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 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 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) [**?**]. 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 [**?**], 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 [**?**], will sustain data rates in excess of 10 TB/s [**?**].

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). 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 [**?**] or machine learning models [**?**]; 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) [**?**]. 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 [**?**]. 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 [**?**], [**?**], [**?**] but their use for the lowest latency use cases has been very limited [**?**]. 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 [**?**]);
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 [**?**], TensorFlow [**?**], MXNet [**?**]), 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) [**?**]. While several such HLS tools exist [**?**], [**?**], [**?**] and despite, often, bundling robust optimizing compilers, they struggle to effectively perform the necessary optimizations in reasonable amounts of time (see Section II-B1).

Recently, deep learning compilers (such as TVM [**?**], MLIR [**?**], and Glow [**?**]) have demonstrated the ability to dramatically reduce inference latencies [**?**], training times [**?**], and memory usage [**?**] 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 [**?**], vectorization [**?**], and memory planning [**?**]). 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 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 ultra-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 ultra-low latency RTL designs suitable for deployment to Xilinx FPGAs. `BraggHLS` is thoroughly tested, open source, and available at https://github.com/makslevental/bragghls/;
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 challanges 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 µs/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 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` 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 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 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 be inferred from the textual representation (i.e., the python source) of the DNN. Furthermore, a priori, the runtime-materialized DFG is only partially materialized[1], 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 produces 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 [**?**]) 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 [**?**] 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 [**?**], 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 [**?**].

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` (structured control flow) dialects. The `scf` 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 [**?**], i.e., techniques that enable loop dependence analysis and loop transformations. Such loop transformations, particularly loop unrolling, are crucial for achieving lowest possible latencies [**?**] because they directly inform the concurrency and parallelism of the final RTL design.

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

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++ [**?**], [**?**]. In particular, Xilinx's Vitis HLS, based on the Autopilot project [**?**], 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 completion time for each operation;
   c) The user's desired clock rate/frequency;
2) HLS associates (called *binding*) operations to particular 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, 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.

While HLS aims to 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

```
def conv2d(
  input: MemRef(b, c_in, h, w),
  output: MemRef(b, c_out, h, w),
  weight: MemRef(c_out, c_in, k, k)
):
  for i1 in range(0, b):
    for i2 in range(0, c_out):
      for i3 in range(0, h):
        for i4 in range(0, w):
          for i5 in range(0, c_in):
            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.

```
1  def conv2d(
2    input: MemRef(b, c_in, h, w),
3    output: MemRef(b, c_out, h, w),
4    weight: MemRef(c_out, c_in, k, k)
5  ):
6    for i1 in range(0, b):
7      for i2 in range(0, c_out):
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).

is expensive and grows (in runtime) as better 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 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 [**?**]. 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 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 **??**).

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 logi-

cal 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 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

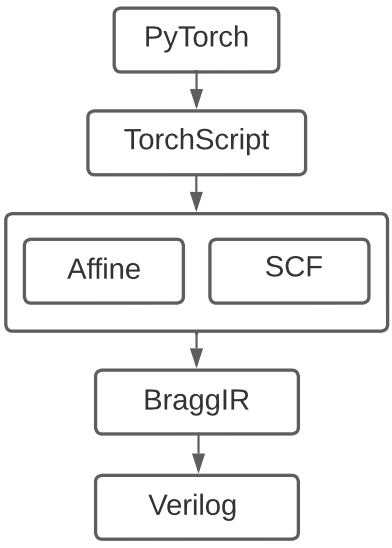


Fig. 1. `BraggHLS` framework overview (placeholder).

