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 evaluate `BraggHLS` on various workloads and present a case-study deep neural network for Bragg peak detection in the context of high-energy diffraction microscopy. We show `BraggHLS` is able to produce an implementation of the network with a throughput 4.8 µs/sample, which is approximately a 4× improvement over the existing implementation.**

Contents

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 50 kB (per collision), this implies a data rate of approximately 2 TB/s. Ultimately, in combination with other detectors, the LHC processes approximately 100 EB 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 80 kB. In cosmology, the Square Kilometer 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 particularly focus 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/maksleventa l/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 evaluates `BraggHLS`'s performance, scalability, and competitiveness with designs generated by Vitis HLS. Section V describes our case study, `BraggHLS` applied to `BraggNN`, a Bragg peak detection DNN with a target latency of 1µs/sample. Finally, Section VI 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 describe each lowering in terms the representations and tools `BraggHLS` employs:

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. `BraggHLS` is implemented using PyTorch, thus 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 `linalg`, `affine` and `scf` (structured control flow) dialects. The `scf` dialect is a straightforward formalization of control flow primitives, such as conditionals and loops. The `affine` dialect, on the other hand, provides a formalization of semantics that lend themselves to polyhedral compilation techniques [30], i.e., techniques that enable loop dependence analysis and loop transformations. Such loop transformations, particularly loop unrolling, are crucial for achieving lowest possible latencies [31].

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

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]. In particular, Xilinx's Vitis HLS, based on the Autopilot project [18], is a state-of-the-art HLS tool. 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 `mulf`, `addf`, `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-accumulate (MAC);
   a) In the case of floating point arithmetic operations, HLS also (with user guidance) determines the precision of the floating point representation;
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 concurrency entails pipelining operations: operations are executed such that they overlap in time, subject to available resources. Maximizing parallelism entails partitioning the DNN into subsets of operation that can be computed independently and simultaneously and whose results are combined upon completion.

Both maximizing for concurrency and parallelism necessitates data-flow analysis in order to identify data dependencies amongst operations, which would lead to data hazards in synthesized designs. Such data-flow analysis is expensive and grows (in runtime) as higher performance is pursued. This can be understood in terms of loop-nest representations of DNN operations; for example consider a convolution as in Listing 1. A schedule for the arithmetic operations for this loop nest

```
def conv2d(
  input: array(b, c_in, h, w),
  output: array(b, c_out, h, w),
  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):
        for iv4 in range(0, w):
          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 $\lfloor k/2 \rfloor$, 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, w),
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):
9          for iv4 in range(0, w):
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).

can be computed by first unrolling all the loops up to some "trip-count" and then computing the topological sort of the operations (known as *list scheduling*). 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 $b \times c_{out} \times h \times w \times c_{in} \times k^2$ store-load pairs. Note also, while in the case of Listing 1 the verification is straightforward, in general it might involve solving a small constraint satisfaction program [32].

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 *behaviorally verified* (see Section III-D).

C. FPGA design

At the register-transfer 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). The entire process is carried out, for example, by Xilinx's Vivado tool.

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

BraggHLS is an extensible (implemented in python) HLS framework which employs MLIR for extracting loop-nest representations of DNNs implemented as PyTorch models. Critically, and distinctly, it handles the DNN transformations as well as scheduling, binding, and FSM extraction; there is no dependence on any commercial HLS tools. Figure 1 shows the architecture of BraggHLS . We discuss the most significant

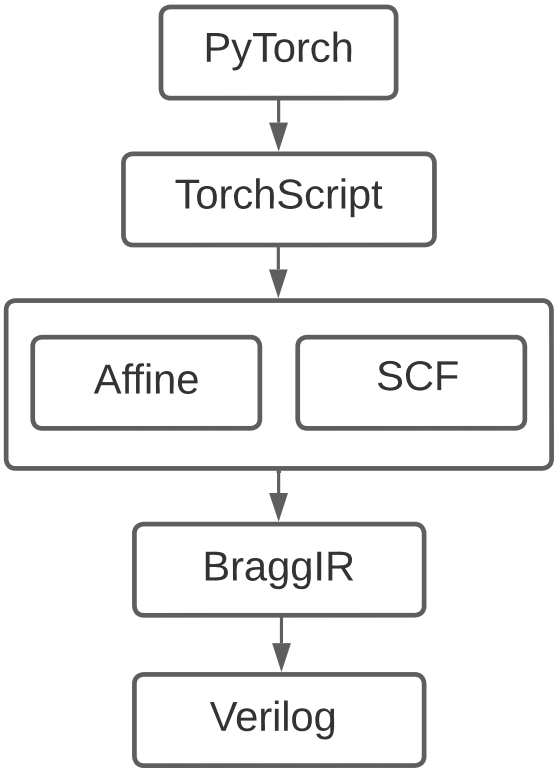


Fig. 1. BraggHLS framework overview (placeholder).

