optimizer which minimizes the latency of the critical path in the DFG by *determining the level of parallelism with which each node in the program's DFG should be executed*. We formulate this optimization problem as an integer programming problem (Section IV-E).

We augment MAFIA with two components. The first component is a library of Verilog templates with one template for each type of matrix operation. Each template is Verilog implementation of the operation, parameterized by the dimensions of the input matrices and a *parallelism factor* (PF). Our Matrix Template Library currently supports various

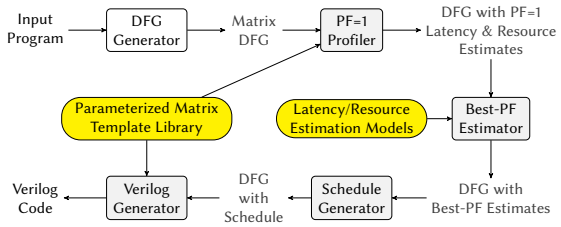

Fig. 1: Overview of MAFIA Flow

dense and sparse matrix operations: sparse matrix-vector multiplication (SpMV), matrix addition/subtraction, dot product, outer product, hadamard product, dense matrix-vector/vector-matrix/matrix-matrix multiplication, scalar-matrix multiplication and non-linear activations including exponentiation, relu, sigmoid and hyperbolic-tan. These operations are sufficient to express a variety of ML models targeting IoT applications [22], including state-of-the-art models [4], [5]. MAFIA's optimizer is agnostic to the exact implementation of the template. Therefore, we can improve performance further by providing an improved template for a particular operation or extend support to other matrix operations by simply adding to the library. The second component is a set of latency/resource estimation models that can predict the amount of FPGA resources and execution latency for an operation given its input dimensions and PF. Since there are finite number of operation types, generating these templates and their regression models is tractable and a one-time-effort during the tool development.

A. MAFIA *Compiler Flow*

Figure 1 shows the compiler's flow. We provide an overview in this section; more details are provided in Section IV.

Given an input program, the *DFG generator* extracts the matrix DFG of the computation along with the matrix dimensions. The *PF-1 Profiler* profiles the resource consumption and latency of each node in the DFG when their PF is set to 1. This information is used by the estimation models to estimate the resource consumption and latency of each node. The *Best-PF Estimator* then uses our optimizer to find the best PF for each node that minimizes the overall latency of the program. The *Scheduler Generator* uses the static DFG to generate a schedule that executes the program in *data flow order*. This allows MAFIA to execute data-independent nodes in parallel. Finally, the *Verilog Generator* uses the identified PFs and the template library to generate the final Verilog code.

IV. Detailed Design

In this section, we describe MAFIA in detail. We begin by describing the *Paramterized Matrix Template Library* and the *Latency/Resource Estimation Models* (yellow blocks in Figure 1). We then explain the flow of the MAFIA compiler.

A. Parameterized Matrix Template Library

At the backend of the MAFIA compiler is a library of hand-optimized Verilog templates, one for each type of matrix operation. Each template consists of two components: 1) the

execution unit and 2) the data interface unit. The execution unit is the core computation unit (eg. adder, multiply-accumulate) with a configurable number of processing elements (PE) specified by the PF. The data interface unit holds logic for supplying data to the execution unit by reading the output memories of the node's predecessors and writing the result generated by the execution unit into the input memories of the node's consumers. The structure of the data interface unit depends both on the execution PF of the node and the PF of the consumer nodes as follows.

Consider two nodes: producer and consumer. In the producer node, each PE writes its partial output to a buffer. However, if the execution PF of producer (i.e. number of PEs) does not match that of the consumer, two sets of buffers with data shuffling logic in between may be needed. For nodes that can complete their execution in linear time, e.g., matrix addition, this data shuffling eliminates any performance benefit gained by parallelizing its execution.

To avoid this problem, we introduce additional constraints on PFs (shown in Figure 2). We classify nodes as: 1) *linear-time nodes*, those that can complete execution in linear time (or less) in terms of the input size (e.g., matrix addition), and 2) *non-linear-time nodes*, those that take worse than linear time to complete execution (e.g., matrix multiplication). We associate each node with multiple PFs: one for each input, one for its execution unit, and one for its output. The figure shows a two-input linear-time node and a two-input non-linear-time node. For a linear-time node, we require the input, execution, and the output PFs to be the same (thereby avoiding the need for any data shuffling). For non-linear time nodes, we introduce a logic before and after the execution to appropriately shuffle the data. Finally, for two nodes with a producer-consumer relationship, the output PF of the producer must be equal to the input PF of the consumer. One natural implication of these constraints is that a sub-graph of connected linear-time nodes will all have the same PFs. We exploit this observation further to enable pipelined execution of linear-time nodes (Section IV-G).