lowering 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

```
@conv2d(
    %input: memref<b × c_in × h × w>,
    %weight: memref<b × c_out × h × w>,
    %output: memref<c_out × c_in × k × k>
) {
  scf.for %i1 = %c0 to b step %c1 {
    scf.for %i2 = %c0 to c_out step %c1 {
      scf.for %i3 = %c0 to h step %c1 {
        scf.for %i4 = %c0 to w step %c1 {
          scf.for %i5 = %c0 to c_in 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.

itself to a 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 [**?**] (an MLIR adjacent project). After scheduling, `BraggHLS` emits corresponding RTL (as Verilog).

`BraggHLS` delegates to the FloPoCo [**?**] IP generator the task of generating pipelined implementations of the standard floating-point arithmetic operations (`mulf`, `divf`, `addf`, `subf`, `sqrtf`) 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`); negation (`neg`); rectified linear units (`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. 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 (see Figure 2). Translating `scf` dialect to python enables

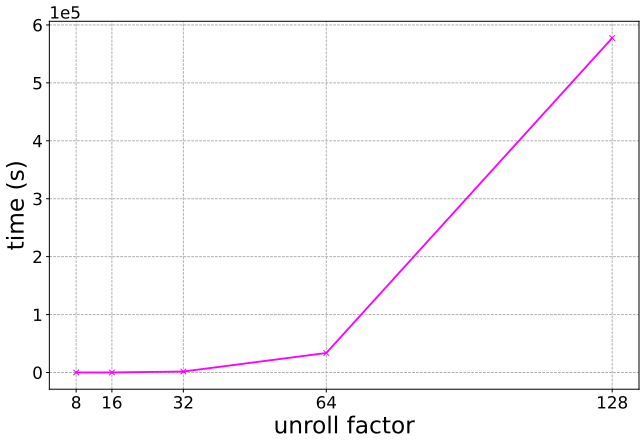


Fig. 2. 3×3-kernel convolution full unrolling time.

`BraggHLS` 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 assignments and reads on those symbol tables;
4) Easily swap evaluation rules in order to support various functionality modes, e.g., evaluating the floating-point arithmetic operations using (python) bindings to FloPoCo's C++ functional models thereby enabling behavioral verification.

See Table I for the translation rules from `scf` dialect to python; with

B. AST transformations and behavioral verification

Prior to intepretation, `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**: DNN `relu` operations are lowered to the `scf` dialect as a decomposition of `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`);
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 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` [**?**] and `iverilog` [**?**], as a part of our continuous integration process[3].

C. Scheduling

Recall that one of the critical functions which HLS fulfills is the scheduling of operations during each clock cycle, in

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

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

TABLE I
SCF DIALECT TO PYTHON TRANSLATION.

scf dialect	python
⟦%5⟧	v5 = Val("%5")
⟦memref<$b \times c_{in} \times h \times w$>⟧	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⟧.__setitem__((⟦%i1⟧, ⟦%i5⟧, ⟦%3⟧, ⟦%4⟧), ⟦%9⟧)
⟦scf.for %i1 = %c0 to b step %c1⟧	for ⟦%iv1⟧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 %64 = arith.select %63, %c0 ⟧	⟦%64⟧= ⟦%10⟧.relu()
⟦ %8 = arith.mulf %5, %6 %9 = arith.addf %7, %8 ⟧	⟦%9⟧= fma(⟦%5⟧, ⟦%6⟧, ⟦%7⟧)

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 [**?**]. 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 particularly 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 [**?**].

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 reprepresented 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 instances[4]. 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

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

where %c1 $\times \mathbb{N}$ represents all multiples of %c1. That is to say, K_i is 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 $j = 1, \ldots, K$, 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 interpretation[5]. Note this ordering coincides with the higher-level

[4]Data-flow within a loop body must still be respected.

[5]BraggHLS only needs to construct a partial precedence ordering $\texttt{op}_a < \texttt{op}_b$ for operations $\texttt{op}_a, \texttt{op}_b$ which CIRCT then combines with the delays of the operations to construct constraints such as $\texttt{start\_op}_a + \texttt{delay}_a \leq \texttt{start\_op}_b$.

```
@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 (%i1, %i2, %i3, %i4) =
               (%c0, %c0, %c0, %c0) to
               (b, c_out, h, w) step
               (%c1, %c1, %c1, %c1) {
    scf.for %i5 = %c0 to c_in 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 4: Parallel loop representation of the convolution in Listing 1.

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

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 `scf.parallel` loop-nests but lower to sequential loop nests (e.g., `sum`, `max`, or `prod`), we fully unroll the loops and transform the resulting, sequential, operations to a reduction tree; we use As-Late-As-Possible scheduling [**?**] amongst the subtrees of such reduction trees.