aspects of the architecture in the following.

A. Symbolic interpretation for fun and profit

First, DNNs are lowered from PyTorch to MLIR through TorchScript and the `torch` dialect. They are then further lowered from the `torch` dialect to the `scf` dialect (through the `linalg` dialect) in such a way that the inherent parallelism of each high-level operation is preserved (for example, see Listing 3). The value of making the parallelism explicit is that we can readily partition a DNN across a known set of hardware resources in order to maximize parallelism; since

```
@conv2d(
    %input: memref<b × c_in × h × w>,
    %weight: memref<b × c_out × h × w>,
    %output: memref<c_out × c_in × k × k>
) {
  scf.parallel (%iv1, %iv2, %iv3, %iv4) =
               (%c0, %c0, %c0, %c0) to
               (b, c_out, h, w) step
               (%c1, %c1, %c1, %c1) {
    scf.for %iv5 = %c0 to c_in step %c1 {
      scf.for %iv6 = %c0 to k step %c1 {
        scf.for %iv7 = %c0 to k step %c1 {
          %3 = arith.addi %iv3, %iv6
          %4 = arith.addi %iv4, %iv7
          %5 = memref.load %input[%iv1, %iv5, %iv3, %3, %4]
          %6 = memref.load %weight[%iv2, %iv5, %iv6, %iv7]
          %7 = memref.load %output[%iv1, %iv2, %iv3, %iv4]
          %8 = arith.mulf %5, %6
          %9 = arith.addf %7, %8
          memref.store %9, %output[%iv1, %iv2, %iv3, %iv4]
        }
      }
    }
  }
  return %2
}
```

Listing 3: Parallel loop representation of the convolution in Listing 1.

for each value of (`%iv1, %iv2, %iv3, %iv4`) the body of the `scf.parallel` is independent of all others, we can bind all the encompassed operations to unique hardware resources (DSPs, LUTs, and FFs). Thus, we can infer peak resource usage by computing the maximum cardinality of the cartesian product of the iteration spaces of the parallel iteration variables over all such parallel loops. For example, the convolution in Listing 3 would bind to

$$K = |\{\texttt{\%iv1} = \texttt{\%c0} + \texttt{\%c1} \times \mathbb{N} \wedge \texttt{\%iv1} < b\}| \times |\{\texttt{\%iv2} = \texttt{\%c0} + \texttt{\%c1} \times \mathbb{N} \wedge \texttt{\%iv2} < c_{out}\}| \times |\{\texttt{\%iv3} = \texttt{\%c0} + \texttt{\%c1} \times \mathbb{N} \wedge \texttt{\%iv3} < h\}| \times |\{\texttt{\%iv4} = \texttt{\%c0} + \texttt{\%c1} \times \mathbb{N} \wedge \texttt{\%iv4} < w\}|$$

collections of resources (i.e., $2K$ DSPs if `mul`, `add` bind to 1 DSP each), where `%c1` $\times \mathbb{N}$ represents all multiples of `%c1`. Note, DNN hyperparameters such as striding and padding make this a non-trivial calculation. Once peak usage is determined, successive instances `scf.parallel` loop bodies evaluated at the same concrete values of (`%iv1, %iv2, %iv3, %iv4`) are bound to the same resources. For DNN operations that lower to sequential loop nests (e.g., `sum`, `max`,or `prod`), we fully unroll the loops and use a reduction tree approach.

In addition to enabling us to perform binding, as discussed in Section II-B1, a loop-nest representation enables us to effectively perform data-flow analysis and schedule the encompassed arithmetic operations (see the next Section regarding scheduling), given that we can first unroll the loops. As also discussed in Section II-B1, the formally correct approach to unrolling a loop nest is prohibitively expensive in terms of runtime. Indeed, for BraggNN, with respect to producing a

RTL representation achieving latency within 1000× of the target latency (i.e., <1 ms), performing this unrolling in a reasonable amount of time (e.g., <12 hours) was extremely challenging. To overcome this runtime cost during the space exploration phase of the design process, the approach that BraggHLS takes is to implement a *symbolic interpreter* for these loop-nest representations. Specifically, BraggHLS translates scf dialect MLIR (such as in Listing 3) into legal python code (such as in Listing 1) and then, after applying some transformations on the resulting python abstract syntax tree (AST), executes that python code.