B. Latency/Resource Estimation Models

The key optimization problem in MAFIA is to determine the best PF for each node in the DFG such that the overall latency is minimized, while fitting inside the resource budget of the FPGA. This entails exploration of a vast solution space of possible PF assignments. To reduce exploration time, we build regression models for each template that can predict the resource consumption and latency of a node. The overall resource consumption of a candidate solution is the sum of the resource consumption of all nodes and the execution latency is the sum of the latency of all the nodes in the *critical* path (path with maximum latency). For our target models, we find that the buffering is done mainly using distributed RAM (LUTRAM) because of smaller matrix dimensions. As a result, there are typically more than enough memory resources (BRAM, Flip Flops) on our target FPGA boards, so we focus on predicting only the compute resources (LUT+LUTRAM, DSP).

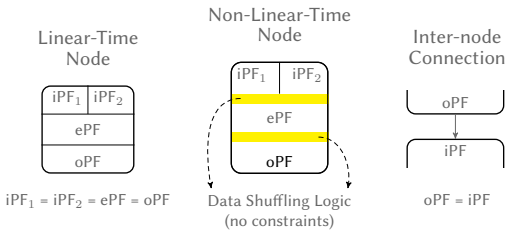

Fig. 2: Linear-time and non-linear-time nodes

We build a separate estimation model for each type of matrix operation. Given the input dimensions of the operation and a target PF, our model estimates the resource consumption (LUTs and DSPs) and execution latency of the operation. In general, we expect LUT consumption to increase linearly with PF. The DSP usage in a PE is known statically for an operation. We expect latency of a node, in general, to reduce linearly as $1/\text{PF}$. However, we find that for some operations, the parallel execution is followed by some linear reduction of the partial sums (e.g., DotProduct). Accordingly, we use the following models for estimating LUT, DSP, and latency consumption of a node. In the equations below, LUT[1] and Latency[1] are the LUT consumption and latency of the corresponding operation with the specified input dimensions when the PF is set to 1.

$$\begin{aligned} \text{Latency}[\text{PF}] &= (\alpha_L + \beta_L \cdot \text{PF} + \gamma_L \cdot \frac{1}{\text{PF}}) \cdot \text{Latency}[1] \\ \text{LUT}[\text{PF}] &= (\alpha_{LUT} + \beta_{LUT}.\text{PF}) \cdot \text{LUT}[1] \\ \text{DSP}[\text{PF}] &= \alpha_{DSP} \cdot \text{PF} \end{aligned}$$

These equations require values for α_{LUT}, β_{LUT}, α_{DSP}, α_L, β_L, and γ_L for each matrix type. Of these, α_{DSP} is set by the template developer to the number of DSPs used by the template's PE. We use a training algorithm to find the remaining parameters. To do this, we generate multiple sets of training data points. Within each set, we fix the input dimensions to an arbitrary value and vary the PF from 1 to a number beyond which the operation cannot be further parallelized by the underlying template. We synthesize and simulate Verilog implementations for each set to obtain the true LUT consumption and execution latency for each operation. We then identify the best parameters that minimize the mean squared error on the training data set. This is an one-time-effort for a family of FPGAs (Artix-7 in our case). These regression models are pre-trained during the tool development and are already included as a part of the MAFIA framework.

C. Data Flow Graph Generator

MAFIA constructs the data flow graph (DFG) of the input program by analyzing the data dependencies in the program. Each node in the DFG is annotated with 1) type of operation, 2) input dimensions for the operation, and 3) any static model parameters for the operation. MAFIA currently includes a DFG generator for SEEDOT DSL [17]. We also support a subset of Tensorflow [16] by converting the Tensorflow program to SEEDOT and extracting the DFG.

D. PF-1 Profiler

The *PF-1 profiler* determines LUT[1] and Latency[1] described earlier. For each node in the program's DFG, it synthesizes the Verilog implementation of the corresponding template with the node's operation dimensions to obtain the value of LUT[1] (once for each node). It then simulates the whole design with a random input to measure the PF=1 latency (one-time). The *PF-1 profiler* tags each node of the DFG with the measured values and passes the DFG to the next stage.

