

BraggHLS

Maksim Levental
and Arham Khan
University of Chicago
Email: test@test.tes

Ryan Chard
and Kazutomo Yoshi
Argonne National Laboratory
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 processing. 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 implementation of 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 of the network with a throughput 4.8 μ s/sample, which is approximately a 4 \times 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	5
III-A	Symbolic interpretation for fun and profit .	6
III-B	AST transformations and behavioral verification	6
III-C	Scheduling	7
IV	Evaluation	8
IV-A	DNN layers	8
IV-B	BraggNN case study	9
V	Conclusion	12
	Appendix	14

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) [18]. 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 per 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 [21], 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 [31], will sustain data rates in excess of 10 TB/s [19].

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, the tradeoff made in employing filtering should be clear: reduced storage at the expense of more stringent latency constraints (on the filtering mechanisms). Typically, these filtering mechanisms consist either of physics based models [11] or machine learning models [17]; in either case maximally efficient and effective use of the target hardware platform is important. 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) [15]. Here we focus primarily on FPGAs.

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 [3]. 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 *weights* that parameterize the

DNN. In recent years they have been investigated for near real-time scientific use cases [28, 36, 27] but their use for the lowest latency use cases has been limited [15]. 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 [1]);
- 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 [35], TensorFlow [2], MXNet [8]), which abstract all implementation details from the user, thereby making portability of existing model architectures (to e.g., FPGA) high 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 low-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) [32] tools. While several such HLS tools exist [7, 43, 16] they struggle to effectively perform the necessary optimizations in reasonable amounts of time (see Section II-B1) despite, often, bundling robust optimizing compilers,.

Recently, deep learning compilers (such as TVM [10], MLIR [24], and Glow [39]) have demonstrated the ability to dramatically reduce inference latencies [26], training times [44], and memory usage [9] 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 [4], vectorization [30], and memory planning [9]). 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 be 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 compatible implementations. BraggHLS uses a combination of compiler and HLS techniques to compile the entire DNN into fully scheduled RTL, thereby eliminating all synchronization overheads and achieving low latency. BraggHLS is general and supports a wide range of DNN layer types, and thus a wide range of

DNNs, but we particularly focus on optimizations relevant to a DNN designed for identifying Bragg diffraction peaks. In summary our specific contributions include:

- 1) We describe and implement a compiler framework, BraggHLS, which can effectively transform unoptimized, hardware-agnostic PyTorch models into low latency RTL designs suitable for deployment to Xilinx FPGAs. BraggHLS is thoroughly tested, open source, and available at <https://github.com/makslevental/braggHLS/>;
- 2) 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 for BraggNN, a DNN designed for identifying Bragg diffraction peaks;
- 3) We discuss some of the challenges faced even after successful synthesis of RTL from a high-level representation of a DNN, namely during the place and route phases of implementation.

The rest of this paper is organized as follows: Section II reviews key concepts from compilers, high-level synthesis, and RTL design for FPGA. 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 IV-B describes our case study, i.e., BraggHLS applied to BraggNN, a Bragg peak detection DNN with a target latency of 1 μ s/sample. 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, representation of a DNN to a register-transfer level representation can be 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 employ in manipulating those representations:

- 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 popular because of their ease of use and large library of example implementations of various DNN model architectures. BraggHLS targets the PyTorch framework, 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 (partial) 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.

On the other hand, 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 easily be inferred from the textual representation (i.e., the python source) of the DNN. Furthermore, a priori, the runtime-materialized DFG is only partially materialized¹, and only as an in-memory data structure. Thus, framework support is necessary for efficiently extracting the full DFG. Indeed, PyTorch supports a Single Static Assignment (SSA) IR, called TorchScript (TS) IR and accompanying tracing mechanism (the TS JIT), which generates 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 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 [24] 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 directly 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 [40], 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 [22].

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 de novo implementations of DNN operations (e.g., for FPGA). Fortunately, MLIR supports lower-level dialects, such as the `linalg`, `affine` and `scf` dialects. The `scf` (structured control flow) dialect describes standard control flow primitives, such as conditionals and loops, and is mutually compatible with the `arith` (arithmetic operations) and `memref` (memory buffers) dialects. The `affine` dialect, on the other hand, provides a formalization of semantics that lend themselves to polyhedral compilation techniques [6], i.e., techniques that enable loop dependence analysis and loop transformations. Such loop transformations, particularly loop unrolling, are crucial for achieving lowest possible latencies [42] because loop nests

directly inform the concurrency and parallelism of the final RTL design.

B. High-level synthesis and FPGA design

1) *High-level synthesis*: High-level synthesis tools produce RTL descriptions of designs from high-level representations, such as C or C++ [7, 16]. In particular, Xilinx’s Vitis HLS, based on the Autopilot project [43], is a state-of-the-art HLS tool. Given a high-level, procedural, representation, HLS carries out three fundamental tasks, 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 delay for each operation;
 - c) The user’s desired clock rate/frequency;
- 2) HLS associates (called *binding*) floating point operations to RTL instantiations of intellectual property (IP) for those operations; for example whether to associate an addition operation followed by a multiply operation to IPs for each, or whether to associate them both with a single IP, designed 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 during the appropriate stages of the schedule.

In addition to fulfilling these three fundamental tasks, high-level synthesis aims to optimize the program. In particular, HLS attempts 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 when possible, 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 aggregated upon completion.

While HLS aims to automatically optimize various characteristics of a design, there are challenges associated with this kind of automated optimization. In particular, maximum concurrency and parallelism necessitates data-flow analysis in order to identify data dependencies amongst operations, both for scheduling and identifying potential data hazards. Such data-flow analysis is expensive and grows (in runtime) as better performance is pursued. This can be understood in

¹“...instead, every intermediate result records only the subset of the computation graph that was relevant to their computation.” [35]

```

def conv2d(
    input: MemRef(b, cin, h, w),
    output: MemRef(b, cout, h, w),
    weight: MemRef(cout, cin, k, k)
):
    for i1 in range(0, b):
        for i2 in range(0, cout):
            for i3 in range(0, h):
                for i4 in range(0, w):
                    for i5 in range(0, cin):
                        for i6 in range(0, k):
                            for i7 in range(0, k):
                                _3 = i3 + i6
                                _4 = i4 + i7
                                _5 = input[i1, i5, _3, _4]
                                _6 = weight[i2, i5, i6, i7]
                                _7 = output[i1, i2, i3, i4]
                                _8 = _5 * _6
                                _9 = _7 + _8
                                output[i1, i2, i3, i4] = _9

```

Listing 1: Python representation of a 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.

terms of loop-nest representations of DNN operations; for example consider the convolution in Listing 1. A schedule that parallelizes (some of) the arithmetic operations for this loop nest can be computed by first unrolling the loops up to some “trip-count” and then computing the topological sort of the operations. Using this scheduling algorithm (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 conventional resolution to this loss of information is to perform *store-load forwarding*: pairs of store and load operations on 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 (i.e., preserve program semantics), for each pair of candidate store and load operations, it must be verified that there are no intervening memory operations on the same memory address. These verifications are non-trivial since the iteration spaces of the loops need not be regular; in general it might involve solving a small constraint satisfaction program [37]. Furthermore, the number of such verifications grows 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 on the output array.