In executing the python code, the BraggHLS interpreter effectively unrolls loops by executing them under a re-defined set semantics, while enforcing SSA. That is to say, for a loop whose body has repeated assignments to the same SSA value (ostensibly violating SSA), we execute the loop and instantiate unique identifiers for the result of each operation. The interpreter then partially evaluates where possible, such as in functions of iteration variables, such as `%3 = arith.addi %iv3, %iv6`. This enables BraggHLS to concretely determine array index operands of all stores and loads, such as `memref.load %input[%iv1,%iv5,%iv3,%3,%4]`.

Note, we do not evaluate values corresponding to floating point arithmetic, as they represent true evaluation of the DNN; our interpreter represents the operands of such operations as symbols and merely records the arithmetic operations performed on them. This enables us to both unroll the loop and track data-flow through arithmetic operations (see Section 2). Finally, the interpreter reinterprets memrefs as *geometric symbol tables* (i.e., symbol tables indexed by array indices rather than identifiers/names) and stores and loads as assignments/reads to/from those symbol tables. Such semantics, in combination with fully evaluated array indices, enable BraggHLS to track the flow of data through arithmetic operations and memrefs, and thus perform store-load forwarding.

B. Scheduling

Recall that one of the critical functions which HLS fulfills is the scheduling of operations during each clock cycle, in such a way that they preserve the data-flow graph of a DNN; that schedule then informs the construction of a corresponding FSM. As already mentioned, scheduling arbitrary DNNs involves formulating and solving an ILP. In the resource-unconstrained case, due to the precedence relations induced by data-flow, the constraint matrix of the associated ILP is *totally unimodular matrix* and the feasible region of the problem is an integral polyhedron. Thus, in such cases, the scheduling problem can be solved optimally in polynomial time with a LP solver [33].

In the resource constrained case it is possible to transform resource constraints into precedence constraints as well, by picking a particular (possible heuristic) linear ordering on the resource-constrained operations. This transformation partitions resource constrained operations into distinct clock cycles, thereby guaranteeing sufficient resources are available for all operations scheduled within the same clock cycle [34]. BraggHLS uses the inherent parallelism of the loop-nest representation to decide a linear ordering on resource constrained operations: since successive instances of scf.parallel loop bodies evaluated at the same concrete values of the iteration variables are bound to the same resources, all resource constrained operations common to both instances are ordered. Note, since each loop nest corresponds to a DNN operation, ordering of scf.parallel loop nests is determined by the higher-level structure of the DNN. For DNN operations that lower to reduction trees, we use As-Late-As-Possible scheduling [35] amongst the subtrees.

C. Floating point arithmetic implementations

We use the FloPoCo [36] hardware generator to generate pipelined implementations of the standard floating point arithmetic operations (mul, div, add, sub, sqrt) at various precisions. In addition, we implement a few generic (parameterized by bit width) operators in order to support a range of DNN operations:

- Two-operand maximum (max);
- Negation (neg);
- Rectified linear units (relu), which could, alternatively, be represented as $\text{relu}\,(x) := \max(0, x)$.

Transcendental functions, such as exp, are implemented using a Taylor series expansion to kth order (where k is determined on a case by case basis). Note, FloPoCo's floating point format differs slightly from IEEE754, foregoing subnormals and differently encoding zeroes, infinities and NaNs, for the benefit of reduced complexity.

D. AST transformations and behavioral verification

In addition to unrolling loop-nests, BraggHLS performs some simple AST transformations on the python generated from scf dialect:

1) **Hoist globals**: all DNN tensors which are fixed (i.e., weights) are moved out of the body of the python [2] and into the parameter list, for the purpose of ultimately exposing them at the RTL module interface;
2) **Remove `if` expressions**: the scf dialect represents relu operations as ternary conditionals, which translate to python ternaries, thus introducing unnecessary control flow;
3) **Remove MACs**: sequences of load-multiply-add-store are very common in DNN implementations, thus we schedule such sequences jointly (this transformation coalesces such sequences into a single FMAC);
4) **Reduce `for`s**: this transformation implements the reduction tree structure for non-parallelizable loop-nests mentioned in Section III-A.