E. Best-PF Estimator

The goal of the Best-PF Estimator is to determine the best PF assignment for each node. The best PF of a node depends on the criticality of the node in the input program, which is influenced by the node's data dimensions. For example, across the 20 datasets we evaluate in Section VI, the PF for the SpMV node generated by the MAFIA optimizer ranges from 3 to 71. This further demonstrates the inefficiency of existing DNN accelerators that optimize for a single type of workload.

Since all the execution PFs are integers, the optimization problem can be expressed as an integer program. We explore two optimization strategies for the best-PF estimator.

1) Black-box optimization: In this approach we employ a generic solver to solve the optimization problem. The critical path of the program can change as PF assignments change. Therefore, we express this optimization metric as a min-max problem: minimize the maximum latency across all paths in the program. Our framework adds constraints for each path in the DFG, stating that the sum of latency of all the nodes for each path should be less than a target latency. The integer program is then setup to minimize this target latency. Since solving an integer program is NP-hard, we relax it to be an optimization problem over real numbers and round the PFs to the nearest integer.

2) Greedy Optimization: As finding a global optimum can be computationally intensive, we also explore a greedy strategy. Our greedy algorithm initializes the PFs of all the nodes to one. It then repeats the following steps while resources are still available.

1) Identify the current critical path of the program.
2) Identify the most critical node in that path. The most critical node is the one which yields the maximum *benefit* when its PF is increased by one subject to the constraints established in Section IV-A. Note that when a node's PF is increased by one, the optimizer may also have to increase the PF of some connected nodes.
3) If the optimizer cannot increase the PF of any node on the critical path, it exits immediately. As MAFIA executes the program in data flow order, there is no benefit to parallelizing a non-critical node even if there are resources available.
4) If not, MAFIA increases the PF of the critical node by one and iterates.

We support two *benefit* metrics: reduction in latency and reduction in latency per additional LUT consumed.

F. Scheduler Generator

MAFIA enables concurrent execution of nodes wherever possible. Each template is associated with a *start* and *done* signal. A node can start execution as soon as its *start* signal is asserted. When a node completes execution, it asserts its *done* signal. MAFIA generates the controller that encodes this scheduling logic for the program's DFG.

G. Pipelining Linear-Time Nodes

When consecutive operations have the same PF, it is possible to view these two nodes as a super node and pipeline their execution (rather than waiting for the first to complete before starting the second). This optimization eliminates the need for memory buffers between pipelined nodes. To perform this optimization, MAFIA identifies clusters of linear-time nodes that are connected to each other and pipelines them. The pipeline begins execution only when *all* the nodes supplying input to the pipeline have completed execution.

V. Evaluation Methodology

A. Hardware Setup and Benchmarks

We target the small form-factor Xilinx Arty-board, which has 20800 logic units (LUTs), 90 DSP (multiplier) slices and 225 KB of on-chip memory operating at 10MHz. For comparison with microcontrollers, we use the results presented by SeeDot [17]. SeeDot targets the Arduino Uno Board (8-bit ATmega328P) with 2KB of SRAM and 32KB of read-only flash operating at 16MHz.

We evaluate different tools using two state-of-the-art machine learning algorithms: Bonsai [5], a decision-tree based classifier, and ProtoNN [4], a k-nearest-neighbour based classifier, from Microsoft's EdgeML library [22]. These algorithms are crafted specifically to run on milliwatt-scale IoT devices and have state-of-the-art accuracies on various tasks. We use ten standard ML datasets (both binary and multi-class): cifar [23], character recognition (cr) [24], usps [25], mnist [26], letter [27] and ward [28]. In total, we evaluate 20 different DFGs as our benchmarks.

B. Comparison Points

We compare MAFIA with four mechanisms that can currently be used for generating Verilog programs from high-level specifications for our target models. We leverage the quantization scheme from a prior work, SeeDot [17]. The fixed-point programs generated from SeeDot are used in our evaluations.

Bambu HLS and Vivado HLS We run Bambu with the `BAMBU-PERFORMANCE-MP` flag (`-O3` optimizations) and Vivado HLS with the default options without any additional compiler hints (denoted by Vivado No Opt. in Figure 3).