Finally, note, though greedy solutions to the scheduling problem solved by HLS are possible, in principle, the scheduling problem is formulated as an integer linear program (ILP), instances of which are NP-hard. In summary, HLS tools solve computationally intensive problems in order to produce a

```

1 def conv2d(
2     input: MemRef(b, cin, h, w),
3     output: MemRef(b, cout, h, w),
4     weight: MemRef(cout, cin, k, k)
5 ):
6     for i1 in range(0, b):
7         for i2 in range(0, cout):
8             for i3 in range(0, h):
9                 for i4 in range(0, w):
10                    ...
11                    # e.g., i5, i6, i7 = 2, 3, 4
12                    _31 = i3 + i6
13                    _41 = i4 + i7
14                    _51 = input[i1, i5, _31, _41]
15                    _61 = weight[i2, i5, i6, i7]
16                    _71 = output[i1, i2, i3, i4]
17                    _81 = _51 * _61
18                    _91 = _71 + _81
19                    output[i1, i2, i3, i4] = _91
20                    # i5, i6, i7 = 2, 3, 5
21                    _32 = i3 + i6
22                    _42 = i4 + i7
23                    _52 = input[i1, i5, _32, _42]
24                    _62 = weight[i2, i5, i6, i7]
25                    _72 = output[i1, i2, i3, i4]
26                    _82 = _52 * _62
27                    _92 = _72 + _82
28                    output[i1, i2, i3, i4] = _92
29                    ...

```

Listing 2: Store-load forwarding across successive iterations (e.g., $i7 = 4, 5$) 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).

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 (but can be *behaviorally verified*) and have much lower runtimes (see Section III-A).

C. FPGA design

Broadly, at the register-transfer level of abstraction, there remain two more steps prior to being able to actually deploy a design 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,

for example, 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. Logic synthesis also determines the number of LUTs and DSPs that a high-level representation of a DNN corresponds to, which 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 configures the FPGA.

III. BRAGGHLS COMPILER AND HLS FRAMEWORK

BraggHLS is a compiler and HLS framework which employs MLIR for extracting loop-nest representations of DNNs. It is implemented in python for ease of use and extensibility. 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. BraggHLS proceeds by first low-

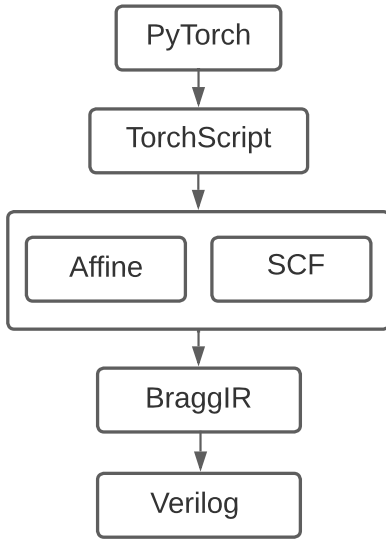


Fig. 1. BraggHLS framework overview (placeholder).

ering DNNs from PyTorch to MLIR through TorchScript and the `torch` dialect (see Section II-A2). They are then further lowered from the `torch` dialect to the `scf` dialect (through the `linalg` dialect). Such a representation lends itself to a

```

@conv2d(
    %input: memref<b × cin × h × w>,
    %weight: memref<b × cout × h × w>,
    %output: memref<cout × cin × k × k>
) {
    scf.for %i1 = %c0 to b step %c1 {
        scf.for %i2 = %c0 to cout step %c1 {
            scf.for %i3 = %c0 to h step %c1 {
                scf.for %i4 = %c0 to w step %c1 {
                    scf.for %i5 = %c0 to cin step %c1 {
                        scf.for %i6 = %c0 to k step %c1 {
                            scf.for %i7 = %c0 to k step %c1 {
                                %3 = arith.addi %i3, %i6
                                %4 = arith.addi %i4, %i7
                                %5 = memref.load %input[
                                    %i1, %i5, %i3, %3, %4]
                                %6 = memref.load %weight[
                                    %i2, %i5, %i6, %i7]
                                %7 = memref.load %output[
                                    %i1, %i2, %i3, %i4]
                                %8 = arith.mulf %5, %6
                                %9 = arith.addf %7, %8
                                memref.store %9, %output[
                                    %i1, %i2, %i3, %i4]
                            }
                        }
                    }
                }
            }
        }
    }
    return %2
}

```

Listing 3: `scf` dialect loop representation of the convolution in Listing 1.

straightforward translation to python (compare Listing 1 to Listing 1) and indeed BraggHLS performs this translation. The benefits of translating `scf` dialect to python are manifold and discussed in the following (see Section III-A). Ultimately, BraggHLS produces a representation of the DNN that is then fully scheduled using the scheduling infrastructure in CIRCT [34] (an MLIR adjacent project). After scheduling, BraggHLS emits corresponding RTL (as Verilog).

BraggHLS delegates to the FloPoCo [14] IP generator the task of generating pipelined implementations of the standard floating-point arithmetic operations (`mulf`, `divf`, `addf`, `subf`, `sqrtof`) at various precisions. In addition, we implement a few generic (parameterized by bit width) operators in order to support a broad range of DNN operations: two-operand maximum (`max`), unary negation (`neg`), and the rectified linear unit (`relu`). Transcendental functions, such as `exp`, are implemented using a Taylor series expansion to k -th order (where k is determined on a case-by-case basis). Note, FloPoCo’s floating-point representation differs slightly from IEEE754, foregoing subnormals and differently encoding zeroes, infinities and NaNs (for the benefit of reduced complexity) and our implementations `max`, `neg`, `relu` are adjusted appropriately.

We now discuss some aspects of BraggHLS in greater detail.