These transformations on the python AST are simple (implemented with procedural pattern matching), extensible, and

[2] BraggHLS translates the MLIR module corresponding to the DNN into a single python function in order to simplify analysis and interpretation.

efficient (marginal runtime cost) because they are unverified: no effort is made to verify their formal correctness. Thus, BraggHLS trades formal correctness for development time performance. This tradeoff enables quick design space iteration, which for example, enabled us to achieve very low latency implementations for BraggNN (see Section V).

As a substitute for formal verification, BraggHLS supports behavioral verification. Specifically, BraggHLS, can generate testbenches for all synthesized RTL. The test vectors for these testbenches are generated by evaluating the transformed representation (i.e., compiling and executing the python AST) on randomly generated inputs but with floating point operations now evaluated using functional models of the corresponding FloPoCo operators[3].The testbenches can then be run using any IEEE 1364 compliant simulator. For example, we run a battery of such testbenches (corresponding to various DNN operation types), using `cocotb` [37] and `iverilog` [38], as a part of our continuous integration process[4].

IV. Evaluation

asdasd

V. BraggNN case study

High-energy diffraction microscopy techniques can provide non-destructive characterization for a broad class of single-crystal and polycrystalline materials. The critical steps in a typical HEDM experiment involve an analysis to determine precise Bragg diffraction peak characteristics (and reconstruction of material grain information from the peak characteristics). Peak characteristics are typically computed by fitting the peaks to a probability distribution, e.g., Gaussian, Lorentzian, Voigt, or Pseudo-Voigt. As already mentioned (in Section I) the experiments can have collection rates of 80 GB/s. The data rates, though more modest than those observed at the LHC, combined with the runtime of the fitting procedure, merit exploring the same low latency approach in order to enable experimental modalities that depend on measurement-based feedback (i.e., experiment steering).

BraggNN [39], a DNN aimed at efficiently characterizing Bragg diffraction peaks, achieves a batched latency of 22 µs/sample. This is a large speedup over the classical pseudo-Voigt peak fitting methods, but still falls far short of the 1 µs/sample target for handling the 1 MHz sampling rates. In addition, the current implementation of BraggNN, deployed to either a data-center class GPU such as a NVIDIA V100, or even a workstation class GPU such as a NVIDIA RTX 2080Ti, has no practicable means to being deployed at the edge, i.e., adjacent or proximal to the high energy microscopy equipment. We applied BraggHLS to the PyTorch representation of BraggNN (see Listing 4) and achieve a RTL implementation which synthesizes to a 1,238 interval design that places, routes, and meets timing closure for a clock period of 10 ns (for a Xilinx Alveo U280). The design consists of a three stage pipeline with long stage measuring 480 intervals. Thus, the throughput of the implementation is 4.8 µs/sample. See Table I for a summary of the resource usage of the implementation.

```
BraggNN(
  (cnn_layers_1): Conv2d(k × 16, kernel=3, stride=1)
  (nlb): NLB(
    (theta_layer): Conv2d(k × 16, k × 8, kernel=1, stride=1)
    (phi_layer): Conv2d(k × 16, k × 8, kernel=1, stride=1)
    (g_layer): Conv2d(k × 16, k × 8, kernel=1, stride=1)
    (out_cnn): Conv2d(k × 8, k × 16, kernel=1, stride=1)
    (soft): Softmax()
  )
  (cnn_layers_2): Sequential(
    (0): ReLU()
    (1): Conv2d(k × 16, k × 8, kernel=3, stride=1)
    (2): ReLU()
    (3): Conv2d(k × 8, k × 2, kernel=3, stride=1)
    (4): ReLU()
  )
  (dense_layers): Sequential(
    (0): Linear(in_features=k × 50, out_features=k × 16)
    (1): ReLU()
    (2): Linear(in_features=k × 16, out_features=k × 8)
    (3): ReLU()
    (4): Linear(in_features=k × 8, out_features=k × 4)
    (5): ReLU()
    (6): Linear(in_features=k × 4, out_features=2)
    (7): ReLU()
  )
)
```

Listing 4: BraggNN for $k = 1, \ldots, 4$.

