Kalpa Publications in Computing
Volume 17, 2023, Pages 48–68
Proceedings of 2023 Concurrent Processes Architectures and Embedded Systems Hybrid Virtual Conference

Evaluation of FPGA Acceleration of Neural Networks

Emil Stevnsborg[1], Sture Oksholm[1], Carl-Johannes Johnsen[1], and James Emil Avery[2]

[1] Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
stevnsborgemil@gmail.com : 0009-0005-0176-949X,
sture.oksholm1@gmail.com ,
carl-johannes@di.ku.dk : 0000-0002-4835-8891
[2] Dep. of Electro- and Computer Engineering, Aarhus University, Aarhus, Denmark
avery@ece.au.dk : 0000-0001-7772-3440

Abstract

This paper explores real-time Convolutional Neural Network inference on Field Programmable Gate Arrays (FPGAs) implemented in Synchronous Message Exchange (SME). We compare SME to the widespread FPGA tool, High-Level Synthesis (HLS), and compare both the SME and HLS implementations of CNNs with the PyTorch implementation for CNN on CPU/GPU. We find that the SME implementation is more flexible than the HLS implementation as it allows for more customization of the hardware. Programming with SME is more difficult than HLS, although easier than traditional Hardware Description Languages. Finally, for a test use case, we find that the SME implementation on FPGA is approximately 2.8/1.4/2.0 times more energy efficient than CPU/GPU/ARM at larger batch sizes, with the HLS implementation on FPGA falling in between CPU/ARM and GPU in terms of energy efficiency. At a batch size of 1, appropriate for edge-device inference, the gap in energy efficiency between the FPGA and CPU/GPU/ARM implementations becomes more pronounced, with the SME implementation on FPGA being approximately 83/47/8 times more energy efficient than the CPU/GPU/ARM implementations, and with the HLS implementation on FPGA being approximately 40/23/4 times more energy efficient than the CPU/GPU/ARM implementations.

1 Introduction

In recent years, Artificial Intelligence (AI) has seen a huge growth in popularity. This has been made possible by the development of complex Neural Network (NN) architectures, which have opened the possibility of using AI to solve increasingly more complex tasks. NNs are composed of numerous computational subtasks that can be executed in parallel, necessitating the use of concurrent hardware to achieve high-speed performance.

Due to their data parallel performance, Graphical Processing Units (GPUs) have become an industry standard for large NN applications. In comparison to Central Processing Units (CPUs), GPUs perform far better in terms of speed both for training and inference tasks, when given large enough batches of samples. However, due to the heavy power consumption of GPUs, it is not favorable to locally install GPUs in power-restricted edge devices (such as drones) that seek to implement AI applications. Instead, the inference for applications in power-restricted applications takes place in remote servers with GPUs installed, bringing on a new range of problems such as latency and the possibility of an unreliable connection.

By developing hardware that is specifically designed to run NNs in AI applications, known as an Application Specific Integrated Circuit (ASIC), one can alleviate the problem of power consumption. However, this also comes with drawbacks, namely, ASICs are very expensive per unit at lower unit counts, and take months to develop to ensure the correctness of the circuits. Furthermore, the circuitry of ASICs cannot be changed, which means future hardware updates cannot be applied to a developed batch of ASICs.

By trading specificity for flexibility, Field-Programmable Gate Arrays (FPGAs) share some of the same advantages as ASICs, such as being able to capture more parallelism, being focused on a specific application, and exhibiting low power consumption. Furthermore, they are reprogrammable, allowing FPGAs to leverage many advantages of ASICs, while also being able to adapt to future hardware updates. As a result, FPGAs are a very attractive alternative to GPUs and ASICs for AI applications in power-restricted edge devices. However, FPGAs share one of the same drawbacks as ASICs, namely, they are notoriously difficult to program for software developers as they are primarily programmed with Hardware Description Languages (HDLs) such as Very high-speed integrated circuit HDL (VHDL) and Verilog. Both of these languages specify the hardware at the (fully parallel) wire level, which is difficult for programmers to reason about [20], making the process of programming FPGAs a substantial task.

1.1 Contribution

In this paper, we explore a high-level approach for implementing Convolutional Neural Network (CNN) (a special type of NN) on FPGAs; namely Synchronous Message Exchange (SME) [4, 5, 11, 17, 18, 21, 22]. Being a high-level FPGA programming model, it is only natural to compare our implementation to the most widespread high-level FPGA programming model: High-Level Synthesis (HLS) [2]. These two approaches are on opposite ends of the spectrum (concerning hardware development) where HLS is a high-level approach, and SME is a low-level approach. We will explore the advantages and disadvantages of each approach, and compare the performance of the implementations created with the two approaches with each other and a PyTorch implementation on CPU and GPU [14]. Both the HLS and SME implementations are cross-validated against the PyTorch implementation to ensure consistent results given identical inputs.

The implementations are configurable through a user's desired configuration, such as the number of input and output channels, and the size and stride of the kernels if relevant. In SME, we created multiple implementation versions for some layers, which vary in the degree of parallelism. The choice between these versions balances the performance and resource consumption of the hardware, allowing the user to tune the implementation for their desired FPGA. The source code is available on Github [8].

2 Programming FPGAs

FPGAs, unlike the fixed instruction set architecture of CPUs and GPUs, offer reconfigurable hardware that suits user needs. This combines ASIC-like efficiency with the flexibility of future hardware updates. Similar to ASICs, FPGAs are employed through Hardware Description Languages (HDLs) for hardware design, rather than software development.

HDLs operate at an intricate, parallel level, which renders programming notably more challenging than conventional software development. Adding to this complexity, studies have demonstrated the inherent difficulty of parallel programming for humans [20], compounding the challenges of FPGA programming. While opting for sequential designs is possible, it is suboptimal due to the resulting extended critical path, constraining the design's clock frequency and overall performance. The shortcomings of HDLs are worsened by their excessive verbosity (increasing error likelihood), incomplete adoption of new language features (only partial support for 2008 standards by vendors), and absence of contemporary software development tools like debuggers and libraries.

Various tools address the issue of FPGA HDL code generation. This paper examines High-Level Synthesis (HLS) [2] and Synchronous Message Exchange (SME) [11] approaches, spanning high and low levels respectively. While other tools exist, some even beyond the span of HLS and SME, they will not be covered in this paper.

2.1 High-Level Synthesis (HLS)

HLS is a tool that enables the user to write C/C++ code, which is then compiled into HDL code, in turn greatly boosting productivity over writing HDL code directly. By targeting C/C++, HLS enables a wide range of software developers to utilize FPGAs as they can write code in a familiar environment. However, the user still has to keep hardware architecture in mind when writing code. If the user writes the code in a traditional, sequential manner, the generated code will not be optimized for the FPGA, since the compiler will adhere to the semantics of a traditional C/C++ program (e.g. sequential execution), rather than the truly parallel nature of the FPGA.

To circumvent this problem, the user can use pragma directives to guide the compiler toward a more optimal solution. Examples of optimizations are pipelining (exploiting pipeline parallelism), unrolling (exploiting hardware replication), array partitioning (exploiting data locality), loop flattening (removing unnecessary control logic), and loop dependency removal (removing unnecessarily aggressive memory management). While guiding the compiler does require the user to know hardware architecture, it is a powerful tool for expressing complex hardware designs in a high-level language.

HLS supports many different design strategies but favors dataflow architectures as it can exploit the parallel nature of FPGAs. As such, if the user writes the program using this design strategy, the compiler will be able to generate a more optimal design. Finally, during development, the user can co-simulate the hardware design with the software code, which enables the user to verify that the hardware design behaves as expected.

