BraggHLS

Maksim Levental
University of Chicago
Email: test@test.tes

Ryan Chard
Ecole Superieure
Nantes, France
Email: second@second.fr

Kyle Chard
and Ian Foster
Star Academy
San Francisco, California 99999-9999
Telephone: (800) 555–5555
Fax: (888) 555–5555

***Abstract*—In many experiment-driven scientific domains, such as high-energy physics, material science, and cosmology, very high data rate experiments impose hard constraints on the corresponding data acquisition systems: collected data must either be indiscriminately stored for post-processing and analysis, thereby necessitating large storage capacity, or accurately filtered in real-time, thereby necessitating low latency execution. Deep neural networks, effective in many other filtering tasks, have not been widely employed in such data acquisition systems, due to design and deployment difficulties. This paper presents an open source, lightweight, compiler framework `BraggHLS`, based on high-level synthesis techniques, for translating high-level representations of deep neural networks to low-level representations, suitable for deployment to near-sensor devices such as field-programmable gate arrays. We present a case study and evaluation of `BraggHLS` on a deep neural network for Bragg peak detection in the context of high-energy diffraction microscopy. We show `BraggHLS` is able to produce an implementation with a throughput 4.7µs/sample, which is approximately a 5x improvement over the existing implementation.**

Contents

I Introduction 1

II Background 2
- II-A Compilers: the path from high to low . . . 2
  - II-A1 PyTorch and TorchScript 2
  - II-A2 MLIR 3
- II-B High-level synthesis and FPGA design . . . 3
  - II-B1 High-level synthesis 3
- II-C FPGA design 4

III `BraggHLS` compiler and HLS framework 4

IV Evaluation 4

V Conclusion 4

References 4

I. Introduction

Very high data rates are observed and, consequently, large datasets are generated across a broad range of experiments in scientific domains, such as high-energy physics, material science, and cosmology. For example, in high-energy physics, the LHCb detector, at the CERN Large Hadron Collider, is tasked with observing the trajectories of particles produced in proton-proton collisions at a rate of 40 million per second (i.e., 40 MHz) [1]. With a packet size of approximately 50kB (per collision), this implies a data rate of approximately 2TB/s. Ultimately, in combination with other detectors, the LHC processes approximately 100EB of data a year. In materials science, high-energy diffraction microscopy (HEDM) techniques, which provide non-destructive characterization of structure and its evolution in a broad class of single-crystal and polycrystalline materials, can have collection rates approaching 1 MHz [2], with a corresponding packet size of 80kB. In cosmology, the Square Kilometre Array, a radio telescope projected to be completed in 2024 and to be operational by 2027 [3], will sustain data rates in excess of 10 TB/s [4].

Naturally, for high data rate experiments, directly storing and distributing such large quantities of data to the associated research communities for further analysis is cost prohibitive. Thus, either compression (in the case of storage and transmission) or outright filtering is necessary, i.e., only a small fraction of the most "interesting" data is selected at time of collection, with the remainder being permanently discarded. In this work we focus on the filtering approach. Note, that the tradeoff made in employing filtering should be clear: reduced storage at the expense of more stringent latency constraints (on the filtering mechanisms). In addition, the risk of discarding meaningful data introduces accuracy (of the filtering mechanisms) as a critical new dimension of the data acquisition systems. Typically, these filtering mechanisms consist either of physics based models [5] or machine learning models [6]; in either case maximally efficient and effective use of the target hardware platform is tantamount to accuracy. Irrespective of the type of technique employed, almost universally, for the ultra-low latency use cases (e.g., sub-microsecond latency constraints), the implementation is deployed to either field-programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs) [7].

Deep neural networks (DNNs), a particular type of machine learning model, have been shown to be effective in many scientific and commercial domains due to their "representational capacity", i.e., they demonstrate a capacity to (approximately) represent diverse sets of mappings [8]. DNNs "learn" to represent a mapping over the course of "training", wherein they are iteratively evaluated on sample data while a "learning rule" periodically updates the parameters (*weights*) that parameterize the DNN. In recent years they have been investigated for near real-time scientific use cases [9], [10], [11] but their use for the lowest latency use cases has been very limited [7]. The reasons for this are threefold:

1) Graphics Processing Units (GPUs), the conventional hardware target for DNNs, until very recently, have not been performant enough for these very high data rate, very low latency, use cases (due to their low clock speeds and low peripheral bandwidth [12]);
2) DNNs, by virtue of their depth, are resource intensive, in terms of both memory (for the weights) and compute (floating point arithmetic), thereby preventing their deployment to FPGAs, which, in particular, have limited static RAM available;
3) DNNs are (typically) defined, trained, and distributed using high-level frameworks (such as PyTorch [13], TensorFlow [14], MXNet [15]), which abstract all implementation details from the user, thereby making portability of existing model architectures (to e.g., FPGA) nigh impossible.