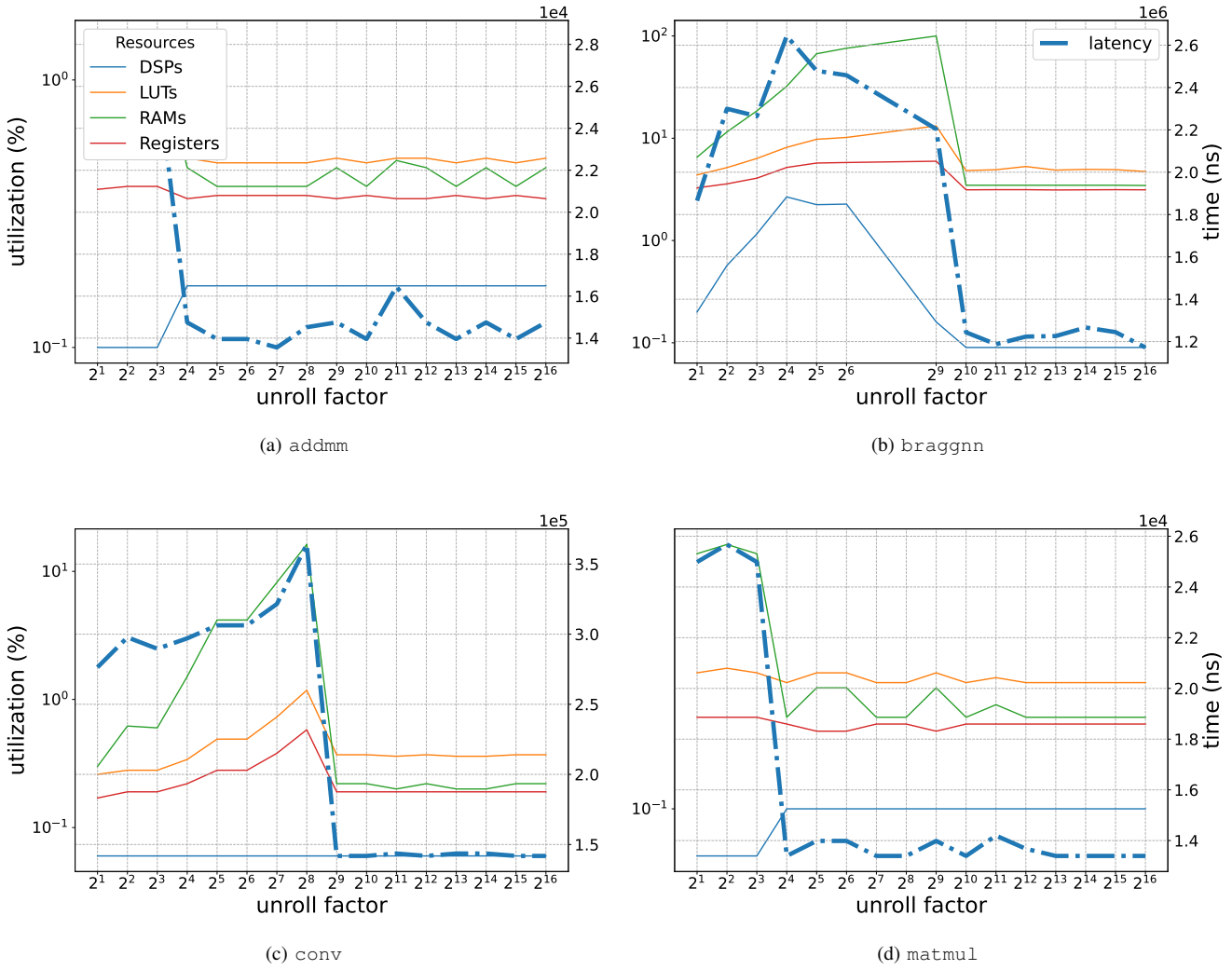
The most challenging aspect of implementing BraggNN was minimizing latency while satisfying compute resource constraints (LUTs, DSPs, BRAMs) and meeting routing "closure", i.e., not exceeding available routing resources and avoiding congestion. Two design choices made for the purposes of reducing resource consumption was reducing the precision used for the floating point operations. We reduced the precision from IEEE half precision (5 bits for the exponent and 11 bits for the mantissa) to FloPoCo $(5, 4)$-precision (5 bits for the exponent and 4 bits for the mantissa). This was justified by an examination of the distribution of the weights of the fully trained BraggNN (see figure 4). Reducing the precision enabled us to eliminate BRAMs from the design as well, since, at the lower precision, all weights could be represented as registered constants. The reduced precision also drives the Vivado synthesizer to infer implementations of the floating point operations that make no use of DSPs; this was not intentional but seemingly cannot be altereted. Most likely this is due to the fact that DSP48 hardware block includes a 18-bit by 25-bit signed multiplier and a 48-bit adder [40], neither of which neatly divide the bit width of FloPoCo $(5, 4)$-precision cores[5].