A. Symbolic interpretation for fun and profit

As discussed in Section II-B1, maximizing concurrency and parallelism for a design entails unrolling loop nests and analyzing the data-flow of encompassed operations. As also discussed in Section II-B1, the formally correct approach to unrolling a loop nest is prohibitively expensive in terms of runtime; see Figure 2 for an illustration in terms of convolution. Indeed, for example, in the case of BraggNN, repeatedly performing this unrolling was a hindrance to effectively searching the design space for a RTL representation achieving the target latency, since it often took an enormous amount of time. Translating scf dialect to python enables BraggHLS

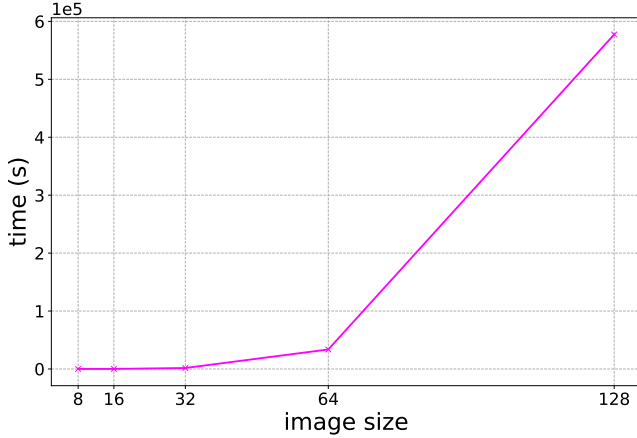


Fig. 2. 3×3 -kernel convolution (cf. Listing 3) full unrolling time vs. input (square) image size, with store-load forwarding using MLIR’s -affine-scalrep pass. Note that the longest time is $577419 \div 3600 \approx 160$ hours (for a loop nest with a trip count of $128 \times 128 \times 3 \times 3 = 147,456$).

to overcome this barrier by enabling us to use the python interpreter as a *symbolic interpreter*. Interpreting the resulting python loop-nests (i.e., running the python program) while treating the arithmetic and memory operations on SSA values as operations on symbols (i.e., python classes with overloaded methods) enables us to:

- 1) Partially evaluate functions of iteration variables, such as `%3 = arith.addi %i3, %i6`, which enables concretely determining array index operands of all stores and loads, such as `memref.load %input[%i1,%i5,%i3,%3,%4]`, and thereupon performing memory dependence checks, thus transforming the problem of statically verifying memory dependence into a matter of checking assertions at runtime;
- 2) Unroll loops by recording each floating-point arithmetic operation executed while enforcing SSA; e.g., for a loop whose body has repeated assignments to the same SSA value (ostensibly violating SSA), we execute the loop and instantiate new, uniquely identified, symbols for the result of each operation;

- 3) Reconstruct all data flow through arithmetic operations and memory operations by interpreting memrefs as *geometric symbol tables* (i.e., symbol tables indexed by array indices rather than identifiers/names) and stores and loads as reads and writes on those symbol tables;
- 4) Easily swap evaluation rules in order to support various functional modes, e.g., evaluating the floating-point arithmetic operations using (python) bindings to FloPoCo’s C++ functional models thereby enabling behavioral verification of our designs.

See Table I for the translation rules from MLIR dialects to python.

B. AST transformations and behavioral verification

Prior to interpretation, 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 generated python function² and into the parameter list, for the purpose of ultimately exposing them at the RTL module interface;
- 2) **Remove if expressions:** DNN relu operations are lowered to the scf dialect as a decomposition into `arith.cmpfugt` and `arith.select`; this transformation recomposes them into a relu;
- 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 operation);
- 4) **Reduce fors:** this transformation implements the reduction tree structure for non-parallelizable loop-nests mentioned in Section III-C.

These transformations on the python AST are simple (implemented with procedural pattern matching), extensible, and 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 IV-B). 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 generated python representation of the DNN on randomly generated inputs but with floating-point operations now evaluated using functional models of the corresponding FloPoCo operators. 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 [38] and iverilog [41], as a part of our continuous integration process.

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

TABLE I
SCF, ARITH, AND MEMREF DIALECT TO PYTHON TRANSLATION.

[[MLIR]]	python
[[%5]]	v5 = Val("%5")
[[memref<b × c _{in} × h × w>]]	MemRef(b, c _{in} , h, w)
[[%5 = memref.load %input[%i1, %i5, %3, %4]]]	[[%5]] = [[input]].__getitem__(([[%i1]], [[%i5]], [[%3]], [[%4]]))
[[memref.store %9, %output[%i1, %i5, %3, %4]]]	[[output]].__getitem__(([[%i1]], [[%i5]], [[%3]], [[%4]]), [[%9]])
[[scf.for %i1 = %c0 to b step %c1]]	for [[%i1]] in range([[%c0]], b, [[%c1]])
[[%3 = arith.addi %i3, %i6]]	[[%3]] = [[%i3]] + [[%i6]]
[[%8 = arith.mulf %5, %6]]	[[%8]] = [[%5]].__mul__(([[%6]]))
[[%9 = arith.addf %7, %8]]	[[%9]] = [[%7]].__add__(([[%8]]))
[[%63 = arith.cmpfugt %10, %c0]]	[[%63 = arith.cmpfugt %10, %c0]]
[[%64].relu(%10)]	[[%64].relu(%10)]
[[%8 = arith.mulf %5, %6]]	[[%8 = arith.mulf %5, %6]]
[[%9 = arith.addf %7, %8]]	[[%9 = arith.addf %7, %8]]
[[%9] = fma(%5, %6, %7)]	[[%9] = fma(%5, %6, %7)]

C. 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 a *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 (possibly 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 [13].

BraggHLS uses the explicit parallelism of the `scf.parallel` loop-nest representation to inform such a linear ordering on resource-constrained operations. By assumption, for loop-nests which can be re-represented as `scf.parallel` loop-nests (see Listing 4), each instance of a floating-point arithmetic operation in the body corresponding to unique values of the iteration variables (e.g., %i1, %i2, %i3, %i4 for Listing 4) is independent of all other such instances³. This exactly determines total resource usage per loop-nest; for example, the convolution in Listing 4, would

bind to $2K_i$ DSPs (assuming `mulf`, `addf` bind to one DSP each), where

$$K_i := |\{\%i1 = \%c0 + \%c1 \times \mathbb{N} \wedge \%i1 < b\}| \times |\{\%i2 = \%c0 + \%c1 \times \mathbb{N} \wedge \%i2 < c_{out}\}| \times |\{\%i3 = \%c0 + \%c1 \times \mathbb{N} \wedge \%i3 < h\}| \times |\{\%i4 = \%c0 + \%c1 \times \mathbb{N} \wedge \%i4 < w\}|$$

where $\%c1 \times \mathbb{N}$ represents all multiples of $\%c1$. That is to say, K_i is the cardinality of the cartesian product of the iteration spaces of the parallel iteration variables. Taking the maximum over such $K := \max_i K_i$ across all `scf.parallel` loop-nests, we can infer peak usage of any resource. Then, after indexing available hardware resources $j = 1, \dots, K$, we can bind the operations of any particular loop-nest. This leads to a linear ordering on resource-constrained operations such that operations bound to the same hardware resource index j must be ordered according to their execution order during symbolic interpretation⁴. Note, this ordering coincides with the higher-level structure of the DNN, since ordering of `scf.parallel` loop nests (and thus interpretation order during execution of the python program) is determined by the higher-level structure of the DNN.