2.2 Synchronous Message Exchange (SME)

SME is a Communicating Sequential Processes (CSP) [10]-derived programming model, enabling the user to write concurrent programs in a C# environment. Like CSP, SME is based on the concept of sequential processes that share nothing but their means of communication, which for CSP is the concept of channels, and for SME is the concept of buses. A bus acts as a broadcast channel in CSP terms, where every connected process receives messages synchronized to a global clock, eliminating the need for the sender to wait for the receiver. Some of the processes in an SME program are executed in parallel and are synchronized by a global clock signal triggering them (known as *clocked processes*), while others are executed in sequence and are triggered by communication on the buses (known as *unclocked processes*). Regardless of which synchronization strategy is used, each process is triggered exactly once per simulated clock cycle, which executes the process to completion for simple processes, or until a synchronization point for more complex processes, which are not covered in this paper. These processes are to be synthesized into HDL code, preventing them from using dynamic constructs, such as dynamic memory allocation, recursion, and dynamic instantiation of processes as these cannot be expressed as hardware.

SME has been designed to fit the architecture of FPGAs, which means that the legal constructs of SME are also legal constructs in the generated HDL code. As a result, while SME is a higher-level programming model, compared to HDLs, it is still very close to the hardware, so the user has to keep the hardware architecture in mind when writing code. By being implemented in C#, SME enables the features of a modern programming language, such as debuggers, libraries (to some degree), generics, inheritance, and interfaces, which makes the process of writing code easier and boosts productivity.

Once the network of processes and how they are connected have been defined, the network is simulated to verify that the program behaves as expected. This is done by describing *simulation processes*, which share the same interface as the processes that are to be synthesized, but allow for the entire software suite of C# to be used without limitation. A simulation process is always a clocked process, but the control of when to synchronize to the global clock signal is left to the user through asynchronously waiting for the simulation to tick the clock signal. The simulation itself is done by SME building a dependency graph of how the processes are connected and running the following simulation loop:

1. Trigger all of the clocked processes to run in parallel.
2. Wait for all of the clocked processes to finish.
3. Propagate all of the written values on the output buses of the just triggered processes to the connected processes.
4. Trigger all of the processes whose entire collection of input buses have been written to.
5. Repeat 3-4 until all of the processes in the network have finished.
6. Repeat 1-5 until all of the simulation processes have finished.

Step 1-5 simulates an entire clock cycle, which means that the simulation loop is synchronized to the global clock signal. It is important to note that two clocked processes cannot communicate within a single clock cycle as SME mimics the behavior of actual hardware, where such a setup would lead to short-circuiting, potentially damaging the hardware. For the same reason, there cannot exist an unclocked dependency cycle within the execution graph.

During this simulation, all of the values that are on every bus at every clock cycle are collected into a trace file. This trace file is then emitted by SME, alongside VHDL implementations for each process, a top-level VHDL file instantiating and connecting the processes, and a VHDL testbench. This testbench loads the trace file and drives all of the top-level inputs (both the clock signal and the buses) to the network, followed by verifying that every bus contains the expected value for each clock cycle. This verifies that the generated VHDL implementation is clock-cycle accurate to the C# simulation, given the same input. While the primary function of the simulation is to verify the correctness of the program, it also provides a good entry point for further debugging as the user can compare the values on the buses at every clock cycle.

2.3 Interfacing with the FPGA

Until now, we have covered HDL code generation for computation but have not addressed FPGA interfacing. Our approach involves utilizing the FPGA as an accelerator in a host system, serving as an offload device. The host handles data transfer and FPGA control, so we need a driver to establish FPGA communication.

This interfacing is not directly part of HDL code, but rather a part of the software suite for FPGA interaction, often termed the ”driver” or ”shell.” Typically, larger boards come with vendor-provided shells, allowing for ease of use. However, we will make a specialized driver as we target smaller boards that lack a shell due to resource constraints.

Our setup is straightforward: the host system interfaces with the on-chip FPGA via shared off-chip memory access. The host transfers data to this memory, conveys input/output addresses and sizes to FPGA registers, and triggers processing initiation. The FPGA's driver reads address and size data, streams it from off-chip memory into our HDL kernels, and upon kernel completion, streams output data back to memory. The host is notified of processing completion. Since we know the sizes of the output, we do not need this information back from the FPGA to the host.

3 Convolutional Neural Networks (CNNs)

A CNN is a specific type of NN designed for image classification tasks. A CNN consists of multiple different chain-connected layers, each manipulating the output of the previous layer. The layers most commonly found in a CNN include the Convolutional, Pooling, Batch normalization, Activation, and Linear layers. In cases of multi-class classification, a Softmax layer can also be included to provide a probability estimate of the different output classes. In this paper, we are investigating inference, which is the process of classifying an image, given a trained CNN. Therefore, we will not be investigating training the network on the FPGA as it is not relevant to our use case. Further information on CNNs can be found elsewhere, so this serves as a brief reiteration of how the layers compute their output. Each description computes the output of a single element in the output feature map, where b is the batch index, f is the feature map index, C is the number of input channels, K and L are window dimensions, and i and j are the spatial indices. We seek to implement the following layers:

Convolutional: A convolutional layer convolves a kernel over the input image, producing a feature map. The formula for computing the output of a convolutional layer is as follows:

$$\mathsf{out}_{b,f,i,j} = \mathsf{bias}_f + \sum_{c=0}^{C-1} \sum_{k=0}^{K-1} \sum_{l=0}^{L-1} \mathsf{in}_{b,c,i+k,j+l} \cdot \mathsf{W}_{f,c,k,l} \tag{1}$$

Batch normalization: A batch normalization layer normalizes the output of the previous layer during training to achieve a mean of 0 and a standard deviation of 1. The formula for computing the output of a batch normalization layer is as follows:

$$\mathsf{out}_{b,f,i,j} = \frac{\mathsf{in}_{b,f,i,j} - \mu_f}{\sqrt{\sigma_f^2 + \epsilon}} \cdot \gamma_f + \beta_f \tag{2}$$

Activation: An activation layer applies a non-linear function to the output of the previous layer. In our network, we use the Rectified Linear Unit (ReLU) activation function, which is defined as follows:

$$\mathsf{out}_{b,f,i,j} = \max(0, \mathsf{in}_{b,f,i,j}) \tag{3}$$

Maxpooling: A maxpooling layer downsamples the output of the previous layer by taking the maximum value of a kernel. The formula for computing the output of a maxpooling layer is as follows:

$$\mathsf{out}_{b,f,i,j} = \max_{k=0}^{K-1} \max_{l=0}^{L-1} \mathsf{in}_{b,f,i\cdot S+k,j\cdot S+l} \tag{4}$$

Linear: A linear layer is a fully connected layer, where each output is connected to each input. The formula for computing the output of a linear layer is as follows:

$$\mathsf{out}_{b,f} = \mathsf{bias}_f + \sum_{c=0}^{C-1} \mathsf{in}_{b,c} \cdot \mathsf{W}_{f,c} \tag{5}$$

Softmax: A softmax layer is used for multi-class classification, where it outputs a probability estimate for each class. The formula for computing the output of a softmax layer is as follows:

$$\mathsf{out}_{b,f} = \frac{e^{\mathsf{in}_{b,f}}}{\sum_{c=0}^{C-1} e^{\mathsf{in}_{b,c}}} \tag{6}$$

4 Implementation

We aim to accelerate inference using pre-trained CNNs from PyTorch on FPGAs, employing floating-point operations and weights to follow PyTorch standards.

4.1 HLS implementation

In the HLS design, each layer is a separate function, called through a top-level entry function. This paper explores three design strategies: naive, optimized, and streaming. While further optimizations are possible, we will not be investigating them in this paper.

4.1.1 Naive implementation

The naive implementation directly translates the layer's math into C++ code with a simple nested loop over input data elements. While it is straightforward, it lacks FPGA optimization and serves as a baseline for the optimized and streaming versions.