IV. Evaluation

We evaluate `BraggHLS both` on individual DNN kernels (i.e., layers) and end-to-end, on our use-case `BraggNN`.

A. *DNN layers*

asdasd

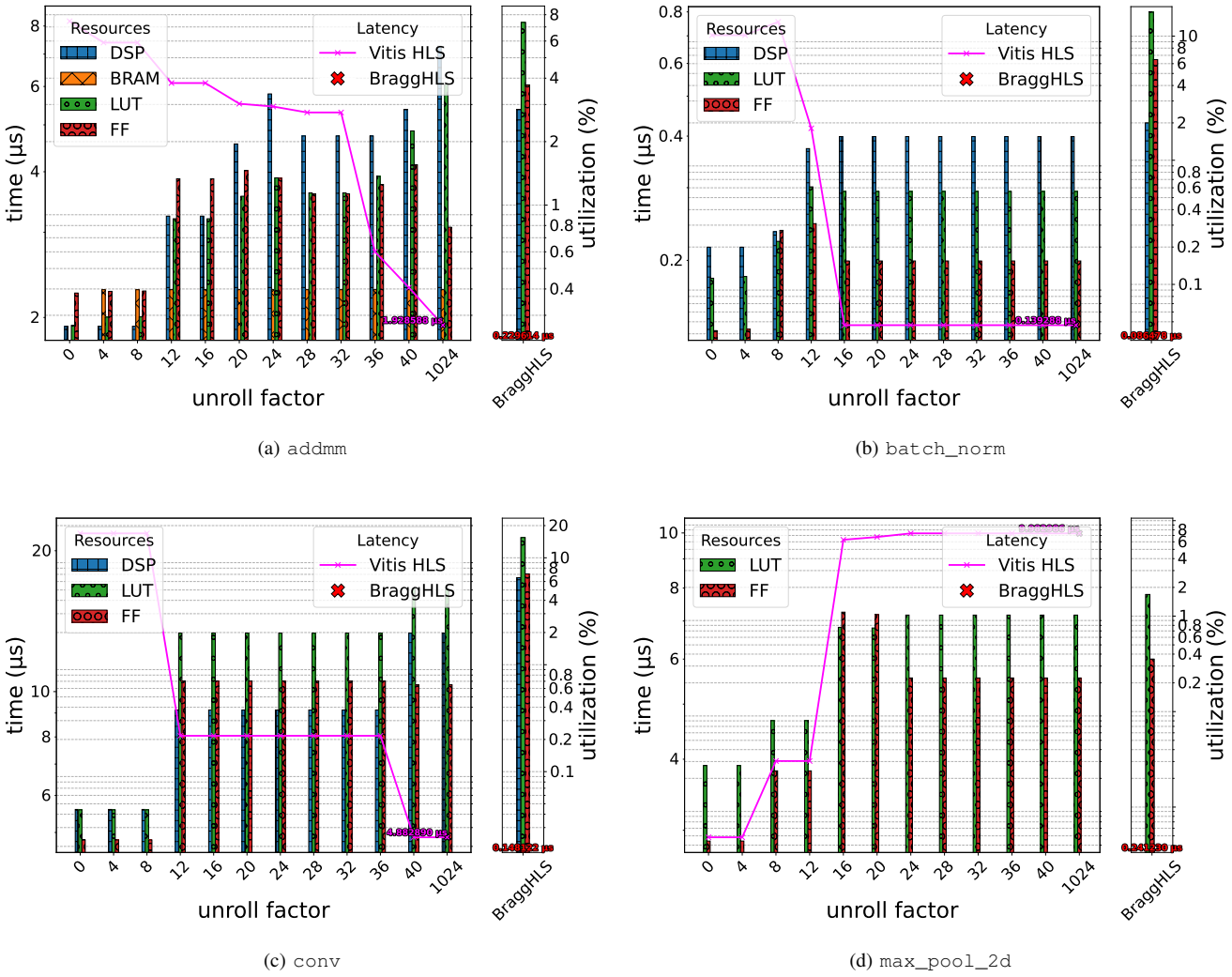
B. *`BraggNN` case study*

TODO: put the comparison to both the dataflow pipelined numbers and the fully unrolled numbers for braggnn here (save the other eval section for just ops).