Vivado Auto Opt. In this variant, we compare our work against a recent work, SeeDot [17]. SeeDot is a language and a compiler to generate efficient implementation of ML inference algorithms targeting low-end IoT devices. The FPGA backend of SeeDot 1) uses a hand-optimized Verilog implementation for Sparse Matrix Vector Multiplication (the most

time consuming kernel in the microcontroller-based implementations of [4], [5]) with a fixed parallelism factor of 10 and 2) accelerates the rest of the program by automatically annotating the input C program to Vivado HLS with loop unrolling and pipelining hints.

Vivado + MAFIA. To understand the best we could do using a commercial HLS tool, we use the parallelism factors obtained from the MAFIA optimizer to generate appropriate hints to the Vivado HLS compiler. We build this variant on top of Vivado Auto Opt. First, we set the parallelism factor for the hand-optimized SpMV implementation to the PF value of the SpMV node generated by the MAFIA optimizer. Second, we iteratively incorporate the other PFs from the MAFIA optimizer by appropriately unrolling the outermost loop in the C-code of the corresponding operation. We then manually improve the design performance by further unrolling the loops of the program, until the solution runs out of resource budget.

MAFIA. In our evaluations, we extract the DFGs from the SEEDOT [17] framework and use them as input for MAFIA. All our main results are presented with the best performing configuration in which the code is generated by our greedy optimizer (see Section VI-C for more details) with latency reduction per additional LUT as the benefit metric.

VI. RESULTS

We first compare MAFIA-generated programs with other prior approaches on both prediction latency and resource utilization (Section VI-A). Our results show that MAFIA generates programs that reduce prediction latency by 2.5× on average compared to the best previous approach. Figure 4 illustrates the efficiency of MAFIA in allocating resources based on criticality of nodes. This is underlined in our evaluation in Section VI-B, where we observe that our approach is more accurate than existing tools. Finally, in Section VI-C, our evaluation of two optimization strategies, greedy and black-box optimization, reveal that the greedy approach is significantly faster and generates programs of similar quality (if not better) compared to the black-box approach.

A. Comparison of Different Mechanisms

Our results show that all FPGA tools perform comfortably better than programs running on microcontrollers. Vivado No Opt. performs 14× better than microcontroller implementations. However, as expected MAFIA outperforms both Bambu HLS and Vivado No Opt significantly. Therefore, in the rest of the paper, we present results on the following mechanisms: 1) Vivado Auto Opt, 2) Vivado + MAFIA, and 3) MAFIA.

Figure 3 plots the prediction latency of these mechanisms for all twenty benchmarks. The y-axis is in log-scale. Table I lists the number of features for each dataset and their baseline (microcontroller) latencies. Figure 4 shows the average utilization of LUT, DSP, LUT RAM, Flip Flops, and Block RAMs. Since we find memory resources not to be a bottleneck, we focus our attention on LUT and DSP.

1) Vivado Auto Opt.: Vivado Auto Opt improves prediction latency by 7× on average and quadruples the utilization of compute resources compared to Vivado No Opt. by using a hand-optimized implementation for SpMV and adding automatic loop unrolling hints. However, this prior approach is still lacking in two aspects. First, it uses a fixed parallelism factor of 10 for SpMV on all data sets. But, we observe that the criticality of this operation varies significantly with different data sets. As detailed in Section IV-E, the PF for SpMV generated by the MAFIA optimizer ranges from 3 to 71. Second, the mechanism used by prior work to estimate resource utilization had high error rate and results in subpar loop unrolling hints.

2) Vivado + MAFIA.: Vivado + MAFIA involves two steps. First, we incorporate the PFs generated by MAFIA's optimizer in the code generated by Vivado Auto Opt by appropriately setting the PF of the SpMV node and the loop unrolling factors for the other nodes. Although, this improved the performance by 27% on average, we observed that this approach still resulted in suboptimal performance. This is because, *Best-PF Estimator* of MAFIA assumes that the program will be executed in data flow order. However, Vivado HLS does not execute independent nodes in parallel. Therefore, even parallelizing the non-critical nodes in the DFG can enable better performance. We manually unrolled loops for non-critical nodes as well till we hit the resource budget. With this improvement, Vivado + MAFIA outperforms Vivado Auto Opt by up to 3.5× on average across all datasets.