4.1.2 Optimized implementation

The optimized implementation enhances performance and resource efficiency over the naive one using the following optimizations applied to each layer:

- `pragma loop_flatten` - merges nested loops into a single loop, reducing the cycles required to enter a nested loop.
- `pragma pipeline` - enables the FPGA to process multiple iterations of the loop in a pipeline parallel fashion. The key metric, Iteration Interval (II), measures the clock cycles between consecutive iterations. An II of 1 implies perfect pipelining with max throughput, while an II of 2 halves throughput as it processes data elements every other clock cycle.
- `pragma unroll` - unrolls the loop, which enables the compiler to replicate the loop body, allowing the FPGA to process multiple iterations of the loop in parallel.
- `pragma inline` - inlines the function to reduce the overhead from function calls as the function body is inserted directly into the calling function. This can also enable further automatic compiler optimizations as the compiler has more information about how the code directly interacts. However, this optimization can also hinder the compiler in re-using hardware, which in turn increases resource consumption.
- `pragma bind_storage` - binds a variable to a specific memory resource, such as the internal FPGA cache-like memory resource BRAM. This binding enables the compiler to optimize the memory access pattern and latencies.
- `pragma array_partition` - partitions an array into smaller arrays, which enables the compiler to parallelize memory access, and improves locality as the data containers can reside closer to the computation.
- `pragma dependence` - removes dependencies between loop iterations, removing potentially unnecessary memory management and correctness checks.
- 2-part reduction - The float32 adder in a multiply-accumulate, like in the convolutional layer, has a 4-cycle latency. As a result, the convolutional layer's loop has an II of 4, disrupting perfect pipelining. To achieve an II of 1, the first pass writes to 4 output

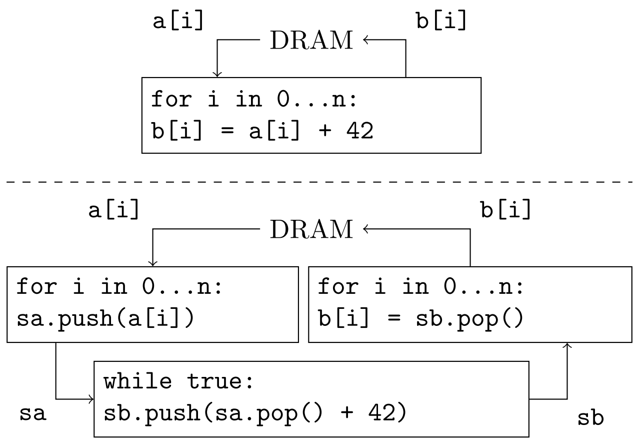

Figure 1: An example of transforming code for streaming between functions. **Top:** The original code. **Bottom:** The transformed code. Each rectangle represents a parallel function, and each arrow represents a memory access/stream.

buffers, and the second pass accumulates the buffers into a single value, reducing the II of the first pass to 1 but adding resources and latency due to the second pass.

- separate buffer initialization - as each nested loop iteration initializes its buffer, the compiler has to insert additional hardware for inter-iteration communication to ensure correctness. By initializing the buffer in a separate loop, the compiler can remove this hardware, reducing resource usage.

It should be noted that the pragma's are compiler hints; if the compiler cannot safely apply the optimization, it will not do so. For instance, while we might specify that we want to pipeline a loop with an II of 1, the compiler may not be able to do so due to dependencies between loop iterations. Hence, we must apply a combination of all of these optimizations to achieve the desired result.

4.1.3 Streaming implementation

The last strategy is the streaming implementation, combining the previous optimizations with data streaming between computational units. The FPGA is a spatial architecture, meaning individual pieces of hardware, which for our HLS implementation are functions, are replicated to increase throughput. If we write standard C++, the compiler will adhere to the semantics of a traditional C++ program, which is sequential execution. As a result, while the functions are replicated and internally pipelined, they are not necessarily executed in parallel as the compiler may not know that the functions are independent.

To circumvent this problem, instead of mimicking CPU control flow in between functions, we replicate each function and stream data in between them, moving the control flow onto the synchronization of the streaming transactions. This is a manual code transformation as the compiler cannot do this automatically, but after transformation, it can be hinted to the compiler using the `pragma dataflow`. An example of this transformation is shown in Figure 1.

To enable streaming, we avoid the naive sequential approach, where each layer assumes full input availability at the start. Instead, we buffer the minimal needed data, e.g., for convolutional and maxpooling layers, buffering $k-1$ rows and k extra entries. We process using a rolling buffer, advancing one element per iteration until we need to buffer more. For linear layers, we buffer the entire input for simplicity, as this layer has minimal impact on overall performance.

4.2 SME implementation

In the design of the SME implementation of the CNN architecture, each layer is constructed individually as its own, independent implementation. A layer is modularized as a parameterized unit, adaptable for various CNNs. All implementations of the layers conform to a common interface for inter-layer communication, covered in Section 4.2.1. Some layers have multiple implementation versions such as a parallelized implementation version that has enhanced throughput. Here the user can select which implementation version to use for a layer instance. The choice of implementation version comes with a trade-off between resource consumption and speed, but the option of choosing between different implementation versions allows for customization for specific FPGAs, as illustrated later.

4.2.1 Communication

We design communication between layer implementations using buses via our custom `ValueBus` interface. This interface contains a 32-bit floating point field (`Value`) and two boolean fields (`Enable` and `LastValue`). A bus transaction is valid (the data in `Value` can be read) when the `Enable` field is set to true. The `LastValue` field is set to true when the transaction is the last transaction on the bus for a given input, e.g. when a full image has been streamed through a layer.

We create both more and less parallel versions of the layer implementations to extend the pallet of options of implementations for a layer instance. The user should select the implementation version of a layer instance based on the specific layer instance's configuration. Parallelism in a layer implementation depends on data streaming: either the range of channels in the data is streamed in parallel (a bus of type `ValueBus[]` containing values for each channel streaming value by value, row by row), or the range of channels in the data is streamed in sequence (a bus of type `ValueBus`, where the next channel is streamed once the streaming of the former channel is done). This information is embedded into the layer implementation version's name. For instance, a convolutional layer implementation version with an input bus of type `ValueBus[]` and an output bus of type `ValueBus` is named "`ConvLayer_01`", where the two digits at the end of the name symbolize the type of the input bus and output bus respectively. If a digit is `0`, it implies that the bus is of type `ValueBus[]`, and if the digit is `1`, it implies that the bus is of type `ValueBus`.

4.2.2 Compositionality

For implementing each layer implementation, it is important to split up the larger, complex computations into a chain of basic arithmetic operations, which we define as isolated processes. The incentive behind it is that the critical path of a design is the longest critical path of all processes, and long processes with many computations result in long critical paths. Processes can be clocked or unclocked; we opt for purely clocked processes to minimize critical path length. This is because, in a chain of unclocked processes executing in sequence within a single clock cycle, the critical path becomes the sum of the critical paths of all processes in the chain. Conversely, for a chain of clocked processes, the critical path becomes the longest critical path of all processes in the chain, which is strictly smaller than the sum.

Leveraging SME's compositional capability, we form and interconnect processes in class constructs, referred to as "wrapper classes". These wrapper classes, behaving like conventional C# classes, enable us to uphold a more abstract representation of constructs in an implementation.

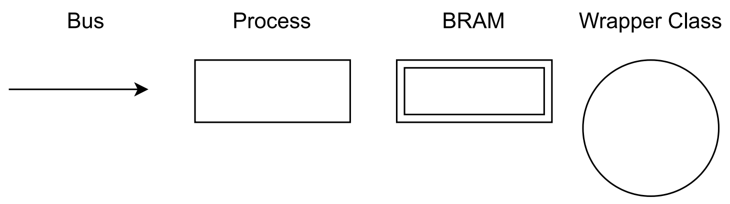

Figure 2: Definitions of the components used in the later flowchart visualizations of the SME implementations of CNN layers.