These three barriers demand of a solution that can simultaneously translate a high-level representation of a DNN to a low-level representation, suitable for deployment to FPGA, while optimizing resource usage and minimizing latency. In general, the task of *lowering* high-level representations of programs to lower-level representations is the domain of a compiler. Similarly, the task of *synthesizing* a *register-transfer level* (RTL) *design*, rendered in a *hardware description language* (HDL), from a program, is the domain of high-level synthesis (HLS) [16]. While several such HLS tools exist [17], [18], [19] and despite, often, bundling robust optimizing compilers, they struggle to effectively perform the necessary optimizations in reasonable amounts of time (see Section IV).

Recently, deep learning compilers (such as TVM [20], MLIR [21], and Glow [22]) have demonstrated the ability to dramatically reduce inference latencies [23], training times [24], and memory usage [25] of DNNs. These compilers function by extracting intermediate-level representations (IRs) of the DNNs, from the representations produced by the frameworks, and performing various optimizations on those IRs (such as kernel fusion [26], vectorization [27], and memory planning [25]). The highly optimized IR is then used to generate code for various target hardware platforms. Given the successes of these compilers, it's natural to wonder whether they can adapted to the task of sufficiently optimizing a DNN such that it might be synthesized to RTL, for deployment to FPGA.

In this paper, we present `BraggHLS`, an open source, lightweight, compiler and HLS framework which can lower DNNs defined as PyTorch models to FPGA implementations. `BraggHLS` uses a combination of compiler and HLS techniques to compile the entire DNN into a *statically scheduled* circuit, thereby eliminating all synchronization overheads and achieving ultra-low latency. `BraggHLS` is general and supports a wide range of DNN layer types, and thus a wide range of DNNs, but we evaluate it on a DNN designed for identifying Bragg diffraction peaks. In summary our specific contributions include:

1) We discuss the challenges faced by a compiler and HLS tool in attempting to lower DNNs to ultra-low latency designs, including runtime costs incurred during design space exploration, challenges meeting resource and timing constraints during synthesis, placement, and routing;
2) We describe and implement a compiler framework, `BraggHLS`, which can effectively transform unoptimized, hardware-agnostic PyTorch models into ultra-low latency RTL designs suitable for deployment to Xilinx FPGAs. `BraggHLS` is thoroughly tested, open source, and available at https://github.com/makslevental/bragghls/;
3) We show that designs generated by `BraggHLS` achieve lower latency than Xilinx's state-of-the-art commercial HLS tool (Vitis HLS) for a variety of DNN layer types. In particular we show that `BraggHLS` can produce synthesizable designs that meet placement, routing, and timing constraints, where Vitis HLS cannot.

The rest of this paper is organized as follows: Section II reviews key concepts from compilers, high-level synthesis, and FPGA design flows. Section III describes the `BraggHLS` compiler and HLS framework in detail. Section IV describes `BraggNN`, the Bragg peak detection DNN, and evaluates `BraggHLS`'s resource efficiency, scalability, and competitiveness with designs generated by Vitis HLS. Finally, Section V concludes with a summary, and related and future work.

II. Background

A. Compilers: the path from high to low

The path from a high-level, abstract, representations of a DNN to a register-transfer level representation can be neatly formulated as a series of progressive lowerings between adjacent levels of abstraction. Each level of abstraction is rendered as a programming language, IR, or HDL and thus we descibe each lowering in terms these representations and the tools that manipulate them:

1) An imperative, *define-by-run,* Python representation, in PyTorch;
2) High-level data-flow graph representation, in TorchScript;
3) Low-level data and control flow graph representation, in MLIR.