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` [**?**], 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 5) 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 II for a summary of the resource usage of the implementation.

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 pur-



(a) `addmm` (b) `batch_norm`

(c) `conv` (d) `max_pool_2d`

Fig. 3. Resource usage and latency vs. unroll factor of various DNN modules.

poses 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 6). 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 [**?**], neither of which neatly divide the bit width of FloPoCo $(5, 4)$-precision cores[6].

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 [**?**], 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[7]. Thus, we further reduce the precision to $(5, 3)$. Finally, since multiple dies constitute

[6]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.

[7]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.

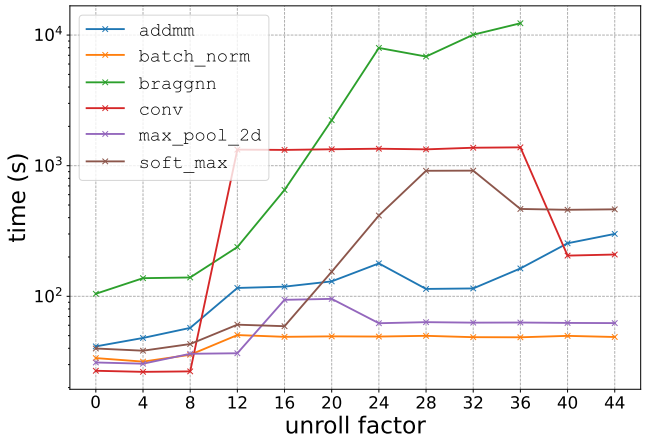


Fig. 4. Runtime of Vitis HLS vs. unroll factor.

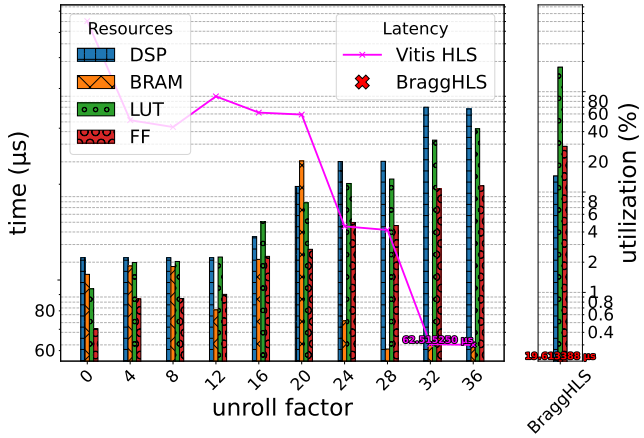


Fig. 5. `braggnn`

independent clock domains, the SLLs that cross SLRS are sensitive hold time violations due to the higher multi-die variability [**?**]. 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 7 for an illustration on the effect of using launch and latch registers as well as placement and routing constraints.