Furthermore, we can nest wrapper classes within one another and establish connections between processes contained within the same wrapper class. By following this iterative approach, we eventually attain a level of abstraction in a wrapper class that aligns with the layer we aim to implement. Communication among processes, as well as between processes and wrapper classes, is facilitated through buses utilizing the `ValueBus` interface. Similarly, interactions between wrapper classes can also be accommodated through this same interface.

4.2.3 Design

Each layer operates on a single input at a time, utilizing a batch size of one. This choice aligns with an embedded inference configuration, where sensor input data arrives in individual samples and is processed in real-time. We use flowcharts to illustrate our design concept, with Figure 2 outlining the component representation in the flowcharts. Processes are depicted as rectangular boxes with single borders, while instances of internal FPGA memory (BRAM) are represented by rectangular boxes with double borders. Arrows symbolize buses, with their direction indicating bus flow. Circles denote wrapper classes. We will walk through the design of the convolutional layer implementation versions, the `ConvLayer_00` and `ConvLayer_01` wrapper classes, to understand the structural difference between the implementation versions. The implementations of the other layers must be analyzed from the code repository.

4.2.4 ConvLayer_00

The `ConvLayer_00`, shown in Figure 3, is a wrapper class that implements a convolutional layer that receives and handles the input and output channels in parallel. Thus, the input and output buses of the wrapper class are both of type `ValueBus[]`. Each input channel is stored in a local BRAM, which an instance of the `InputCtrl_ParFilter` process has access to. `InputCtrl_ParFilter` handles the correct order of outgoing values from the input channel according to the kernel and stride values in the configurations of the layer. Two values from a 2D region of the channel stored in the BRAM are streamed at each clock tick to each `Filter` (which is described in Section 4.2.5).

4.2.5 Filter

The `Filter` class is a wrapper class that is used in `ConvLayer_00` only, and it contains the functionality for generating one output channel. The internals of a `Filter` are shown in Figure 4, where each input goes to a designated instance of the `ConvKernel_type00` wrapper class. The notation `type00` implies that the wrapper class is used only by the `ConvLayer_00` implementation version of the convolutional layer. `ConvKernel_type00` computes the sum-product between the values on the input buses, and matching weights of the kernel. An instance of the `ValueArrayCtrl` process then buffers the results in a field, which corresponds to registers on the FPGA. The field is an array of the same size as the number of kernels in the `Filter` instance (equivalent to the number of input channels in the ConvLayer_00 instance), and the ValueArrayCtrl instance outputs one value at each clock tick if the buffer is filled. When the last value of this buffer has been outputted, the LastValue field of the output bus is set to true. At every clock tick, an instance of the PlusCtrl process accumulates the incoming values, outputting and resetting the accumulated value when the LastValue field of the input bus is set to true. Finally, the Bias process adds the bias of the Filter to the accumulated value, completing a single output value of the Filter.

4.2.6 ConvKernel_type00

ConvKernel_type00, outlined in Figure 5, computes the sum-product between input values and the corresponding weights. The kernel weights are stored in a field, which is an array and corresponds to registers on the FPGA, inside an instance of the KernelCtrl process. The kernel starts when the Enable fields of the input buses are true. The inputs come from a 2D region of a designated input channel, and once the LastValue field is set to true, all values from the 2D region have passed through the KernelCtrl instance. Each value on the two input buses in KernelCtrl is matched with corresponding a weight and sent to a WeightValue process, where they are multiplied. Their product continues to PlusCtrl for accumulation in a buffer implemented as a field in the process. When PlusCtrl receives a Value with the LastValue field set to true, it outputs the buffer. Finally, the outputs of the two PlusCtrl processes are added together in the PlusTwo process, which outputs the final sum-product. It is important to note that the dual-branch design optimizes FPGA dual-port BRAM utilization.

4.2.7 ConvLayer_01

A convolutional layer can also be implemented with the wrapper class ConvLayer_01; a less parallel implementation version compared to ConvLayer_00. It receives the input channels in parallel using an input bus of type ValueBus[], but computes the output channels in sequence using an output bus of type ValueBus. While this version uses more clock ticks to process the same input, it is a viable option when ConvLayer_00 is too resource-demanding for a specific configuration. Its design is presented in Figure 6 and the core idea is to maintain a maximum of one filter that applies all the filters in sequence, thus saving resources by using fewer total processes than the ConvLayer_00 implementation version. ConvLayer_01 does not use an instance of the Filter class, because of the slight change in required functionality, but the same principles are used.

The kernel weights across filters that operate on the same input channel are stored in a single BRAM instance, which is controlled by an instance of the InputCtrl _SeqFilter process. Each InputCtrl_SeqFilter outputs one value from a 2D region of the designated input channel and a corresponding weight every clock cycle. Once all 2D regions in the input channel have been traversed, InputCtrl_SeqFilter repeats its output process, but with weights related to the next filter until all filters have been applied to the input channel. ConvKernel_type01, illustrated in Figure 7, applies the sum-product between the 2D regions of the input channel and the weights it receives. The notation type01 implies that the wrapper class is used only by the ConvLayer_01 implementation version of the convolutional layer. The sum-products of all ConvKernel_type01 instances are buffered in registers on the FPGA in a ValueArrayCtrl instance and are accumulated in a buffer inside a PlusCtrl instance. When the PlusCtrl instance receives the last sum-product, indicated by LastValue being true, it outputs its buffer. The Align process contains the bias values of the filters, and when PlusCtrl outputs the last sum-product, it outputs the corresponding bias value along with the sum-product. Emitting the correct bias is done by counting the number of transactions as the number of outputs from a

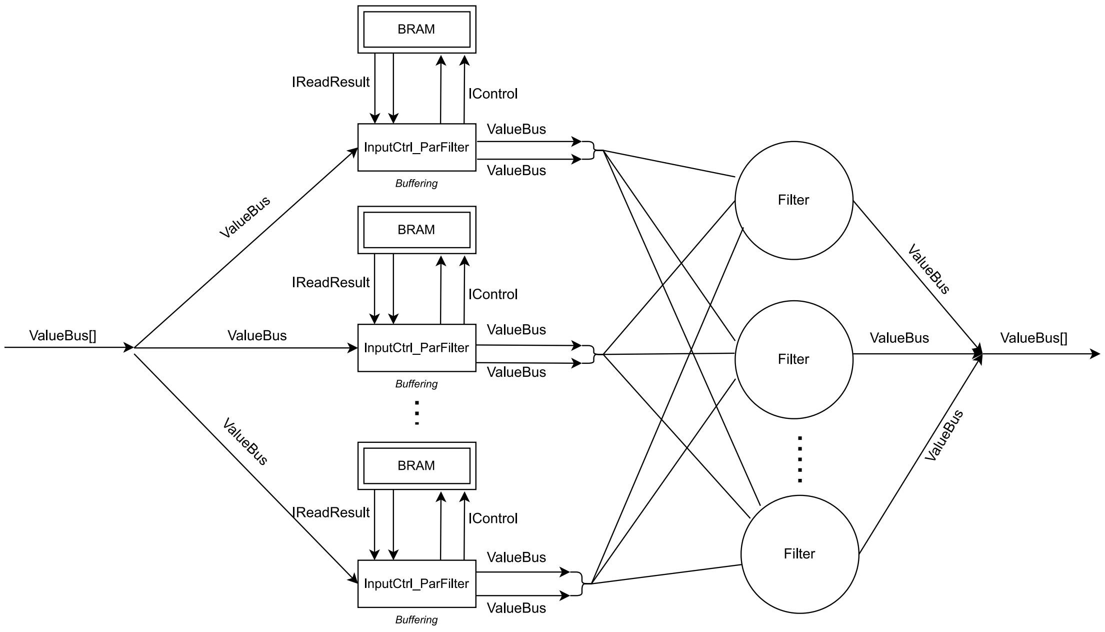

Figure 3: Design of `ConvLayer_00`. Each `InputCrtl_ParFilter` stores a whole channel, value by value at each clock cycle, in a BRAM. Afterward, two values from the BRAM are outputted each clock cycle. They both go to a designated `ConvKernel_type00` in all `Filters`.