For DNN operations that do not lower to `scf.parallel` loop-nests but do lower to sequential loop nests (e.g., `sum`, `max`, or `prod`), we fully unroll the loops and transform the resulting, sequential, operations into a reduction tree; we use

³Data-flow within a loop body must still be respected.

⁴BraggHLS only needs to construct a partial precedence ordering $op_a < op_b$ for operations op_a, op_b which CIRCT then combines with the delays of the operations to construct constraints such as $start_{op_a} + delay_a \leq start_{op_b}$.

```

@conv2d(
  %input: memref<b × cin × h × w>,
  %weight: memref<b × cout × h × w>,
  %output: memref<cout × cin × k × k>
) {
  scf.parallel (%i1, %i2, %i3, %i4) =
    (%c0, %c0, %c0, %c0) to
      (b, cout, h, w) step
      (%c1, %c1, %c1, %c1) {
    scf.for %i5 = %c0 to cin step %c1 {
      scf.for %i6 = %c0 to k step %c1 {
        scf.for %i7 = %c0 to k step %c1 {
          %3 = arith.addi %i3, %i6
          %4 = arith.addi %i4, %i7
          %5 = memref.load %input[%i1, %i5, %i3, %4]
          %6 = memref.load %weight[%i2, %i5, %i6, %i7]
          %7 = memref.load %output[%i1, %i2, %i3, %i4]
          %8 = arith.mulf %5, %6
          %9 = arith.addf %7, %8
          memref.store %9, %output[%i1, %i2, %i3, %i4]
        }
      }
    }
  }
  return %2
}

```

Listing 4: Parallel loop representation of the convolution in Listing 1.

As-Late-As-Possible scheduling [5] amongst the subtrees of such reduction trees.

IV. EVALUATION

We evaluate BraggHLS both on individual DNN layers, and end-to-end, on our use-case BraggNN. We compare BraggHLS to Xilinx’s Vitis HLS by comparing the latencies and resource usages of the final designs generated by each. We also compare the runtimes of the tools themselves. Both BraggHLS and Vitis HLS produce Verilog RTL, on which we run a synthesis pass using Xilinx’s Vivado. The particular FPGA target is Xilinx Alveo U280. We measure LUT, DSP, BRAM, and FF usage. For the DNN layer evaluations, we use FloPoCo (5,11)-floating point representations (5 bits for the exponent and 11 bits for the mantissa) corresponding to Vitis HLS’s IEEE half precision IPs. We synthesize all designs for a 10 ns target clock period and report end-to-end latency as a product of the total schedule interval count of the design and achieved clock period ($10 - WNS$, where WNS is the worst negative slack reported). Note, in the case of Vitis HLS, which potentially explicitly pipelines the design and therefore implements with an initiation interval strictly less than the total schedule interval count, we report in terms of the best possible interval count (LatencyBest from the Vitis HLS reports). All other measurements are collected from Vivado synthesis reports. Note, since Vitis HLS operates on C++ representations, we generate such a representation for our test cases by first lowering each DNN layer to the affine dialect and then using the `scalehls-translate` tool of the ScaleHLS project [42] to emit C++. Importantly, we do not make any use of `scalehls-opt` optimization tool (of the same project).

Since our ultimate goal is low latency inference, and since the strategy that BraggHLS employs in the pursuit of this goal is loop unrolling, in order to produce a like for like comparison, we similarly unroll the representation that is passed to Vitis HLS. Thus, all Vitis HLS measurements are reported in terms of *unroll factor*: an unroll factor of k corresponds to a k -fold increase in the number of statements in the body of a loop and commensurate k -fold decrease in the trip count of the loop. For loop nests we unroll inside out: if k is greater than the trip count t of the innermost loop, we unroll the innermost loop completely and then unroll the enclosing loop by a factor of $k - t$. We do not perform any store-load forwarding during this preprocessing but we annotate all arrays with the directive `array_partition complete dim=1` in order that Vitis HLS can effectively pipeline. All representations generated by BraggHLS correspond to full unrolling of the loop nests.

A. DNN layers

See Figure 3; we evaluate BraggHLS against Xilinx’s Vitis HLS by comparing the latency of the final design on the following, individual, DNN layer types:

- `addmm(a, b, c)`: Matrix multiplication of the matrices a and b . The matrix c is added to the final result;
- `batch_norm_2d(num_features)`: Batch Normalization over a 4D input as described in [23];
- `conv_2d(cin, cout, k)`: 2D convolution with bias, with $k \times k$ kernel, over a $b \times c_{in} \times h \times w$ input, producing $b \times c_{out} \times h' \times w'$ output;
- `max_pool_2d(k, stride)`: 2D max pooling, with $k \times k$ kernel, and striding;
- `soft_max`: The softmax function

$$\text{softmax}(\mathbf{x}) := \left[\frac{\exp(x_i)}{\sum_j \exp(x_j)} \right]$$

The layers were chosen to cover a range of arithmetic operations (`mulf`, `divf`, `addf`, `subf`, `sqrtd`) and data access patterns (iteration, accumulation, and reduction). The concrete parameter values and dimensions of inputs used during evaluation are summarized in Table II.

See Figure 3 for Vitis HLS vs. BraggHLS resource usage and latency vs. unroll factor and Figure 4 for the runtimes of Vitis HLS as function of increasing unroll factor. We observe that Vitis HLS indeed approaches lower and lower end-to-end latencies as a function of unroll factor but that for none of the unroll factors is it able to match the end-to-end latency achieved by BraggHLS. Even at unroll factor equal to 1024 (which corresponds to fully unrolled for all of the loop nests comprising these layer types), Vitis HLS is only within $10\times$ of BraggHLS. We hypothesize that in general this is due to Vitis HLS’s inability to effectively pipeline due to inability to eliminate memory dependencies, either through store-

TABLE II
DNN LAYERS USED FOR EVALUATION OF BRAGGHLS.

Layer	Parameter values	Input dimensions
addmm	N/A	$\dim(a) = \dim(b) = \dim(c) = (16, 16)$
batch_norm_2d	$\text{num_features} = 2$	$\dim(\text{input}) = (10, 2, 3, 3)$
conv_2d	$c_{in} = 1, c_{out} = 3, k = 3$	$\dim(\text{input}) = (1, 1, 16, 16)$
max_pool_2d	$k = 3, \text{stride} = 2$	$\dim(\text{input}) = (1, 3, 16, 16)$
soft_max	N/A	$\dim(\text{input}) = (1, 3, 16, 16)$

load forwarding or further array partitioning⁵. Conversely, BraggHLS’s ability to effectively perform store-load forwarding is evident in the complete lack of BRAM usage: all weights are kept on FFs or LUTs. While this is infeasible for larger designs (which would be constrained by the number of available FFs), for our particular use case, this unconstrained usage of FFs is acceptable. Note, the increasing latency (as a function of unroll factor) in the case of max_pool_2d is due to Vitis HLS’s failure to meet timing, i.e., while the interval count decreases as a function of unroll factor, the clock period increases (see Appendix Table III for exact values).