1) PyTorch and TorchScript: Typically DNN models are represented in terms of high-level frameworks, themselves implemented within general purpose programming languages. Such frameworks are widely used because of their ease of use and large library of example implementations of various DNN model architectures. Since `BraggNN` is implemented using PyTorch, we focus on relevant aspects of PyTorch. DNNs developed within PyTorch are *defined-by-run*: the author imperatively describes the DNN in terms of high-level operations, using python, which when executed materializes the high-level data-flow graph (DFG) corresponding to the DNN (e.g., for the purposes of reverse-mode automatic differentiation). From the perspective of the user, define-by-run enables fast iteration

at development time, possibly at the cost of some runtime performance.

From the perspective of compilation, define-by-run precludes efficient extraction of the high-level DFG; since the DFG is materialized only at runtime, it cannot be inferred from the textual representation (i.e., the python source) of the DNN. Furthermore, apriori, the runtime-materialized DFG is only partially materialized[1], and only as an in-memory data structure. Thus, framework support is necessary. Indeed, PyTorch supports a Single Static Assignment (SSA) IR, called TorchScript (TS) IR and accompanying tracing mechanism (the TS JIT) to produce TS IR from conventionally defined PyTorch models. Lowering from PyTorch to TS IR enables various useful analyses and transformations on a DNN at the level of the high-level DFG (such as kernel fusion [26]) but targeting FPGAs requires a broader collection of transformations. To this end, we turn to a recent addition to the compiler ecosystem.

2) MLIR: Multi-level Intermediate Representation [21] presents a new approach to building reusable and extensible compiler infrastructure. MLIR is composed of a set of *dialect* IRs, subsets of which are mutually compatible, either outright or by way of translation/legalization. The various dialects aim to capture and formalize the semantics of compute intensive programs at varying levels of abstraction, as well as namespace related sets of IR transformations. The entrypoint into this compiler framework, from PyTorch, is the `torch` dialect [28], a high-fidelity mapping from TS IR to MLIR native IR, which, in addition to performing the translation to MLIR, fully refines all shapes of intermediate tensors in the DNN (i.e., computes concrete values for all dimensions of each tensor); this is necessary for downstream optimizations and eliminating inconsistencies in the DNN [29].

While the `torch` dialect is necessary for lowering to MLIR and shape refinement, it is a representation of a DNN at the same level of abstraction as TS IR: it does not capture the precise data flow and control flow necessary for novel implementations of DNN operations (e.g., for FPGA). Fortunately, MLIR supports lower-level dialects, such as the `affine` and `scf` (structured control flow) dialects. The `scf` dialect is a straightforward formalization of control flow primitives, such as conditionals and loops, so we do not discuss it in great detail. The `affine` dialect, on the otherhand, provides a formalization of semantics that lend themselves to polyhedral compilation techniques [30], i.e., techniques that enable loop dependence analysis and loop transformations. We discuss the importance of loop transformations in the following section.

B. High-level synthesis and FPGA design

1) High-level synthesis: High-level synthesis tools produce RTL descriptions of digital designs from high-level representations, such as C or C++ [17], [19] or LLVM IR. In particular, Xilinx's Vitis HLS, based on the Autopilot project [18], recently enabled passing LLVM IR to the tool, rather than C/C++. Given a high-level, procedural, representation, HLS proceeds in three steps, in order to produce a corresponding RTL design:

1) HLS schedules operations (such as `fmul`, `fadd`, `load`, `store`) in order to determine which operations should occur during each clock-cycle. Such a schedule depends on three characteristics of the high-level representation:
   a) The topological ordering of the DFG/CFG of the procedural representation (i.e., the dependencies of operations on results of other operations and resources);
   b) The completion time for each operation;
   c) The user's desired clock rate/frequency;
2) HLS associates operations to particular RTL instantiations (called *binding*) for those operations; for example whether to associate an add operation followed by a multiply operation to two separate instances, or whether to associate them both with a single instance (e.g., configured to perform a fused-multiply-add);
3) HLS builds a finite-state machine (FSM) that implements the schedule of operations as control logic, i.e., logic that initiates operations and routes signals between them during the appropriate FSM stages.

In addition to fulfilling these three fundamental tasks, high-level synthesis aims to optimize the program, during synthesis. In particular, they try to maximize concurrency and parallelism (number of concurrent operations scheduled during a clock-cycle) in order maximize the throughput and minimize the latency of the final implementation.