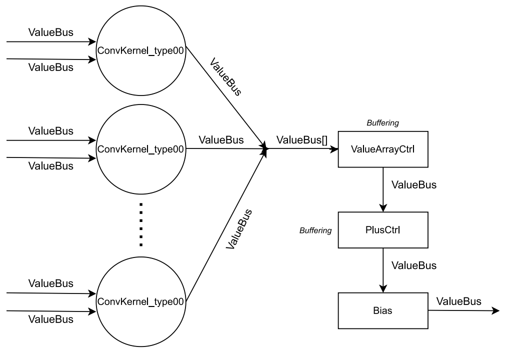

Figure 4: Design of `Filter`.

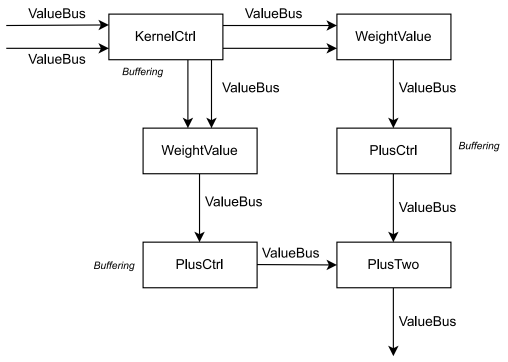

Figure 5: Design of `ConvKernel_type00`.

filter is known, because it is the total size of a single output channel in the specific convolutional layer instance configuration.

4.2.8 Clock Cycles of layers

The equations for calculating the latency in clock cycles that a layer implementation version spends processing are defined by the equations presented in Table 1, derived from source code analysis. The equations assume that the input is consistently streamed each clock cycle until the layer has finished. In each row, the "Layer" column specifies which layer type is implemented, the "Version" column specifies the version of the layer implementation, and the "Clock Cycles Equation" column specifies how to calculate the latency in clock cycles that the layer

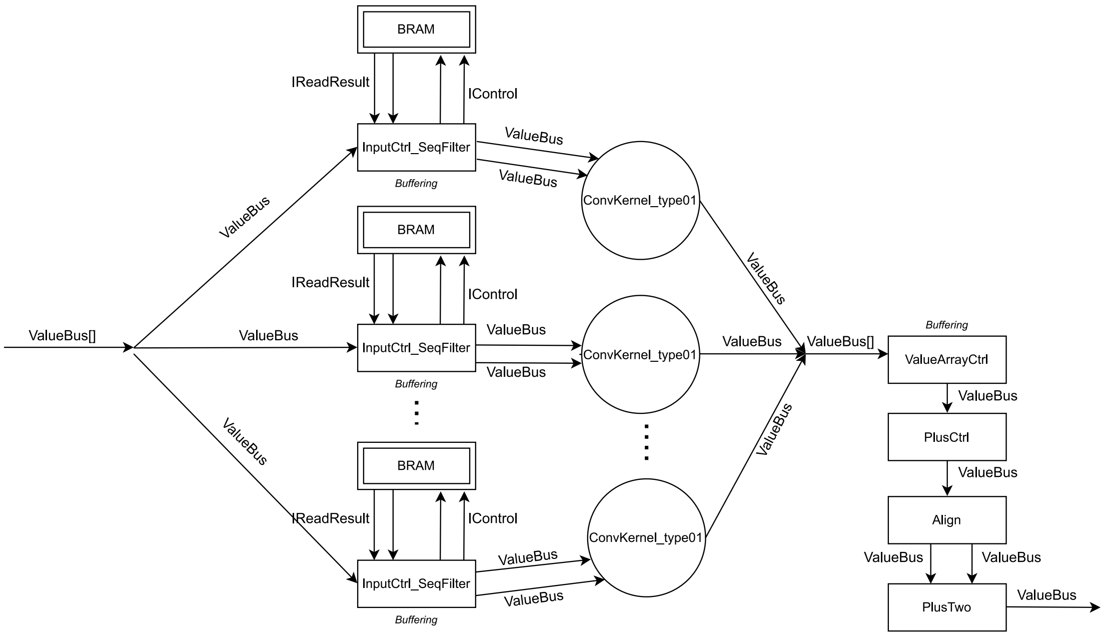

Figure 6: Design of `ConvLayer_01`.

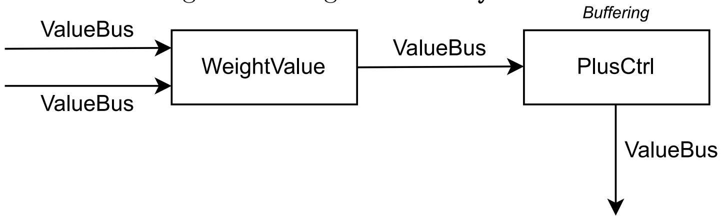

Figure 7: Design of `ConvKernel_type01`.

implementation version uses to process one input. Here, the variables can be derived from the configurations, the layer implementation version uses for generating a layer instance. As an example, the row with a value of "Convolutional" in the "Layer" column and a value of "00" in the "Version" column implies the `ConvLayer_00` implementation version. To validate the accuracy of the equations, we count clock ticks when testing our benchmark CNN architecture using simulation processes, and compare this to the result of an equation. We see that for all layers the number of recorded clock cycles match the equations.

4.2.9 Integer width optimization

When applying our layer implementations to a specific CNN architecture, we manually convert integer variables within processes to a predefined number of bits. This alignment is based on their known maximum values, derived from the layer configurations. These conversions are specifically applied to indexing variables within the control processes, which are used for array indexing and traversal. We can statically determine their maximum values, and they never require the full 32-bit integer range. The number of bits in these integer types can be defined using SME integer types.

To determine the bit widths, we compute the numerical range for all indexing variables in a

Table 1: Latency in clock cycles of each layer implementation to process one entire input.

Layer	**Version**	**Clock Cycles Equation**
Convolutional	00	$h_{in} \cdot w_{in} + 2 + \left\lceil \frac{k_h \cdot k_w}{2} \right\rceil \cdot h_{out} \cdot w_{out} + c_{out} + 5$
	01	$h_{in} \cdot w_{in} + 2 + k_h \cdot k_w \cdot h_{out} \cdot w_{out} \cdot c_{out} + 3$
Batchnormalization	00	$h_{in} \cdot w_{in} + 4$
	11	$h_{in} \cdot w_{in} \cdot c_{out} + 8$
ReLU	00	$h_{in} \cdot w_{in} + 1$
	11	$h_{in} \cdot w_{in} \cdot c_{out} + 1$
Maxpooling	00	$h_{in} \cdot w_{in} + 2 + \left\lceil \frac{k_h \cdot k_w}{2} \right\rceil \cdot h_{out} \cdot w_{out}$
	11	$h_{in} \cdot w_{in} \cdot c_{in} + 2 + \left\lceil \frac{k_h \cdot k_w}{2} \right\rceil \cdot h_{out} \cdot w_{out} \cdot c_{out}$
Linear	00	$h_{in} \cdot w_{in} \cdot (c_{in} - 1) + 2 + 3$
	10	$h_{in} \cdot w_{in} \cdot c_{in} + 2 + 3$
Softmax	00	$1 + (c_{in} - 1) + 1 + 1$

The table presents the equations for calculating the latency in clock cycles used by all available SME implementation versions of all layers, each consistently processing a single input stream. Each row in the table provides information on the layer type (found in the ”Layer” column), the implementation version (identified by the input and output bus type notation in the ”Version” column), and the equation used to calculate the latency in clock cycles required to process a single input. h and w are the height and the width of a channel, and c is the number of channels. The subscripts *in* and *out* determine whether a variable belongs to an input or output channel. k_h and k_w are the kernel height and kernel width.

control process. The range is derived from the configurations of all layer instances in the CNN architecture, which are implemented with layer implementation versions that use said control process. Subsequently, we select the maximum range for each variable across all instances and use that as the bit count for the integer type employed for that variable. It is worth noting that although this process is currently manual, prior work has demonstrated automation of this capability in SME [4]. However, this automation has not been implemented in our current SME version.