B. 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 (followed by 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) these experiments can have collection rates in excess of 80 GB/s. These 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 (as explored employed by LHC experiments) in order to enable experiment modalities that depend on measurement-based feedback (i.e., experiment steering).

BraggNN [29], a DNN aimed at efficiently characterizing Bragg diffraction peaks, achieves a throughput of approximately 22 $\mu\text{s}/\text{sample}$ (via batch inference). This is a large speedup over the classical pseudo-Voigt peak fitting methods, but still falls far short of the 1 $\mu\text{s}/\text{sample}$ 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

```

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

```

Listing 5: BraggNN for scale $s = 1, 2$.

to being deployed at the edge, i.e., adjacent or proximal to the high energy microscopy equipment. Towards this end, we applied BraggHLS to the PyTorch representation of BraggNN($s = 1$) (see Listing 5) and achieved a RTL implementation which synthesizes to a 1,238 interval count 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 the longest stage measuring 480 intervals. Thus, the throughput of the implementation is 4.8 $\mu\text{s}/\text{sample}$. See Table IV for a summary of the resource usage of the implementation and Figure 5 for a comparison with designs generated by Vitis HLS.

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 were made for the purposes of reducing resource consumption. The first was reducing the precision used for the floating-point operations. We reduced the precision from half precision to FloPoCo (5, 4)-

⁵Indeed Vitis HLS can be seen to produce warnings such as
WARNING: [HLS 200-885] Unable to schedule ‘load’
operation ... due to limited memory ports (II = 1).
Please consider using a memory core with more ports
or partitioning the array ...

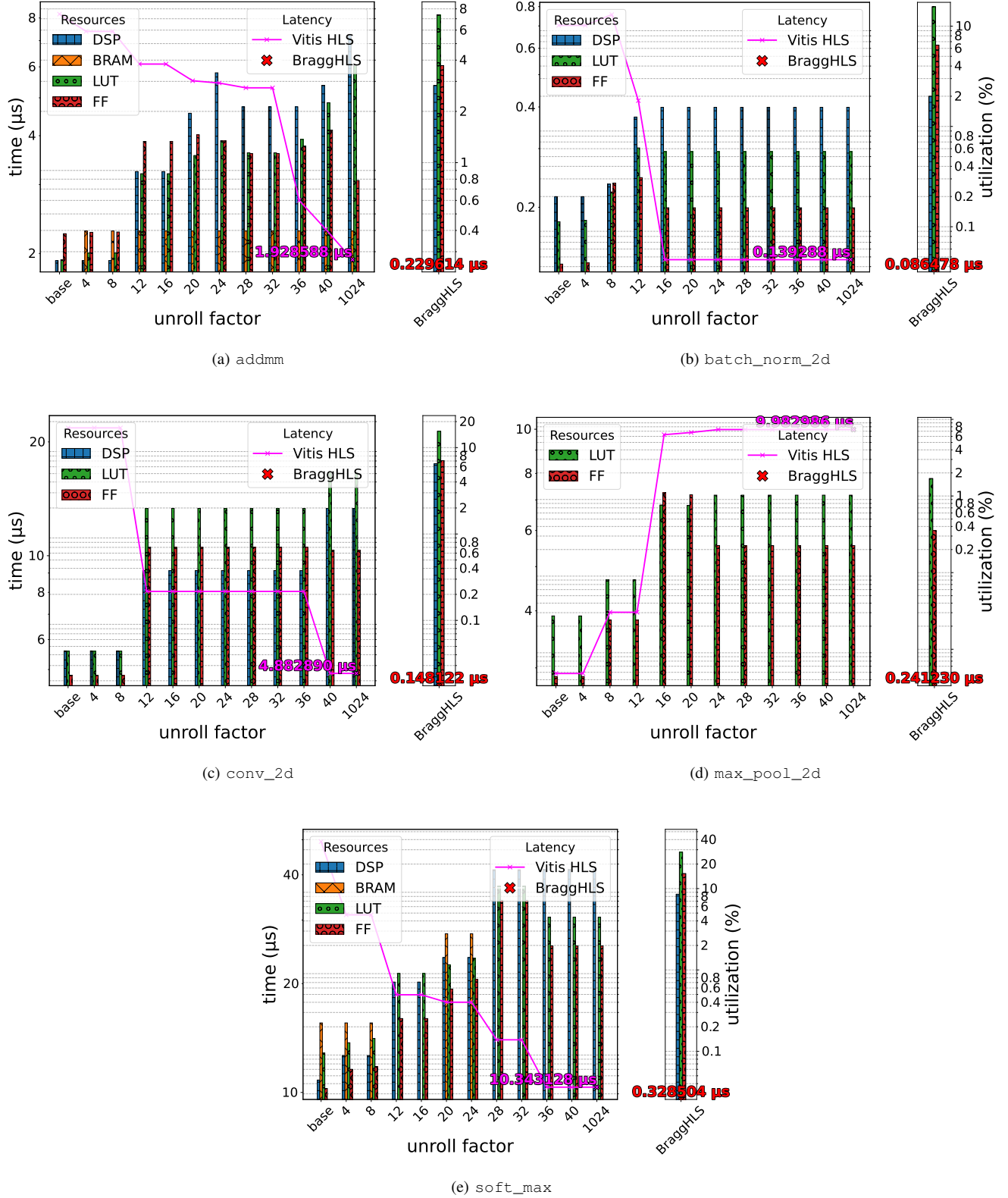


Fig. 3. Vitis HLS vs. BraggHLS resource usage and latency vs. unroll factor of various DNN modules. All y-scales are log.

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 6). Reducing the precision enabled the second design choice: we eliminate

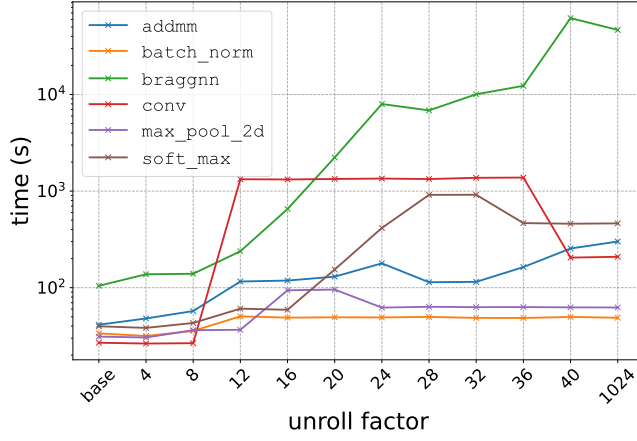


Fig. 4. Runtime of Vitis HLS vs. unroll factor.