Maximizing parallelism entails rigorous data-flow analysis in order to identify data dependencies that would lead to data hazards in synthesized designs. This data-flow analysis exhibits extremely high runtime as lower latency designs are pursued. This can be understood in terms of loop-nest representations of DNN operations; for example consider a convolution as in listing 1. Parallel schedules of the arithmetic operations for this loop nest can be computed by first unrolling all the loops up to some "trip-count" and then computing the topological sort of the operations. The degree to which the loops are unrolled determines how many arithmetic operations can be scheduled in parallel. The issue is that the stores and loads on the `output` array prevent reconstruction of explicit relationships between the inputs and outputs of the arithmetic operations across loop iterations. The standard resolution is to perform *store-load forwarding*: pairs of store and load operations to/from the same memory address are eliminated, with the operand of the store forwarded to the uses of the load (see listing 2). In order for this transformation to be correct (preserve program semantics), for each pair of candidate store and load operations, it must be verified that there are no intervening memory operations. Note, the number of such checks scales polynomially in the parameters of the convolution since the loop nest unrolls into $b \times c_{out} \times (h-2) \times (w-2) \times c_{in} \times k^2$ store-load pairs. Note also, while in the case of listing 1

[1] "...instead, every intermediate result records only the subset of the computation graph that was relevant to their computation." [13]

```
def conv2d(
  input = array(b, c_in, h, w),
  output = array(b, c_out, h − 2, w − 2),
  weight = array(c_out, c_in, k, k)
):
  for iv1 in range(0, b):
    for iv2 in range(0, c_out):
      for iv3 in range(0, h − 2):
        for iv4 in range(0, w − 2):
          for iv5 in range(0, c_in):
            for iv6 in range(0, k):
              for iv7 in range(0, k):
                _3 = (iv3 + iv6)
                _4 = (iv4 + iv7)
                _5 = input[iv1, iv5, _3, _4]
                _6 = weight[iv2, iv5, iv6, iv7]
                _7 = output[iv1, iv2, iv3, iv4]
                _8 = _5 * _6
                _9 = _7 + _8
                output[iv1, iv2, iv3, iv4] = _9
```

Listing 1: Padding 1, stride 1, c_{out} filter convolution with $k \times k$ kernel applied to (b, c_{in}, h, w)-dimensional `input` tensor, where b is the batch size, c_{in} is the number of channels, and (h, w) are the height and width, respectively.

```
1  def conv2d(
2    input = array(b, c_in, h, w),
3    output = array(b, c_out, h − 2, w − 2),
4    weight = array(c_out, c_in, k, k)
5  ):
6    for iv1 in range(0, b):
7      for iv2 in range(0, c_out):
8        for iv3 in range(0, h − 2):
9          for iv4 in range(0, w − 2):
10           ...
11           # e.g., iv5, iv6, iv7 = 2, 3, 4
12           _31 = (iv3 + iv6)
13           _41 = (iv4 + iv7)
14           _51 = input[iv1, iv5, _31, _41]
15           _61 = weight[iv2, iv5, iv6, iv7]
16           _71 = output[iv1, iv2, iv3, iv4]
17           _81 = _51 * _61
18           _91 = _71 + _81
19           output[iv1, iv2, iv3, iv4] = _91
20           # iv5, iv6, iv7 = 2, 3, 5
21           _32 = (iv3 + iv6)
22           _42 = (iv4 + iv7)
23           _52 = input[iv1, iv5, _32, _42]
24           _62 = weight[iv2, iv5, iv6, iv7]
25           _72 = output[iv1, iv2, iv3, iv4]
26           _82 = _52 * _62
27           _92 = _72 + _82
28           output[iv1, iv2, iv3, iv4] = _92
29           ...
```

Listing 2: Store-load forwarding across successive iterations (e.g., `iv7` = 4, 5) of the inner loop in listing 1, after unrolling. The forwarding opportunity is from the store on line 19 to the load on line 25; both can be eliminated and `_91` can replace uses of `_72`, such as in the computation of `_92` (and potentially many others).

the verification is straightforward, in general it might involve solving a small constraint satisfaction program [31].