3) MAFIA.: Our analysis of the performance and resource consumption of the different mechanisms show that MAFIA improves upon the state-of-the-art on two fronts. First, MAFIA is able to better estimate required parallelism factors for each node. Second, MAFIA executes the program in data flow order, rather than sequentially. MAFIA outperforms Vivado Auto Opt by 4.2× and even Vivado + MAFIA (with manual hints) by 2.5× despite consuming only half the LUT resources compared to Vivado + MAFIA. This highlights the fact that the performance improvement in C-based HLS tools is only due to intra-node parallelism. With static knowledge of the program's data flow graph, MAFIA is able to schedule operations in data flow order, and thereby execute independent nodes in parallel. Unfortunately, there are no simple hints that can be provided to Vivado HLS to express this opportunity.

DATASET	Num Features	BONSAI Baseline Latency(us)	PROTONN Baseline Latency(us)
cifar-b	400	6121	14112
cr-b	400	6263	28446
mnist-b	784	11568	15983
usps-b	256	4099	9206
ward-b	1000	14733	23241
cr-m	400	29030	34667
curet-m	610	39731	37769
letter-m	16	11161	35377
mnist-m	784	16026	18491
usps-m	256	9140	14017

TABLE I: Baseline (microcontroller) latency and number of features for the evaluated datasets

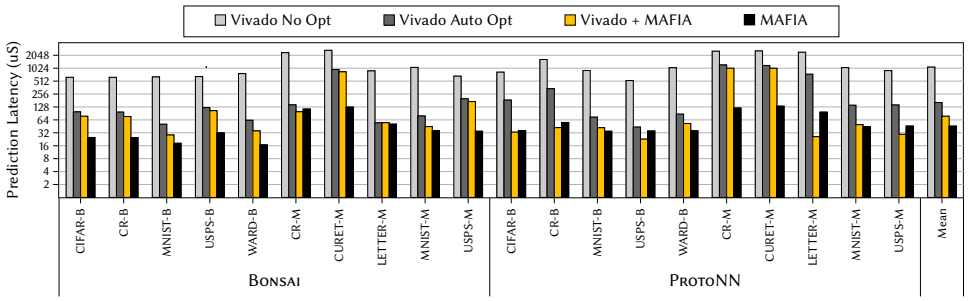

Fig. 3: Prediction latency using different compilers (lower is better). Y-axis is in log-scale.

Also, the hand-optimized implementation of each matrix operation template allows MAFIA to more efficiently perform the underlying arithmetic operations. Therefore, MAFIA exploits both inter-node and intra-node parallelism and can offer higher benefits on more resource constrained FPGAs.

B. MAFIA *Resource/Latency Estimation Models*

The accuracy of MAFIA's estimation models ensures that the optimizer does not overestimate resources and yields suboptimal results by setting lower PFs. The final error of MAFIA estimation models is 36% for LUT, 17% for DSPs, and 99% for overall latency. The high error in latency estimate is in part due to the fact that our estimation algorithm does not capture the pipelining optimization (Section IV-G). Pipelining results in significant reduction in latency while consuming similar amount of compute resources. However, the latency model correctly captures the relative latencies of all nodes, which is sufficient to guide the MAFIA-optimizer. In any case, to put these numbers in context, the estimation error of the Vivado HLS compiler for programs generated by Vivado + MAFIA is 73% for LUT, 673% for DSP, and fails to provide latency estimates.

C. *Greedy vs Black-box optimization*

So far, we have presented results using our greedy optimization strategy. As described in Section IV-E1, we also explore the use of a generic solver. We observe that the prediction latency of programs generated by the greedy approach are on average 10% *lower* than those obtained from the black-box approach (for BONSAI across all data sets). The greedy optimizer is also *significantly* faster than the black-box approach (22× on average). The better performance of the greedy approach can be attributed to our rounding mechanism in the black-box optimization. To ensure that we fit within the resource budget of the FPGA, we round down all the PF numbers obtained from the black-box solver. Optimal rounding is itself an NP-hard problem.

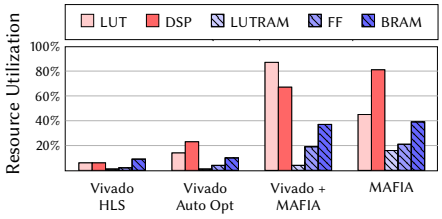

Fig. 4: FPGA resource utilization

Power Consumption: The average power consumption across all implementations (20 datasets) is 76.15 mW.