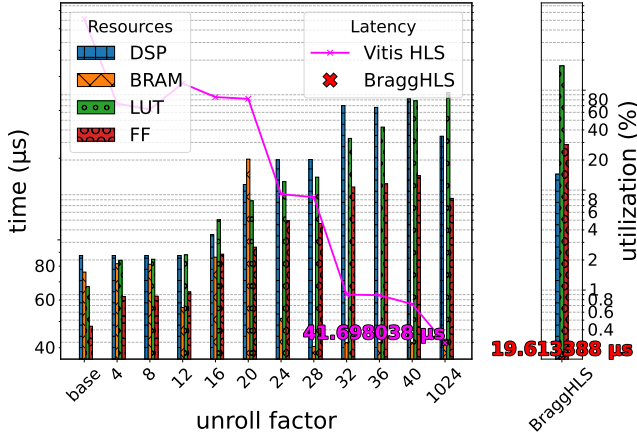


Fig. 5. BraggNN Vitis HLS vs. BraggHLS resource usage and latency vs. unroll factor.

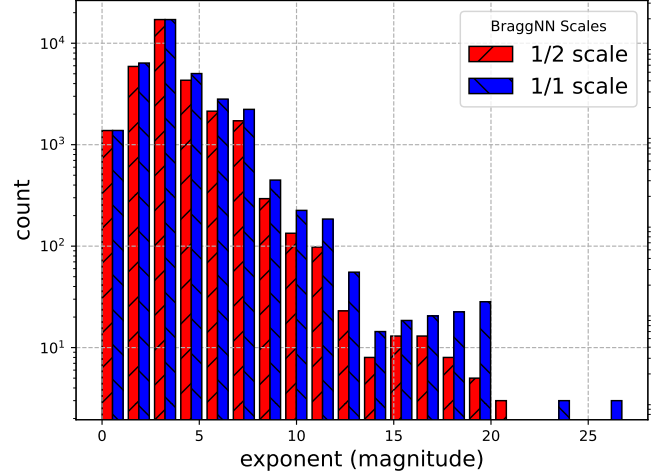


Fig. 6. BraggHLS weights exponent distribution.

BRAMs from the design, since, at the lower precision, all weights can 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 altered. Most likely this is due to the fact that the DSP48 hardware block includes a 18-bit by 25-bit signed multiplier and a 48-bit adder [20], neither of which neatly divide the bit width of FloPoCo (5, 4)-precision cores⁶.

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 [25], which implies 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 spans two SLRs. On the Alveo U280 there are exactly 23,040 SLLs available between adjacent SLRs and at (5, 4)-precision BraggNN ($s = 1$) needs 23,328 SLLs between SLR2 and SLR1⁷. Thus, we further reduced the precision to (5, 3). Finally, since multiple dies constitute independent clock domains, the SLLs that cross SLRs are sensitive to hold time violations due to the higher multi-die variability [12]. This multi-die variability leads to high congestion if not addressed. Thus, routing across SLRs needs to be 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 7 for an illustration on the effect of using launch and latch registers as well as placement and routing constraints.

⁶The 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.

⁷We 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.

- Systems. 2015. DOI: [10.48550/ARXIV.1512.01274](https://arxiv.org/abs/1512.01274). URL: <https://arxiv.org/abs/1512.01274>.
- [9] Tianqi Chen et al. *Training Deep Nets with Sublinear Memory Cost*. 2016. DOI: [10.48550/ARXIV.1604.06174](https://arxiv.org/abs/1604.06174). URL: <https://arxiv.org/abs/1604.06174>.
- [10] Tianqi Chen 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.
- [11] “Comparison of particle selection algorithms for the LHCb Upgrade”. In: (2020). URL: <https://cds.cern.ch/record/2746789>.
- [12] *Create placed and routed DCP to cross SLR*. https://www.rapidwright.io/docs/SLR_Crosser_DCP_Creator_Tutorial.html. Accessed: 2022-10-15.
- [13] Steve Dai, Gai Liu, and Zhiru 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*. FPGA ’18. Monterey, CALIFORNIA, USA: Association for Computing Machinery, 2018, pp. 137–146. ISBN: 9781450356145. DOI: [10.1145/3174243](https://doi.org/10.1145/3174243). URL: <https://doi.org/10.1145/3174243>.
- [14] Florent de Dinechin. “Reflections on 10 Years of FloPoCo”. In: *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. 2019, pp. 187–189. DOI: [10.1109/ARITH.2019.00042](https://doi.org/10.1109/ARITH.2019.00042).
- [15] J. Duarte et al. “Fast inference of deep neural networks in FPGAs for particle physics”. In: *Journal of Instrumentation* 13.07 (2018), P07027–P07027. DOI: [10.1088/1748-0221/13/07/p07027](https://doi.org/10.1088/1748-0221/13/07/p07027). URL: <https://doi.org/10.1088/1748-0221/13/07/p07027>.
- [16] Fabrizio Ferrandi et al. “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.
- [17] V V Gligorov and M Williams. “Efficient, reliable and fast high-level triggering using a bonsai boosted decision tree”. In: *Journal of Instrumentation* 8.02 (2013), P02013–P02013. DOI: [10.1088/1748-0221/8/02/p02013](https://doi.org/10.1088/1748-0221/8/02/p02013). URL: <https://doi.org/10.1088/1748-0221/8/02/p02013>.
- [18] Vladimir 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*. Ed. by Glen Cowan et al. Vol. 42. Proceedings of Machine Learning Research. Montreal, Canada: PMLR, 2015, pp. 1–18. URL: <https://proceedings.mlr.press/v42/glig14.html>.
- [19] Keith Grainge et al. “Square Kilometre Array: The radio telescope of the XXI century”. In: *Astronomy reports* 61.4 (2017), pp. 288–296.
- [20] Advance Specification User Guide. “UltraScale Architecture DSP Slice”. In: ().
- [21] M. Hammer, K. Yoshii, and A. Miceli. “Strategies for on-chip digital data compression for X-ray pixel detectors”. In: *Journal of Instrumentation* 16.01 (2021), P01025–P01025. DOI: [10.1088/1748-0221/16/01/p01025](https://doi.org/10.1088/1748-0221/16/01/p01025). URL: <https://doi.org/10.1088/1748-0221/16/01/p01025>.
- [22] Momoko Hattori, Naoki Kobayashi, and Ryosuke Sato. *Gradual Tensor Shape Checking*. 2022. DOI: [10.48550/ARXIV.2203.08402](https://arxiv.org/abs/2203.08402). URL: <https://arxiv.org/abs/2203.08402>.
- [23] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. DOI: [10.48550/ARXIV.1502.03167](https://arxiv.org/abs/1502.03167). URL: <https://arxiv.org/abs/1502.03167>.
- [24] Chris Lattner et al. *MLIR: A Compiler Infrastructure for the End of Moore’s Law*. 2020. DOI: [10.48550/ARXIV.2002.11054](https://arxiv.org/abs/2002.11054). URL: <https://arxiv.org/abs/2002.11054>.
- [25] Steve Leibson and Nick Mehta. “Xilinx ultrascale: The next-generation architecture for your next-generation architecture”. In: *Xilinx White Paper WP435* 143 (2013).
- [26] Yizhi Liu et al. “Optimizing CNN Model Inference on CPUs”. In: (2018). DOI: [10.48550/ARXIV.1809.02697](https://arxiv.org/abs/1809.02697). URL: <https://arxiv.org/abs/1809.02697>.
- [27] Yongtao Liu et al. “Exploring physics of ferroelectric domain walls in real time: deep learning enabled scanning probe microscopy”. In: *Advanced Science* (2022), p. 2203957.
- [28] Zhengchun Liu et al. “Deep learning accelerated light source experiments”. In: *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 2019, pp. 20–28.
- [29] Zhengchun Liu et al. “BraggNN: fast X-ray Bragg peak analysis using deep learning”. In: *IUCrJ* 9.1 (2022), pp. 104–113.
- [30] Saeed Maleki et al. “An evaluation of vectorizing compilers”. In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 372–382.
- [31] J McMullin et al. “The Square Kilometre Array project update”. In: *Ground-based and Airborne Telescopes IX*. Vol. 12182. SPIE, 2022, pp. 263–271.
- [32] Razvan Nane et al. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604. DOI: [10.1109/TCAD.2015.2513673](https://doi.org/10.1109/TCAD.2015.2513673).
- [33] Julian Oppermann. “Advances in ILP-based Modulo Scheduling for High-Level Synthesis”. en. PhD thesis. Darmstadt: Technische Universität, 2019. URL: <http://tuprints.ulb.tu-darmstadt.de/9272/>.
- [34] Julian Oppermann, Mike Urbach, and John Demme. “How to Make Hardware with Maths: An Introduction to CIRCT’s Scheduling Infrastructure”. In: *2022 European LLVM Developers’ Meeting (EuroLLVM)*. 2022.