5 Verification

The main objective of this paper is to explore the possibility of using SME as an implementation platform for embedding CNNs onto FPGA. Thus, we will need to verify that it produces the results intended for a given CNN architecture. The HLS implementation is verified in the same way.

We validate our layer implementations by comparing them to equivalent PyTorch implementations of the same layers. Each implementation is tested with both random input data and MNIST dataset samples [15]. The CNN (CNN_small), shown in Figure 8, is used to for-

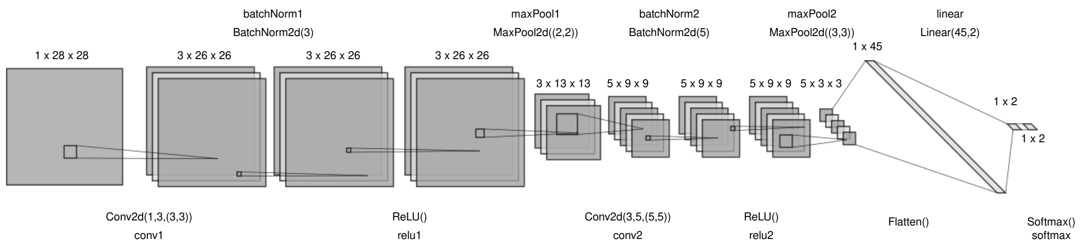

Figure 8: Design of CNN_small.

The figure provides a visual representation of the CNN_small architecture and the transformations applied to the incoming data by the layers throughout the entire CNN. The names of each layer, along with the specific methods and configurations employed in PyTorch using the `torch.nn` module, are written between the gaps of the transformations applied to the input. The dimensions of the input at various stages during the CNN's manipulation process are displayed either above or below the data representation.

mally test our method for a CNN embedding onto an FPGA. CNN_small is trained in PyTorch to identify a ”one” or ”two” in MNIST images. The labels have been one-hot encoded ([1,0] for ”one” and [0,1] for ”two”). The layers, which CNN_small consists of, are named **`conv1`**, **`batchNorm1`**, **`relu1`**, **`maxPool1`**, **`conv2`**, **`batchNorm2`**, **`relu2`**, **`maxPool2`**, **`linear`**, **`softmax`**, and their representations and manipulations on the input data can be seen in Figure 8. The layer implementation versions, which we use for each layer instance, are based on ensuring that resource consumption does not exceed what is available on our target FPGA and highlighted in Section 6.2.

We provide the following metrics for our verification: the mean of the error, the variance of the error, the maximum observed error, and the Relative Root Mean Squared Error (RRMSE). The RRMSE is defined in Equation (7), where $\hat{y}_i$ is the estimated value, y_i is the true value, and n is the number of values.

$$\text{RMMSE} = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}}{\sum_{i=1}^{n} y_i^2} \tag{7}$$

The default datatype used by PyTorch is 32-bit floating point (float32), which follows the IEEE-754 standard [9]. Float32 designates 23 bits to capture the significant digits of a number, enabling it to accurately represent the initial 6 digits of a number, introducing representation discrepancies from the 7th digit onward. However, multiple arithmetic operations can accumulate this error, increasing it beyond the 7th digit. Furthermore, float32 is non-associative, which means the order of operations potentially introduces additional discrepancies. Therefore, we consider arithmetic equivalence even with an error on the 6th digit, which is verified by RRMSE values of 1e-6 or lower. For both the SME and the HLS implementations, we do not see an error above this threshold, indicating that they are correct compared to the PyTorch implementation.

6 Benchmarking

We benchmark the FPGA implementations of CNN _small against PyTorch on a laptop CPU and GPU, and Pynq-Z2's CPU through ONNX [13]. Laptop specifications are in Table 2,

Table 2: Specifications of the laptop and Pynq-Z2 CPU used for benchmarking.

	Laptop [12]	Pynq-Z2 [19]
CPU model	Intel i7-7700HQ	ARM Cortex-A9
CPU speed	4 cores @ 3.8 GHz	2 cores @ 650 MHz
CPU TDP	45.000 W	1.256 W
RAM size	32 GB DDR4	512 MB DDR3
GPU	NVIDIA GeForce 940MX	-
GPU TDP	23.000 W	-
OS	Arch Linux 6.2.11	PYNQ 2.7.0
Python	3.10.10	3.8.2
PyTorch	2.0.1	-
ONNX runtime	-	1.9.1

Table 3: Specifications of the FPGA on the Pynq-Z2.

Name	LUT	FF	BRAM	DSP
XC7Z020	53200	106400	140	220

with Pynq-Z2 specs in Table 3. Our laptop is from 2017, with a 7th-gen Intel CPU and a 9th-gen NVIDIA GPU, chosen to align closer with Pynq-Z2's age (2012) and its low-power profile, ensuring a fairer comparison. Newer generation hardware will improve all of our results for all platforms, but the relative performance should still show the same trends. It is also worth noting that the laptop, while mobile, has a much higher power budget than the Pynq-Z2, which is reflected in the power consumption results. Furthermore, the two platforms are in different price ranges at their release, with the laptop costing around 3000 USD and the Pynq-Z2 costing around 120 USD.

6.1 HLS

We have three implementation versions with HLS for CNN_architectures; the Naive, Optimized, and Streaming implementations mentioned in Section 4.1. To emphasize the need for increasing optimizations, we gauge HLS implementation performance and resource usage with the Vivado HLS tool [2]. It simulates the design and analyzes generated HDL code, with estimates detailed in Table 4. The naive approach exhausts DSP resources, rendering it infeasible to implement. The optimized version sacrifices resources for performance, resulting in higher latency but can fit the board. The streaming version slightly increases resource usage over the optimized version but reduces latency almost threefold, making it the optimal choice for our application.

6.2 SME

The choices of layer implementation versions of the layer instances in CNN_small are chosen such that they retain high throughput while fitting onto the FPGA. We optimize all our layer implementation versions to the specific use case of CNN_small without changing the core structure of our layer implementation versions in the following manner.

In Table 5, we show the resource consumption and clock cycle counts for each CNN_small layer instance using the implementation versions. Clock cycle counts are determined using the

Table 4: Resource consumption of FPGA implementations of CNN_small using SME and HLS implementations. SME and HLS-Stream are from the placed and routed results, while HLS-Naive and HLS-Opt are from the HLS tool.

	SME	**HLS-Naive**	**HLS-Opt**	**HLS-Stream**
LUT	19,109	44,174	13,886	8,873
Flip Flops	27,472	29,624	8,164	10,935
BRAM	10	44	21	11
DSP	100	246	32	60
Dynamic (W)	0.335	0.552	0.163	0.216
Static (W)	0.145	0.145	0.140	0.141
Cycles	14,908	169,113	202,663	82,410
Period (ns)	20	10	10	10
μs/Sample	298.16	1,691.13	2,026.63	824.10
mJ/Sample	0.1431	1.1787	0.6140	0.2950

equations in Table 1. We synthesize the generated VHDL code for each layer in Vivado [3], which gives us the resource estimates. From Table 5 we derive that the implementation of CNN_small, where we choose the layer implementation versions indicated by regular font and font that is underlined (parallel implementation versions), is not able to fit onto the FPGA due to excessive resource utilization. Thus, we reach a point of diminishing returns, where the cost of using the parallel versions outweighs the benefit of the fewer clock cycles spent operating on the data using parallel versions. Instead, we opt for the layer implementation versions indicated by regular font and **bold** font, which fits onto the FPGA. This demonstrates the importance of having multiple versions with different degrees of parallelism of the implementations of the layers.