Finally, note, though greedy solutions to the scheduling problem solved by HLS are possible, in principle scheduling is an integer linear programming problem (ILP), instances of which are NP-hard. In summary, HLS tools solve computationally intensive problems in order to produce a RTL description of a high-level representation of a DNN. These phases of the HLS process incur "development time" costs (i.e., runtime of the tools) and impose practical limitations on the amount of design space exploration (for the purpose of achieving latency goals) which can be performed. `BraggHLS` addresses these issues by enabling the user to employ heuristics (during both the parallelization and scheduling phases) which, while not guaranteed to be correct, can be *behaviourally verified*.

C. FPGA design

At the RTL level of abstraction, there remain two more steps prior to being able to actually deploy to an FPGA; one of them being a final lowering, so called logic synthesis, and the other being place and route (P&R). Logic synthesis is the process of mapping RTL to actual hardware primitives on the FPGA (so-called *technology mapping*), such as lookup tables (LUTs), block RAMs (BRAMs), flip-flops (FFs), and digital signal processors (DSPs). Logic synthesis produces a network list (*netlist*) describing the logical connectivity of various parts of the design. Logic synthesis effectively determines the implementation of floating point operations in terms of DSPs; depending on user parameters and other design features, DSP resource consumption for floating point multiplication and addition can differ greatly. The number of LUTs and DSPs that a high-level representation of a DNN corresponds to is relevant to both the performance and feasibility of that DNN when deployed to FPGA.

After the netlist has been produced, the entire design undergoes P&R. The goal of P&R is to determine which configurable logic block within an FPGA should implement each of the units of logic required by the digital design. P&R algorithms need to minimize distances between related units of functionality (in order to minimize wire delay), balance wire density across the entire fabric of the FPGA (in order to reduce route congestion), and maximize the clock speed of the design (a function of both wire delay, logic complexity, and route congestion). The final, routed design, can then be deployed to the FPGA by producing a proprietary *bitstream*, which is written to the FPGA.