Achieving routing closure was very difficult due to the nature of the Xilinx's UltraScale architecture, of which the Alveo U280 is an instance. The UltraScale architecture achieves its scale through "Stacked Silicon Interconnect" (SSI) technology

[3]FloPoCo provides these functional models as C++ implementations - we simply add python bindings to these C++ implementations.

[4]https://github.com/makslevental/bragghls/actions

[5]The actual bit width for FloPoCo $(5, 4)$-precision is 12 bits: 1 extra bit is needed for the sign and 2 bits are needed for FloPoCo's handling of exceptional conditions.



(a) `addmm` (b) `braggnn`

(c) `conv` (d) `matmul`

Fig. 2. Resource usage and latency vs. unroll factor of various DNN modules.

[41], which amounts to multiple distinct FPGA dies, called Super Logic Regions (SLRs), on the same chip, connected by interposers. Adjacent SLRs communicate with each other using a limited set of Super Long Lines (SLLs). These SLLs determine the maximum bus width that span two SLRs. On the Alveo U280 there are exactly 23,040 SLLs available between adjacent SLRs and at $(5, 4)$-precision `BraggNN` needs 23,328 SLLs between SLR2 and SLR1[6]. Thus, we further reduce the precision to $(5, 3)$. Finally, since multiple dies constitute independent clock domains, the SLLs that cross SLRS are sensitive hold time violations due to the higher multi-die variability [42]. This multi-die variability leads to high congestion if not addressed. Thus, routing across them needs to handled manually using placement and routing constraints for logic in each SLR and the addition of so-called "launch" and "latch" registers in each SLR. See figure 5 for an illustration on the effect of using launch and latch registers as well as placement and routing constraints.

[6]We route the output of `cnn_layers_1` ($1 \times 16 \times 9 \times 9 \times 12$ wires) as well as the output of `soft(theta_layer` $\times$ `phi_layer)` $\times$ `g_layer` ($1 \times 8 \times 9 \times 9 \times 12$ wires) from SLR2 to SLR1.

VI. Conclusion

287 + 471 + 480

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, 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.

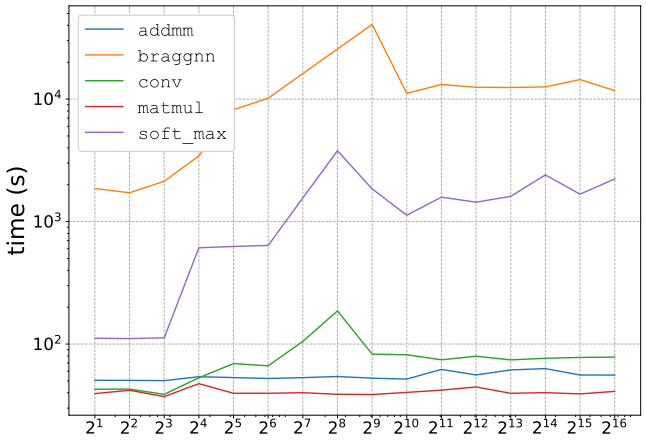


Fig. 3. Runtime of Vitis HLS vs. unroll factor.

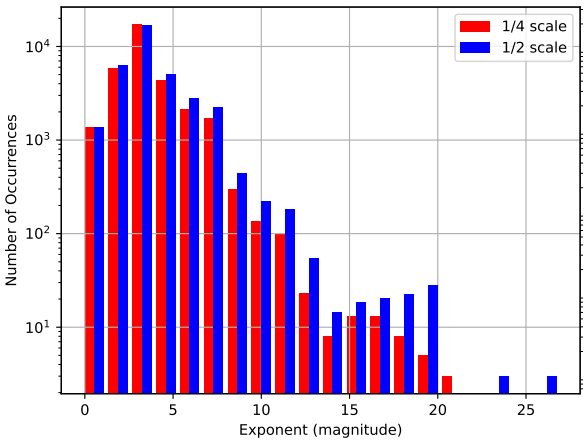


Fig. 4. `BraggHLS` weights exponent distribution (placeholder).

[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, 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.