As our implementation is pipelined, each layer can start processing as soon as it receives the first value of the input. This is visualized in Figure 9, where we see that CNN_small uses a total of 14908 clock cycles to process one entire input. Note that the second half of the network uses more clock cycles than the first half, highlighting the change in the degree of parallelism.

6.3 Comparison to PyTorch

The FPGA implementations (implemented with SME and HLS), intended for embedded systems near the data source, are tested with a batch size of 1. In contrast, for the CPU and GPU implementations, we test a range of batch sizes from 1 to 5000 to demonstrate how performance scales with batch size. The results in Figure 10 are the mean values gathered using 10 warmups followed by 1000 iterations for the laptop, and 100 iterations for the Pynq Z2 (due to its slower speeds), of predicting two consecutive batches. Since PyTorch is not directly supported on the Pynq Z2, we are instead running the model through ONNX, which we denote as ARM.

In the runtime results, SME matches CPU and GPU performance at batch size 1, while HLS lags behind by a factor of 2.76. SME outperforms ARM by a factor of 3.18, and HLS by 1.15. This quickly diverges as the batch size increases, where the CPU and GPU drastically reduce per-sample time, reaching 34.35 times faster than SME at batch size 5000. ARM surpasses

Table 5: Layer implementation options for layers in CNN_small.

Configuration	Version	Cycles	LUT	BRAM	DSP	Flip Flops	Dynamic power (W)	Frequency (MHz)
conv1	00	417	6520 (12.3%)	1 (0.7%)	36 (16.4%)	8949 (8.4%)	0.17	123
batchNorm1	00	680	1857 (3.5%)	0	24 (10.9%)	3062 (2.9%)	0.13	299
relu1	00	677	50 (0.1%)	0	0	98 (0.1%)	0.02	913
maxPool1	00	1016	1029 (1.9%)	3 (2.1%)	0	809 (0.8%)	0.11	97
conv2	00	1234	23300 (43.8%)	3 (2.1%)	140 (63.6%)	31533 (29.6%)	0.64	83
	01	10302	4207 (7.9%)	3 (2.1%)	16 (7.3%)	5308 (5.0%)	0.12	57
batchNorm2	00	85	3093 (5.8%)	0	40 (18.2%)	5102 (4.8%)	0.23	299
	11	413	970 (1.8%)	0	8 (3.6%)	1278 (1.2%)	0.06	59
relu2	00	82	82 (0.2%)	0	0	162 (0.2%)	0.03	913
	11	406	18 (∼0%)	0	0	34 (∼0%)	0.01	913
maxPool2	00	128	2039 (3.8%)	5 (3.6%)	0	1367 (1.3%)	0.22	83
	11	632	358 (0.7%)	1 (0.7%)	0	274 (0.3%)	0.04	116
linear	00	41	13312 (25.0%)	2 (1.4%)	12 (5.5%)	6008 (5.7%)	0.29	125
	10	50	2424 (4.6%)	2 (1.4%)	12 (5.5%)	3286 (3.1%)	0.09	179
softmax	00	7	4103 (7.7%)	0	4 (1.8%)	5180 (4.9%)	0.18	255

This table displays resource utilization, clock cycles, dynamic power, and frequency for various CNN_small layer implementation versions when synthesized in isolation. The layer type is found in the "Layer" column, and the implementation version is found in the "Version" column, identified by the input and output bus type notation. For some layer instances, one has to choose between two options for the implementation version. To meet bus type requirements, one can only choose one of the two paths highlighted in the "Version" column using **bold** (indicating a fully parallel path) and underlined (indicating a sequenced path) text fonts.

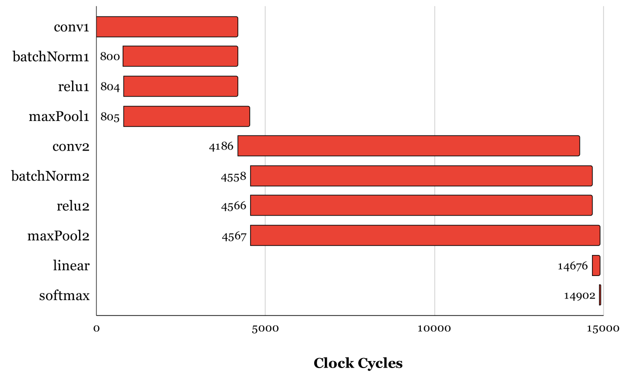

Figure 9: Gantt chart showing how each layer in the SME implementation of CNN_small is scheduled, highlighting the pipeline parallel execution.

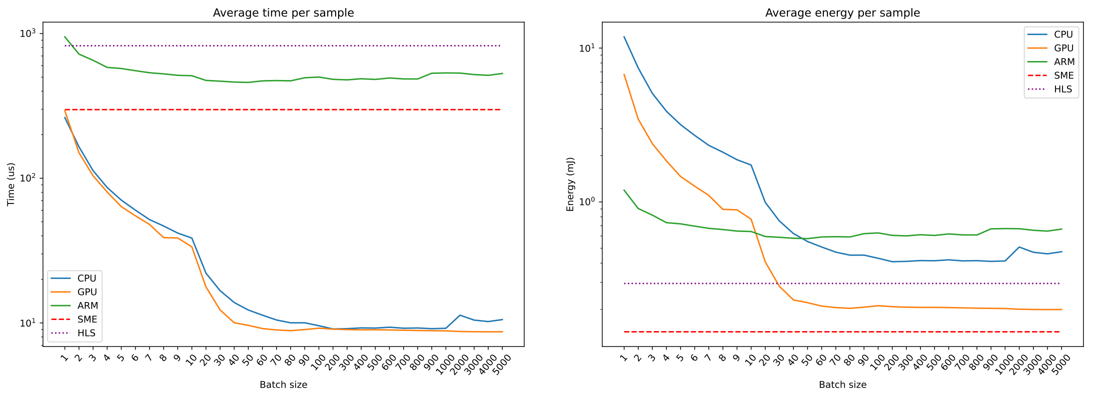

Figure 10: (**Left**:) Runtime and (**Right:**) energy consumption of CNN_small on different platforms. Note that the y-axis is logarithmic, and the x-axis is uneven. ARM is the processor on the Pynq Z2.

HLS by 1.79 times but lags behind SME by 1.54 times.

This may seem suboptimal at first sight, but note that the FPGA board uses an order of magnitude less power than the CPU and GPU. If we instead look at the energy consumption, we see that the FPGA implementations are more energy-efficient than the CPU and GPU implementations. As we do not have proper power measurement equipment, we gauge energy consumption by comparing CPU/GPU TDP to FPGA dynamic power plus static power, which is comparable to TDP. This is a rough estimate, but it is sufficient to demonstrate the energy efficiency of the FPGA implementations. The HLS implementation is the least energy-efficient of the two, beating only the CPU implementation by a factor of 1.38 while being beaten by the GPU by a factor of 1.47. The SME implementation is the most energy-efficient, beating the CPU and GPU implementations at their best batch size by a factor of 2.85 and 1.39, respectively. Looking at the ARM processor, even though it consumes less power than the CPU/GPU, it is still not able to beat SME and HLS, which is more efficient by a factor of 4.03 and 1.95.

While the FPGA may seem suboptimal initially, it consumes significantly less power than the CPU and GPU. When considering energy efficiency, FPGA implementations outperform CPU and GPU ones. We estimate energy consumption by comparing CPU/GPU TDP to FPGA dynamic plus static power, which is comparable to TDP. HLS is the least efficient, beating the CPU by 1.38 times and being outdone by the GPU by 1.47 times. SME is the most efficient, surpassing CPU and GPU at their best batch size by 2.85 times and 1.39 times, respectively. Even though the ARM processor consumes less power than CPU/GPU, it cannot outperform SME, which is 4.03 times more efficient.