III. BraggHLS compiler and HLS framework

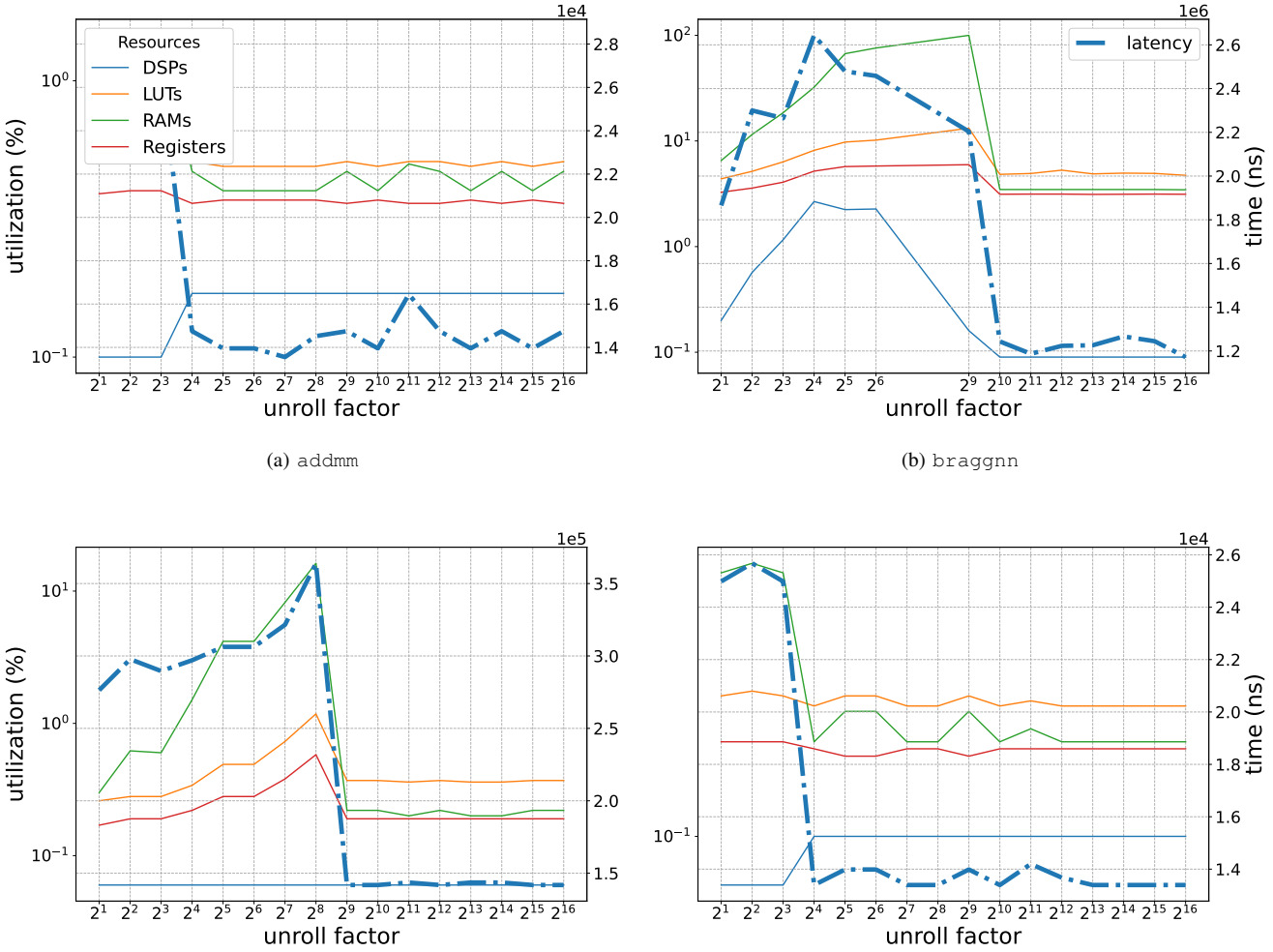
IV. Evaluation

asdasd

V. Conclusion

References

[1] V. Gligorov, "Real-time data analysis at the lhc: present and future," in *Proceedings of the NIPS 2014 Workshop on High-energy Physics and Machine Learning*, ser. Proceedings of Machine Learning Research,



(a) `addmm`

(b) `braggnn`

(c) `conv`

(d) `matmul`

Fig. 1. Resource usage and latency vs. unroll factor of various DNN modules.

G. Cowan, C. Germain, I. Guyon, B. Kegl, and D. Rousseau, Eds., vol. 42. Montreal, Canada: PMLR, 13 Dec 2015, pp. 1–18. [Online]. Available: https://proceedings.mlr.press/v42/glig14.html

[2] M. Hammer, K. Yoshii, and A. Miceli, "Strategies for on-chip digital data compression for x-ray pixel detectors," *Journal of Instrumentation*, vol. 16, no. 01, pp. P01 025–P01 025, jan 2021. [Online]. Available: https://doi.org/10.1088%2F1748-0221%2F16%2F01%2Fp01025

[3] J. McMullin, P. Diamond, M. Caiazzo, A. Casson, T. Cheetham, P. Dewdney, R. Laing, B. Lewis, A. Schinckel, L. Stringhetti *et al.*, "The square kilometre array project update," in *Ground-based and Airborne Telescopes IX*, vol. 12182. SPIE, 2022, pp. 263–271.

[4] K. Grainge, B. Alachkar, S. Amy, D. Barbosa, M. Bommineni, P. Boven, R. Braddock, J. Davis, P. Diwakar, V. Francis *et al.*, "Square kilometre array: The radio telescope of the xxi century," *Astronomy reports*, vol. 61, no. 4, pp. 288–296, 2017.

[5] "Comparison of particle selection algorithms for the LHCb Upgrade," 2020. [Online]. Available: https://cds.cern.ch/record/2746789

[6] V. V. Gligorov and M. Williams, "Efficient, reliable and fast high-level triggering using a bonsai boosted decision tree," *Journal of Instrumentation*, vol. 8, no. 02, pp. P02 013–P02 013, feb 2013. [Online]. Available: https://doi.org/10.1088%2F1748-0221%2F8%2F02%2Fp02013

[7] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, pp. P07 027–P07 027, jul 2018. [Online]. Available: https://doi.org/10.1088%2F1748-0221%2F13%2F07%2Fp07027

[8] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions," *Journal of big Data*, vol. 8, no. 1, pp. 1–74, 2021.

[9] Z. Liu, T. Bicer, R. Kettimuthu, and I. Foster, "Deep learning accelerated light source experiments," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 2019, pp. 20–28.

[10] R. M. Patton, J. T. Johnston, S. R. Young, C. D. Schuman, D. D. March, T. E. Potok, D. C. Rose, S.-H. Lim, T. P. Karnowski, M. A. Ziatdinov *et al.*, "167-pflops deep learning for electron microscopy: from learning physics to atomic manipulation," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 638–648.