V. Conclusion

287 + 471 + 480

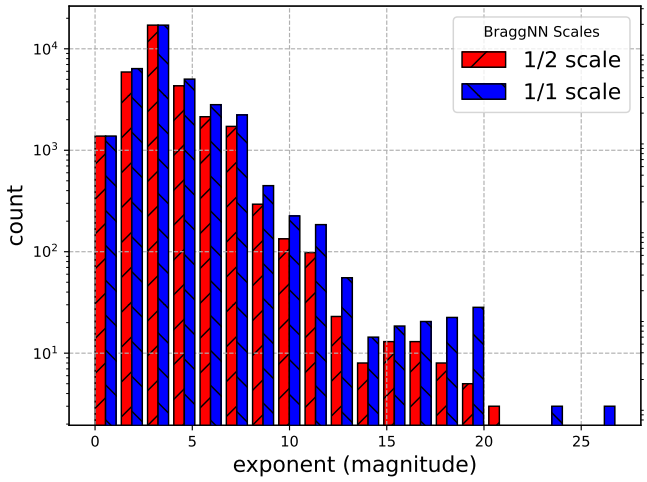
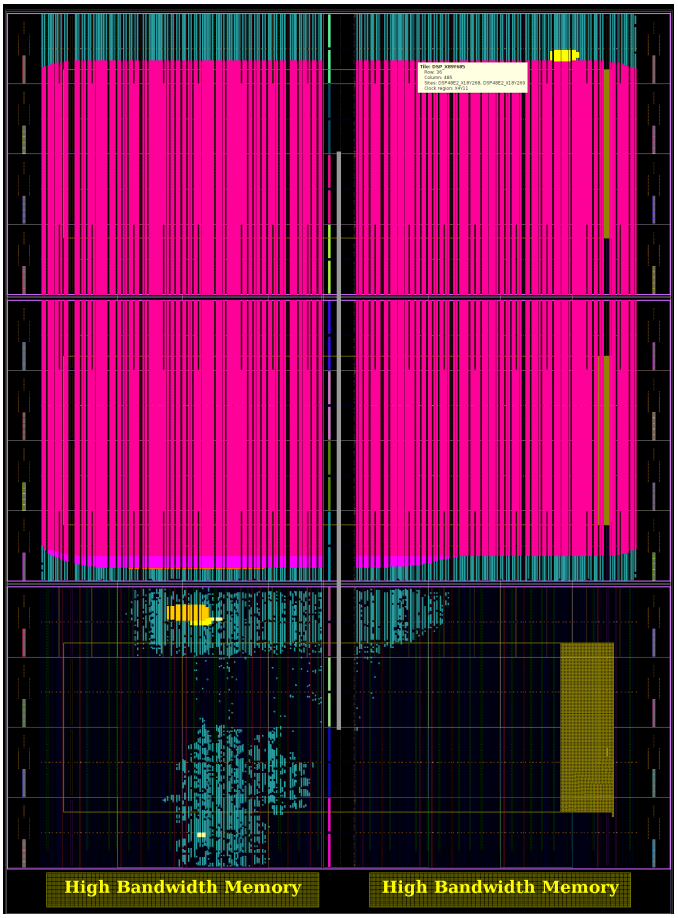
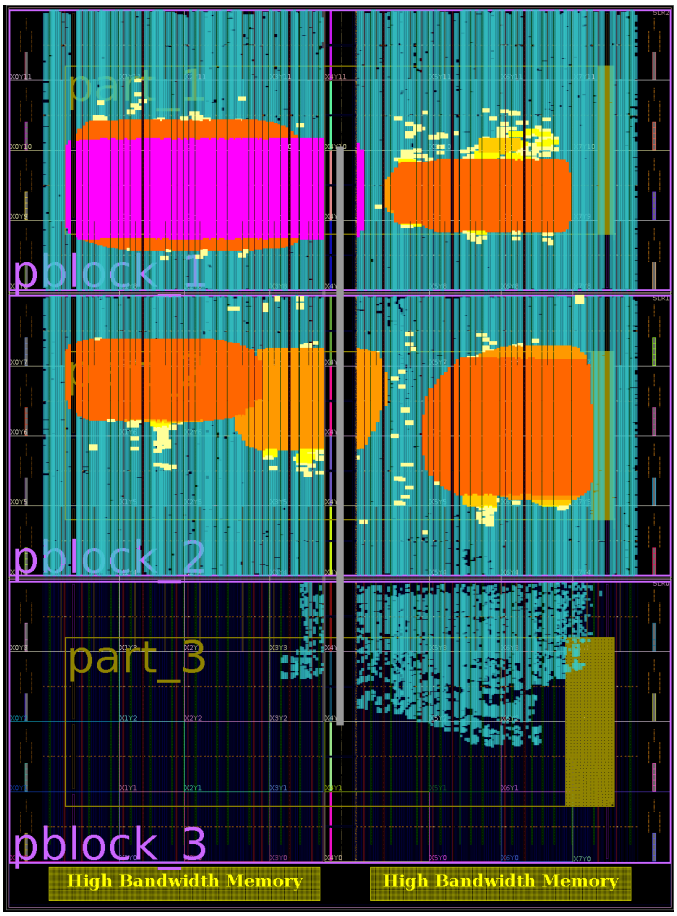


Fig. 6. `BraggHLS` weights exponent distribution.



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

TABLE II
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 III
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			