VII. RELATED WORK

Development of small, accurate ML models for IoT devices is a budding research area [4], [5], [29], [30] with several novel and interesting applications including face-detection, assistance for visual impaired, remote farming and radar classification. We believe MAFIA can allow these applications to be run on similar form-factor FPGA instead of microcontrollers, thereby providing significant boost in the performance of the underlying models. Many *ML-HLS tools* [18], [19], [31]–[36] exist to compile ML inference to FPGAs from popular frameworks like Tensorflow and Pytorch. Unfortunately, all of them target DNN workloads and focus their attention on high-end FPGAs [37]–[40]. Other works on automatically extracting parallelism in general-purpose HLS tools include [41]–[43]. The closest work to MAFIA is SEEDOT [17], a framework that compiles efficient code for microcontrollers and low-end FPGAs from a high-level specification.

VIII. CONCLUSION

ML inference models targeting milli-watt powered IoT devices depart from the DNN architecture and use sophisticated primitives with fewer parameters [4], [5]. However, most current HLS for ML research focuses on neural networks and target high-end FPGAs, resulting in tools that lack the requisite expressiveness. Thus, developers in the IoT space have to rely on general purpose HLS tools that have sufficient expressiveness but suffer from poor resource utilization. MAFIA achieves both high expressiveness and efficiency; it is a fully automatic toolchain that compiles device-agnostic ML algorithms to efficient Verilog code. We have evaluated MAFIA on state-of-the-art ML algorithms in the IoT space and it outperforms prior work by 2.5× on average.

IX. ACKNOWLEDGEMENTS

This work was funded by NSERC through the COHESA strategic network.

REFERENCES

[1] Z. Ji, I. Ganchev, *et al.*, "A cloud-based car parking middleware for IoT-based smart cities: Design and implementation," *Sensors*, vol. 14, no. 12, pp. 22 372–22 393, 2014.

[2] J. Gubbi, R. Buyya, *et al.*, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[3] M. Hassanalieragh, A. Page, *et al.*, "Health monitoring and management using Internet-of-Things (IoT) sensing with cloud-based processing: Opportunities and challenges," in *2015 IEEE international conference on services computing (SCC)*, IEEE, 2015, pp. 285–292.

[4] C. Gupta, A. S. Suggala, *et al.*, "ProtoNN: Compressed and accurate kNN for resource-scarce devices," in *International Conference on Machine Learning*, 2017, pp. 1331–1340.

[5] A. Kumar, S. Goyal, *et al.*, "Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things," in *International Conference on Machine Learning*, 2017, pp. 1935–1944.

[6] S. G. Patil, D. K. Dennis, *et al.*, "Gesturepod: Enabling on-device gesture-based interaction for white cane users," in *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '19, New Orleans, LA, USA: Association for Computing Machinery, 2019, pp. 403–415.

[7] D. Vasisht, Z. Kapetanovic, *et al.*, "FarmBeats: An IoT Platform for Data-Driven Agriculture," in *Networked Systems Design and Implementation (NSDI)*, USENIX, Mar. 2017.

[8] G. Korol, M. G. Jordan, *et al.*, "Mcea: A resource-aware multicore cgra architecture for the edge," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, 2020, pp. 33–39.

[9] C. Tan, A. Kulkarni, *et al.*, "Locus: Low-power customizable many-core architecture for wearables," in *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, 2016, pp. 1–10.

[10] C. Tan, M. Karunaratne, *et al.*, "Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 575–587.

[11] T. Gomes, F. Salgado, *et al.*, "Towards an fpga-based network layer filter for the internet of things edge devices," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–4.

[12] J. Jung, J. Ryoo, *et al.*, "Gateway over the air: Towards pervasive internet connectivity for commodity iot," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '20, Toronto, Ontario, Canada: Association for Computing Machinery, 2020, pp. 54–66.

[13] T. M. Mitchell, *Machine learning, International Edition*, ser. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[14] J. R. Quinlan, "Induction of Decision Trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986.

[15] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[16] M. Abadi, A. Agarwal, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. eprint: 1603.04467.

[17] S. Gopinath, N. Ghanathe, *et al.*, "Compiling KB-sized Machine Learning Models to Tiny IoT Devices," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA, 2019, pp. 79–95.

[18] H. Sharma, J. Park, *et al.*, "From High-level Deep Neural Models to FPGAs," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49, Taipei, Taiwan, 2016, 17:1–17:12.