TABLE I
RESOURCE USAGE FOR BRAGGNN WITH $k = 1$ AND $(5, 3)$-PRECISION FLOPOCO

Site Type	SLR0	SLR1	SLR2	SLR0 %	SLR1 %	SLR2 %
CLB	5047	52648	53900	9.18	97.50	99.81
CLBL	2773	28613	29227	9.47	97.72	99.82
CLBM	2274	24035	24673	8.86	97.23	99.81
CLB LUTs	19797	263733	311794	4.50	61.05	72.17
LUT as Logic	19797	263733	311794	4.50	61.05	72.17
using O5 output only	277	3944	4304	0.06	0.91	1.00
using O6 output only	17176	202564	266733	3.91	46.89	61.74
using O5 and O6	2344	57225	40757	0.53	13.25	9.43
LUT as Memory	0	0	0	0.00	0.00	0.00
LUT as Distributed RAM	0	0	0	0.00	0.00	0.00
LUT as Shift Register	0	0	0	0.00	0.00	0.00
CLB Registers	12527	286226	339820	1.42	33.13	39.33
CARRY8	244	5184	5184	0.44	9.60	9.60
Block RAM Tile	0	0	0	0.00	0.00	0.00
RAMB36/FIFO	0	0	0	0.00	0.00	0.00
RAMB18	0	0	0	0.00	0.00	0.00
URAM	0	0	0	0.00	0.00	0.00
DSPs	0	0	0	0.00	0.00	0.00
Unique Control Sets	189	2641	3179	0.17	2.45	2.94

TABLE II
SUPER LONG LINE USAGE ACROSS SUPER LOGIC REGIONS FOR BRAGGNN

	Used	Fixed	Available	Util %
SLR2 $\leftrightarrow$ SLR1	21366		23040	92.73
SLR1 $\rightarrow$ SLR2	2			< 0.01
SLR2 $\rightarrow$ SLR1	21364			92.73
SLR1 $\leftrightarrow$ SLR0	3904		23040	16.94
SLR0 $\rightarrow$ SLR1	2			< 0.01
SLR1 $\rightarrow$ SLR0	3902			16.94
Total SLLs Used	25270			

[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

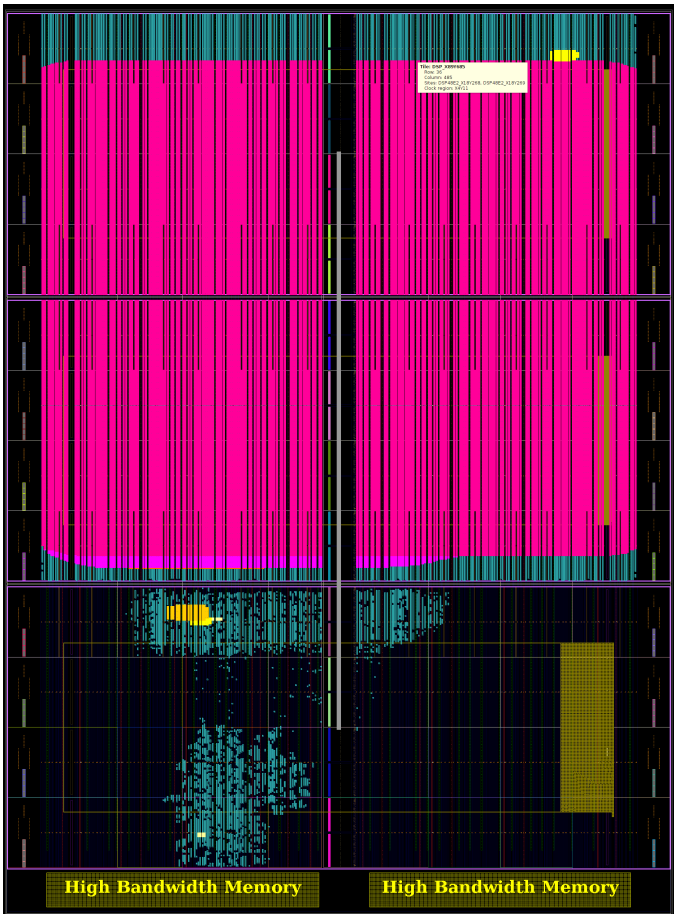
[26] A. Ashari, S. Tatikonda, M. Boehm, B. Reinwald, K. Campbell, J. Keenleyside, and P. Sadayappan, "On optimizing machine learning workloads via kernel fusion," vol. 50, no. 8, pp. 173–182, jan 2015. [Online]. Available: https://doi.org/10.1145/2858788.2688521

[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.