- [35] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: (2017).
- [36] Robert M Patton 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.
- [37] Sanjay V Rajopadhye. *Dependence Analysis and Parallelizing Transformations*. 2002.
- [38] Benjamin John Rosser. *Cocotb: a Python-based digital logic verification framework*. 2018.
- [39] Nadav Rotem et al. *Glow: Graph Lowering Compiler Techniques for Neural Networks*. 2018. DOI: [10.48550/ARXIV.1805.00907](https://arxiv.org/abs/1805.00907). URL: <https://arxiv.org/abs/1805.00907>.
- [40] Sean Silva and Anush Elangovan. *Torch-MLIR*. <https://mlir.llvm.org/OpenMeetings/2021-10-07-The-Torch-MLIR-project.pdf>. 2021.
- [41] Stephen Williams. “Icarus Verilog, 1998–2020”. In: URL <http://iverilog.icarus.com/>).
- [42] Hanchen Ye et al. “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.
- [43] Zhiru Zhang et al. “AutoPilot: A Platform-Based ESL Synthesis System”. In: *High-Level Synthesis: From Algorithm to Digital Circuit*. Ed. by Philippe Coussy and Adam Morawiec. Dordrecht: Springer Netherlands, 2008, pp. 99–112. ISBN: 978-1-4020-8588-8. DOI: [10.1007/978-1-4020-8588-8_6](https://doi.org/10.1007/978-1-4020-8588-8_6). URL: https://doi.org/10.1007/978-1-4020-8588-8_6.
- [44] S. Zheng et al. “NeoFlow: A Flexible Framework for Enabling Efficient Compilation for High Performance DNN Training”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.11 (2022), pp. 3220–3232. ISSN: 1558-2183. DOI: [10.1109/TPDS.2021.3138862](https://doi.org/10.1109/TPDS.2021.3138862).

APPENDIX

TABLE III: Latency, resource usage, and runtimes for all BraggHLS evaluations.