[11] Y. Liu, K. P. Kelley, H. Funakubo, S. V. Kalinin, and M. Ziatdinov, "Exploring physics of ferroelectric domain walls in real time: deep learning enabled scanning probe microscopy," *Advanced Science*, p. 2203957, 2022.

[12] R. Aaij, J. Albrecht, M. Belous, P. Billoir, T. Boettcher, A. Brea Rodríguez, D. Vom Bruch, D. Cámpora Pérez, A. Casais Vidal, D. Craik *et al.*, "Allen: A high-level trigger on gpus for lhcb," *Computing and Software for big Science*, vol. 4, no. 1, pp. 1–11, 2020.

[13] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin,

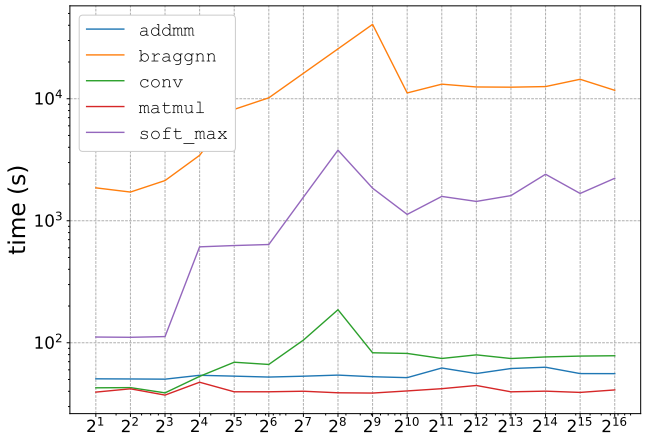


Fig. 2. Runtime of Vitis HLS vs. unroll factor.

A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," 2017.

[14] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2016. [Online]. Available: https://arxiv.org/abs/1603.04467

[15] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015. [Online]. Available: https://arxiv.org/abs/1512.01274

[16] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[17] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, sep 2013. [Online]. Available: https://doi.org/10.1145/2514740

[18] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, *AutoPilot: A Platform-Based ESL Synthesis System*. Dordrecht: Springer Netherlands, 2008, pp. 99–112. [Online]. Available: https://doi.org/10.1007/978-1-4020-8588-8_6

[19] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo, "Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1327–1330.

[20] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[21] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: A compiler infrastructure for the end of moore's law," 2020. [Online]. Available: https://arxiv.org/abs/2002.11054

[22] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph lowering compiler techniques for neural networks," 2018. [Online]. Available: https://arxiv.org/abs/1805.00907

[23] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing cnn model inference on cpus," 2018. [Online]. Available: https://arxiv.org/abs/1809.02697

[24] S. Zheng, R. Chen, Y. Jin, A. Wei, B. Wu, X. Li, S. Yan, and Y. Liang, "Neoflow: A flexible framework for enabling efficient compilation for high performance dnn training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 3220–3232, nov 2022.

[25] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," 2016. [Online]. Available: https://arxiv.org/abs/1604.06174

[26] A. Ashari, S. Tatikonda, M. Boehm, B. Reinwald, K. Campbell, J. Keenleyside, and P. Sadayappan, "On optimizing machine learning workloads via kernel fusion," vol. 50, no. 8, pp. 173–182, jan 2015. [Online]. Available: https://doi.org/10.1145/2858788.2688521

[27] S. Maleki, Y. Gao, M. J. Garzar, T. Wong, D. A. Padua *et al.*, "An evaluation of vectorizing compilers," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 372–382.

[28] S. Silva and A. Elangovan, "Torch-MLIR," https://mlir.llvm.org/OpenMeetings/2021-10-07-The-Torch-MLIR-project.pdf, 2021.

[29] M. Hattori, N. Kobayashi, and R. Sato, "Gradual tensor shape checking," 2022. [Online]. Available: https://arxiv.org/abs/2203.08402

[30] U. Bondhugula, "Polyhedral Compilation Opportunities in MLIR," https://acohen.gitlabpages.inria.fr/impact/impact2020/slides/IMPACT_2020_keynote.pdf, 2020.

[31] S. V. Rajopadhye, "Dependence analysis and parallelizing transformations." 2002.