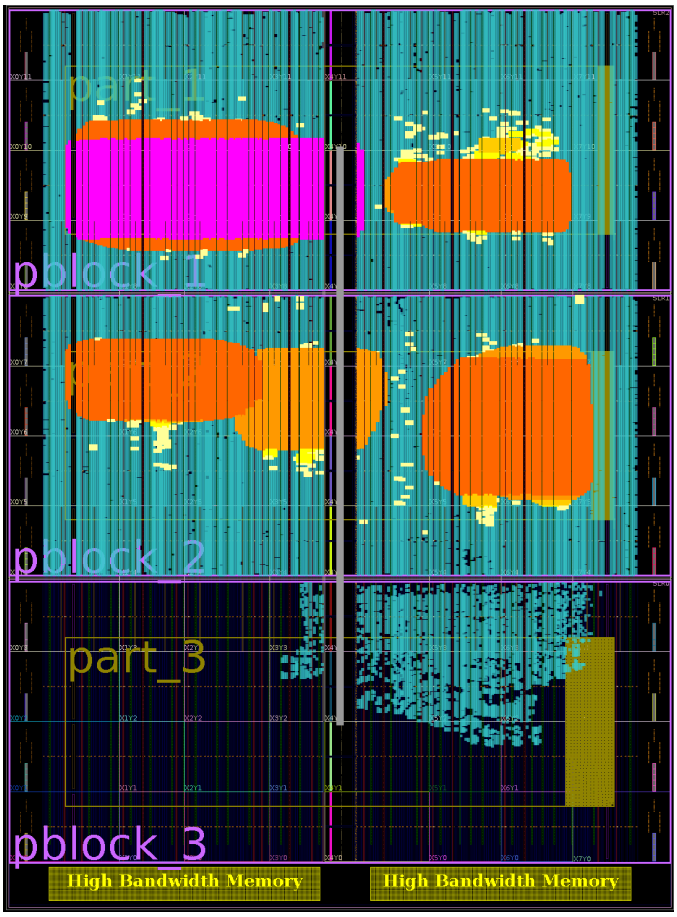
[28] S. Silva and A. Elangovan, "Torch-MLIR," https://mlir.llvm.org/OpenMeetings/2021-10-07-The-Torch-MLIR-project.pdf, 2021.

[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_



(a) `BraggNN` fails to achieve routing closure without placement and routing constraints and launch and latch registers.



(b) `BraggNN` achieves routing closure with use of per SLR placement and routing constraints (`pblock_1`, `pblock_2`, `pblock_3`) and launch and latch registers (not highlighted).

Fig. 5. Congestion maps for `BraggNN` on a Xilinx Alveo U280. Magenta indicates area of high congestion.

2020_keynote.pdf, 2020.

[31] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

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

[33] J. Oppermann, "Advances in ilp-based modulo scheduling for high-level synthesis," Ph.D. dissertation, Technische Universität, Darmstadt, 2019. [Online]. Available: http://tuprints.ulb.tu-darmstadt.de/9272/

[34] S. Dai, G. Liu, and Z. Zhang, "A scalable approach to exact resource-constrained scheduling based on a joint sdc and sat formulation," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 137–146. [Online]. Available: https://doi.org/10.1145/3174243.3174268

[35] Z. Baruch, "Scheduling algorithms for high-level synthesis," *ACAM Scientific Journal*, vol. 5, no. 1-2, pp. 48–57, 1996.

[36] F. de Dinechin, "Reflections on 10 years of flopoco," in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, 2019, pp. 187–189.

[37] B. J. Rosser, "Cocotb: a python-based digital logic verification framework," 2018.

[38] S. Williams, "Icarus verilog, 1998–2020," *URL http://iverilog. icarus. com*.

[39] Z. Liu, H. Sharma, J.-S. Park, P. Kenesei, A. Miceli, J. Almer, R. Kettimuthu, and I. Foster, "*BraggNN*: fast X-ray Bragg peak analysis using deep learning," *IUCrJ*, vol. 9, no. 1, pp. 104–113, Jan 2022.

[40] A. S. U. Guide, "Ultrascale architecture dsp slice."

[41] S. Leibson and N. Mehta, "Xilinx ultrascale: The next-generation architecture for your next-generation architecture," *Xilinx White Paper WP435*, vol. 143, 2013.

[42] "Create placed and routed dcp to cross slr," https://www.rapidwright.io/docs/SLR_Crosser_DCP_Creator_Tutorial.html, accessed: 2022-10-15.