module	metric_name unroll_factor	metric_val				Latency	Clock Period	Runtime
		BRAM	DSP	FF	LUT			
addmm	0	0.00E+00	2.66E-01	3.83E-01	2.69E-01	1334	6.13E+00	4.14E+01
	4	3.97E-01	2.66E-01	3.90E-01	2.94E-01	1205	6.13E+00	4.80E+01
	8	3.97E-01	2.66E-01	3.91E-01	2.95E-01	1205	6.13E+00	5.73E+01
	12	3.97E-01	8.87E-01	1.33E+00	8.61E-01	993	6.14E+00	1.16E+02
	16	3.97E-01	8.87E-01	1.33E+00	8.61E-01	993	6.14E+00	1.19E+02
	20	3.97E-01	1.95E+00	1.46E+00	1.10E+00	900	6.14E+00	1.30E+02
	24	3.97E-01	3.37E+00	1.34E+00	1.35E+00	888	6.14E+00	1.79E+02
	28	3.97E-01	2.13E+00	1.13E+00	1.14E+00	821	6.46E+00	1.14E+02
	32	3.97E-01	2.13E+00	1.13E+00	1.14E+00	821	6.46E+00	1.15E+02
	36	3.97E-01	2.13E+00	1.25E+00	1.37E+00	425	6.43E+00	1.63E+02
	40	3.97E-01	2.84E+00	1.55E+00	2.25E+00	359	6.42E+00	2.55E+02
batch_norm	1024	3.97E-01	5.67E+00	7.86E-01	3.74E+00	314	6.14E+00	3.00E+02
	0	0.00E+00	1.99E-01	4.24E-02	1.12E-01	116	6.06E+00	3.36E+01
	4	0.00E+00	1.99E-01	4.36E-02	1.16E-01	116	6.06E+00	3.16E+01
	8	0.00E+00	2.66E-01	2.72E-01	2.22E-01	125	6.06E+00	3.57E+01
	12	0.00E+00	1.24E+00	3.09E-01	6.11E-01	69	6.06E+00	5.05E+01
	16	0.00E+00	1.55E+00	1.55E-01	5.61E-01	23	6.06E+00	4.90E+01
	20	0.00E+00	1.55E+00	1.55E-01	5.61E-01	23	6.06E+00	4.94E+01
	24	0.00E+00	1.55E+00	1.55E-01	5.61E-01	23	6.06E+00	4.92E+01
	28	0.00E+00	1.55E+00	1.55E-01	5.61E-01	23	6.06E+00	4.98E+01
	32	0.00E+00	1.55E+00	1.55E-01	5.61E-01	23	6.06E+00	4.86E+01
	36	0.00E+00	1.55E+00	1.55E-01	5.61E-01	23	6.06E+00	4.85E+01
braggnn	40	0.00E+00	1.55E+00	1.55E-01	5.61E-01	23	6.06E+00	4.98E+01
	1024	0.00E+00	1.55E+00	1.55E-01	5.61E-01	23	6.06E+00	4.88E+01
	0	1.51E+00	2.22E+00	4.34E-01	1.09E+00	85851	7.59E+00	1.05E+02
	4	1.84E+00	2.22E+00	8.61E-01	1.99E+00	49310	6.45E+00	1.38E+02
	8	1.81E+00	2.22E+00	8.69E-01	2.03E+00	49257	6.14E+00	1.39E+02
	12	6.70E-01	2.22E+00	9.56E-01	2.25E+00	61499	6.14E+00	2.38E+02
	16	2.13E+00	3.59E+00	2.29E+00	5.04E+00	52515	6.40E+00	6.52E+02
	20	2.04E+01	1.14E+01	2.69E+00	7.82E+00	51768	6.40E+00	2.23E+03
	24	5.21E-01	2.01E+01	4.95E+00	1.22E+01	24004	6.14E+00	7.97E+03
	28	2.73E-01	2.02E+01	4.64E+00	1.35E+01	22377	6.42E+00	6.85E+03
	32	2.98E-01	7.01E+01	1.07E+01	3.29E+01	9793	6.42E+00	1.01E+04
conv	36	2.98E-01	6.73E+01	1.15E+01	4.27E+01	9730	6.42E+00	1.23E+04
	40	2.98E-01	8.21E+01	1.40E+01	7.83E+01	9032	6.42E+00	6.18E+04
	1024	3.22E-01	3.46E+01	8.27E+00	9.48E+01	6789	6.14E+00	4.67E+04
	0	0.00E+00	4.43E-02	2.30E-02	4.40E-02	3551	6.13E+00	2.69E+01
	4	0.00E+00	4.43E-02	2.30E-02	4.40E-02	3551	6.13E+00	2.64E+01
	8	0.00E+00	4.43E-02	2.30E-02	4.40E-02	3551	6.13E+00	2.66E+01
	12	0.00E+00	3.77E-01	7.04E-01	1.97E+00	1310	6.14E+00	1.33E+03
	16	0.00E+00	3.77E-01	7.03E-01	1.98E+00	1310	6.14E+00	1.32E+03
	20	0.00E+00	3.77E-01	7.03E-01	1.98E+00	1310	6.14E+00	1.34E+03
	24	0.00E+00	3.77E-01	7.03E-01	1.97E+00	1310	6.14E+00	1.35E+03
	28	0.00E+00	3.77E-01	7.03E-01	1.97E+00	1310	6.14E+00	1.33E+03
	32	0.00E+00	3.77E-01	7.03E-01	1.97E+00	1310	6.14E+00	1.37E+03
	36	0.00E+00	3.77E-01	7.03E-01	1.97E+00	1310	6.14E+00	1.38E+03
	40	0.00E+00	1.97E+00	6.49E-01	5.07E+00	795	6.14E+00	2.05E+02

Continued on next page

TABLE III: Latency, resource usage, and runtimes for all BraggHLS evaluations.

module	metric_name unroll_factor	metric_val				Latency	Clock Period	Runtime
		BRAM	DSP	FF	LUT			
max_pool_2d	1024	0.00E+00	1.97E+00	6.49E-01	5.07E+00	795	6.14E+00	2.09E+02
	0	0.00E+00	0.00E+00	4.41E-03	2.72E-02	889	3.28E+00	3.11E+01
	4	0.00E+00	0.00E+00	4.41E-03	2.72E-02	889	3.28E+00	3.05E+01
	8	0.00E+00	0.00E+00	2.40E-02	8.04E-02	641	6.20E+00	3.63E+01
	12	0.00E+00	0.00E+00	2.40E-02	8.04E-02	641	6.20E+00	3.66E+01
	16	0.00E+00	0.00E+00	1.10E+00	7.63E-01	419	2.32E+01	9.40E+01
	20	0.00E+00	0.00E+00	1.04E+00	7.51E-01	412	2.39E+01	9.56E+01
	24	0.00E+00	0.00E+00	2.25E-01	1.03E+00	262	3.81E+01	6.22E+01
	28	0.00E+00	0.00E+00	2.25E-01	1.03E+00	262	3.81E+01	6.34E+01
	32	0.00E+00	0.00E+00	2.25E-01	1.03E+00	262	3.81E+01	6.29E+01
	36	0.00E+00	0.00E+00	2.25E-01	1.03E+00	262	3.81E+01	6.30E+01
	40	0.00E+00	0.00E+00	2.25E-01	1.03E+00	262	3.81E+01	6.25E+01
soft_max	1024	0.00E+00	0.00E+00	2.25E-01	1.03E+00	262	3.81E+01	6.24E+01
	0	2.23E-01	4.43E-02	3.49E-02	9.52E-02	7742	6.38E+00	3.99E+01
	4	2.23E-01	8.87E-02	6.01E-02	1.27E-01	5053	6.13E+00	3.83E+01
	8	2.23E-01	8.87E-02	6.50E-02	1.44E-01	5053	6.13E+00	4.31E+01
	12	0.00E+00	7.09E-01	2.53E-01	9.04E-01	3037	6.13E+00	6.06E+01
	16	0.00E+00	7.09E-01	2.53E-01	9.04E-01	3037	6.13E+00	5.90E+01
	20	2.78E+00	1.42E+00	5.82E-01	1.16E+00	2893	6.14E+00	1.54E+02
	24	2.78E+00	1.42E+00	7.64E-01	1.39E+00	2893	6.14E+00	4.15E+02
	28	0.00E+00	1.70E+01	7.17E+00	1.07E+01	2277	6.14E+00	9.14E+02
	32	0.00E+00	1.70E+01	7.17E+00	1.07E+01	2277	6.14E+00	9.16E+02
	36	0.00E+00	1.72E+01	1.97E+00	4.45E+00	1684	6.14E+00	4.66E+02
	40	0.00E+00	1.72E+01	1.97E+00	4.45E+00	1684	6.14E+00	4.60E+02
	1024	0.00E+00	1.72E+01	1.97E+00	4.45E+00	1684	6.14E+00	4.63E+02

TABLE IV
RESOURCE USAGE FOR BRAGGNN ($s = 1$) 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 V
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			