[19] Y. Guan, H. Liang, *et al.*, "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2017, pp. 152–159.

[20] Y. Wang, J. Xu, *et al.*, "Deepburning: Automatic generation of fpga-based learning accelerators for neural network family," in *53nd ACM/EDAC/IEEE Design Automation Conference*, 2016, pp. 1–6.

[21] N. K. Jha and S. Mittal, "Modeling data reuse in deep neural networks by taking data-types into cognizance," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[22] Dennis, Don Kurian and Gopinath, Sridhar and Gupta, Chirag and Kumar, Ashish and Kusupati, Aditya and Patil, Shishir G and Simhadri, Harsha Vardhan, *EdgeML: Machine Learning for resource-constrained edge devices*, version 0.1.

[23] A. Krizhevsky, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[24] T. E. de Campos, B. R. Babu, *et al.*, "Character Recognition in Natural Images," in *VISAPP 2009 - Proceedings of the Fourth International Conference on Computer Vision Theory and Applications, Lisboa, Portugal, 2009, Vol. 2*, 2009, pp. 273–280.

[25] J. J. Hull, "A database for handwritten text recognition research," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 16, no. 5, pp. 550–554, 1994.

[26] Y. LeCun, L. Bottou, *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[27] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass support vector machines," *IEEE transactions on Neural Networks*, vol. 13, no. 2, pp. 415–425, 2002.

[28] J. Yang, Y. Li, *et al.*, "Group-sensitive multiple kernel learning for object categorization," in *Computer Vision, 2009 IEEE 12th International Conference on*, IEEE, 2009, pp. 436–443.

[29] D. Dennis, C. Pabbaraju, *et al.*, "Multiple Instance Learning for Efficient Sequential Data Classification on Resource-constrained Devices," in *Proceedings of the Thirty-first Annual Conference on Neural Information Processing Systems (NeurIPS)*, slides/DennisPSJ18.pdf, 2018, pp. 10 976–10 987.

[30] A. Kusupati, M. Singh, *et al.*, "FastGRNN: A Fast, Accurate, Stable and Tiny Kilobyte Sized Gated Recurrent Neural Network," in *Proceedings of the Thirty-first Annual Conference on Neural Information Processing Systems (NeurIPS)*, slides/fastgrnn.pdf, 2018, pp. 9031–9042.

[31] D. Holanda Noronha, K. Gibson, *et al.*, "LeFlow: Automatic Compilation of TensorFlow Machine Learning Applications to FPGAs," in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 393–396.

[32] S. Mouselinos, V. Leon, *et al.*, "TF2FPGA: A Framework for Projecting and Accelerating Tensorflow CNNs on FPGA Platforms," in *2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2019, pp. 1–4.

[33] J. Duarte, S. Han, *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 7, P07027, Jul. 2018. arXiv: 1804.06913 [physics.ins-det].

[34] X. Zhang, J. Wang, *et al.*, "DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[35] J. Fowers and et. al., "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2018, pp. 1–14.

[36] S. Hussain, M. Javaheripi, *et al.*, "FastWave: Accelerating Autoregressive Convolutional Neural Networks on FPGA," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019.

[37] X. Lian, Z. Liu, *et al.*, "High-Performance FPGA-Based CNN Accelerator With Block-Floating-Point Arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1874–1885, 2019.

[38] Y. Sun, B. Liu, *et al.*, "An OpenCL-Based Hybrid CNN-RNN Inference Accelerator On FPGA," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 283–286.

[39] M. Hardieck, M. Kumm, *et al.*, "Reconfigurable Convolutional Kernels for Neural Networks on FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 43–52.

[40] D. J. Moss, S. Krishnan, *et al.*, "A Customizable Matrix Multiplication Framework for the Intel HARPv2 Xeon+FPGA Platform: A Deep Learning Case Study," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18, Monterey, CALIFORNIA, USA, 2018, pp. 107–116.

[41] W. Zuo, Y. Liang, *et al.*, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13, Monterey, California, USA, 2013, pp. 9–18.

[42] G. Natale, G. Stramondo, *et al.*, "A polyhedral model-based framework for dataflow implementation on fpga devices of iterative stencil loops," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8.

[43] J. Liu, J. Wickerson, *et al.*, "Polyhedral-based dynamic loop pipelining for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1802–1815, 2018.