However, since we are targeting a batch size of 1, the results favor the FPGA implementations. SME is the most energy-efficient, surpassing CPU, GPU, and ARM by factors of 82.61, 47.03, and 8.32, respectively. HLS ranks second, outperforming CPU, GPU, and ARM by factors of 40.07, 22.81, and 4.03.

Considering the platforms, their power consumption, their price, and their age differences, we see that the FPGA implementations are a viable alternative to the CPU, GPU, and even ARM implementations. The FPGA implementations are slower, but they are more energy-efficient, which is a crucial factor for embedded systems.

7 Future Work

We have shown that embedding CNN architectures in FPGAs is both feasible and beneficial, but there is still room for improvement.

7.1 SME Process optimization

The processes inside the SME layer implementations that use BRAMs to store channels currently wait for entire channels to be stored before outputting values. This is to ensure that data, which will be used multiple times can be accessed again. However for some configurations, it is not necessary to store the entire channel for the duration of the process working, and thus clock cycles could potentially be spared.

We have not dived into the specific order of statements within a process, and we could potentially optimize the order of statements to reduce the length of the critical path, in turn increasing the clock rate. For instance, ensuring that writes are isolated in branches and are not followed by reads prevents complex logic from guarding the path to ensure sequential semantics.

7.2 SME resource sharing

In our current solution, we utilize the compositionality of SME to express layers as classes, which after initialization become interconnected. This, however, prevents resource sharing among layers, leading to idle processes and inefficient resource utilization. By employing resource sharing, it could become possible to implement even larger CNN architectures given an effective strategy, albeit at the cost of performance, especially for sample pipelining.

7.3 Sample pipelining

Currently, we pipeline the processing of a single sample, but we could also pipeline the processing of multiple samples. This would provide further speedup at higher batch sizes, increasing the gap between the CPU/GPU and the FPGA implementations, shown in Figure 10. Given the existing internal pipelining, this change should be straightforward.

7.4 Proper power measurements

We have used TDP as the metric for power consumption, but it is not precise. Instead, using a power measurement tool for accurate data would enhance energy consumption comparisons among platforms, potentially bringing new insights to the results.

7.5 Related work

Numerous frameworks utilize FPGAs for AI applications, with one being hls4ml [7], a Python package using Vivado HLS [2]. It generates FPGA-customized VHDL code for various NN layers. While hls4ml simplifies FPGA-based AI deployment, optimizing the VHDL code for peak efficiency might demand manual adjustments due to specific FPGA architectures. Nevertheless, hls4ml acts as a valuable gateway between neural network frameworks and FPGA acceleration.

DaCeML [16] is another framework, which can load AI models from PyTorch [14] and ONNX [13] into the Data-Centric parallel programming framework (DaCe) [6], enabling support for various device targets, including FPGA acceleration via HLS kernels. DaCe has the advantage gives the capability to optimize the generated code through data-centric graph transformations, tailoring it for target-specific enhancements like memory access patterns and data reuse, all while preserving program semantics. While DaCeML generates FPGA kernels, its original focus on GPU kernel generation necessitates users to manually leverage FPGA-specific optimizations to achieve peak performance on FPGAs.

Finally, there is the Vitis AI framework [1], which is a framework that enables the user to generate AI processors called Deep learning Processing Units (DPUs) that run AI applications on FPGAs. These DPUs follow the same design strategy as many other AI accelerators, namely that they employ an instruction set architecture hardware platform that is programmable through software. While this provides a flexible solution, the generality of the framework hinders potential optimizations for a specific application, rendering it unable to fully exploit the spatial architecture of FPGAs.

All of these framework posts interesting solutions to the same problem, making it an interesting comparison to the work in this paper.

8 Conclusion

We have demonstrated FPGA-based CNN integration using SME and HLS, showcasing on-par performance compared to PyTorch CPU/GPU/ARM for batch size 1, and improved energy efficiency across all batch sizes. For our CNN architecture use case, the SME implementation on FPGA equals CPU/GPU speed for batch size 1 while being ∼83x, ∼47x, and ∼8.32x more energy-efficient. The HLS implementation on FPGA, although slower, offers ∼40x, ∼23x, and ∼4x better energy efficiency than CPU/GPU. For larger batch sizes, CPU/GPU is ∼35x faster than the SME implementation on FPGA, but the SME implementation on FPGA beats the ARM by ∼1.5x while remaining ∼2.8x, ∼1.4x and ∼4x more energy-efficient than CPU, GPU, and ARM. The HLS implementation on FPGA is ∼1.4x and ∼2x more energy-efficient than CPU and ARM, but ∼1.5x less than GPU.

These results underscore the promising potential of FPGA-based implementations, especially for embedded systems with limited power budgets, even when compared to more expensive and newer CPU/GPU platforms, as shown in this paper. Furthermore, we have shown that SME is a viable alternative to HLS for FPGA-based applications, offering a more intuitive and flexible development experience, while still providing competitive performance and energy efficiency.

References

[1] AMD. Vitis ai. https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html. [Online; accessed August 2023].

[2] AMD. Vitis high-level synthesis. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html. [Online; accessed August 2023].

[3] AMD. Vivado. https://www.xilinx.com/products/design-tools/vivado.html. [Online; accessed August 2023].

[4] Truls Asheim. Smeil: A domain-specific language for synchronous message exchange networks. 2019.

[5] Truls Asheim, Kenneth Skovhede, and Brian Vinter. Vhdl generation from python synchronous message exchange networks. In *Communicating Process Architectures 2015 & 2016*, pages 479–500. IOS Press, 2018.

[6] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, 2019.

[7] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, Luca P Carloni, Giuseppe Di Guglielmo, Philip Harris, Jeffrey Krupa, et al. hls4ml: An open-

source codesign workflow to empower scientific low-power machine learning devices. *arXiv preprint arXiv:2103.05579*, 2021.

[8] Github. Fpga-implementation. `https://github.com/EmilStevnsborg/FPGA-implementation`. [Online; accessed September 2023].

[9] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.

[10] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[11] Carl-Johannes Johnsen, Alberte Thegler, Kenneth Skovhede, and Brian Vinter. Sme: A high productivity fpga tool for software programmers. *arXiv preprint arXiv:2104.09768*, 2021.

[12] Lenovo. Lenovo thinkpad t470p. `https://www.lenovo.com/us/en/laptops/thinkpad/thinkpad-t-series/ThinkPad-T470p/p/22TP2TT470P`. [Online; accessed August 2023].

[13] ONNX. Onnx: an open ecosystem for interoperable ai models. `https://github.com/onnx/onnx`. [Online; accessed August 2023].

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[15] PyTorch. Mnist handwritten digit recognition in pytorch. `https://pytorch.org/vision/main/generated/torchvision.datasets.MNIST.html`. [Online; accessed August 2023].

[16] Oliver Rausch, Tal Ben-Nun, Nikoli Dryden, Andrei Ivanov, Shigang Li, and Torsten Hoefler. DaCeML: A data-centric optimization framework for machine learning. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, 2022.

[17] Kenneth Skovhede and Brian Vinter. Building hardware from c# models. In *FSP 2016; Third International Workshop on FPGAs for Software Programmers*, pages 1–9. VDE, 2016.

[18] Kenneth Skovhede and Brian Vinter. C++ support for better hardware/software co-design in c# with sme. In *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*, pages 1–8. VDE, 2017.

[19] TUL. Tul pynq-z2. `https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html`. [Online; accessed August 2023].

[20] Brian Vinter and Mads Ohm Larsen. Teaching concurrency: 10 years of programming projects at ucph. In *Communicating Process Architectures*, 2017.

[21] Brian Vinter and Kenneth Skovhede. Bus centric synchronous message exchange for hardware designs.

[22] Brian Vinter, Kenneth Skovhede, et al. Synchronous message exchange for hardware designs. *Communicating Process Architectures*, pages 201–212, 2014.