Acceleration of a Deep Neural Network for the Compact Muon Solenoid

Tarik Ourida[1,*], *Wayne* Luk[1,**], *Alex* Tapper[1,***], *Marco* Barbone[1,****], and *Robert* Bainbridge[1,†]

[1]Imperial College London, South Kensington, London, United Kingdom

Abstract. There are ongoing efforts to investigate theories that aim to explain the current shortcomings of the Standard Model of particle physics. One such effort is the Long-Lived Particle Jet Tagging Algorithm, based on a DNN (Deep Neural Network), which is used to search for exotic new particles. This paper describes two novel optimisations in the design of this DNN, suitable for implementation on an FPGA-based accelerator. The first involves the adoption of cyclic random access memories and the reuse of multiply-accumulate operations. The second involves storing matrices distributed over many RAM memories with elements grouped by index. An evaluation of the proposed methods and hardware architectures is also included. The proposed optimisations can yield performance enhancements by more than an order of magnitude compared to software implementations. The innovations can also lead to smaller FPGA footprints and accordingly reduce power consumption, allowing for instance duplication of compute units to achieve increases in effective throughput.

1 Introduction

The CMS (Compact Muon Solenoid) at the Large Hadron Collider (LHC) at CERN makes use of a LLP (Long-Lived Particle) Jet Tagging Algorithm [1], inspired by the DeepJet Tagging Algorithm [2], to search for new physics by tagging hadronic jets which stem from exotic long-lived particles. The LLP Jet Tagging Algorithm (Figure 1) is a multi-class classifying DNN, passing 638 input features per jet through 12 banks of convolutional layers, followed by 3 dense layers, with interleaved activation functions and dropout layers. The model classifies jets of particles into one of four target classes: LLP jet, heavy-flavour quark jet, light-flavour quark jet and gluon jet. Its architecture is highly computationally intensive, not meeting real-time latency constraints for data selection systems when executed on a CPU; while a hardware accelerated FPGA implementation would meet real-time requirements. This work presents an FPGA acceleration of the LLP Jet Tagging Algorithm, which uses kernelisation to divide the algorithm into self-contained processing units, allowing simple orchestration of dataflow within the network and reuse of multiplication units.

This paper highlights the following two contributions:

*e-mail: tarik.ourida17@imperial.ac.uk
**e-mail: w.luk@imperial.ac.uk
***e-mail: a.tapper@imperial.ac.uk
****e-mail: marco.barbone19@imperial.ac.uk
†e-mail: r.bainbridge96@imperial.ac.uk

- Employing cyclic random access memories and reused multiply-accumulate operations as opposed to fully parallelised convolution implementations.
- Storing matrices distributed over many RAM memories, grouping elements by index, as opposed to standard methods of storing matrices whereby all data from a single matrix is stored contiguously in memory.

The above optimisations do not affect the accuracy of the LLP Jet Algorithm.

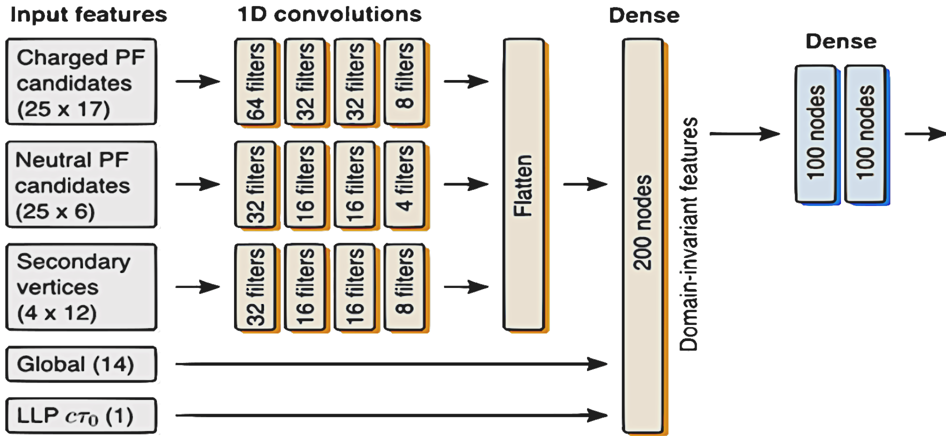

Figure 1. LLP Jet Tagging Algorithm Forward Inference

2 Background

The planned High-Luminosity LHC upgrade [3] will cause a projected upscale in particle collisions at the LHC, and therefore in data collection and processing requirements. The LLP Jet Tagging Algorithm's inputs are properties of a number of particles observed in the aftermath of proton-proton collisions, sourced directly from the CMS detector. With protons colliding at 40 MHz, there are petabytes of data to process per second, which will increase tenfold with the scheduled High Luminosity LHC upgrade.

This very high throughput criterion requires the implementation of matrix multipliers/convolvers heavily optimised for high throughput. This is required in order to establish an LLP Jet Tagging Algorithm design suitable for deployment on the real time Level-1 trigger [4], which requires a forward inference latency in the sub-microsecond range.

3 Serialised Cyclic RAM Multiply-Accumulate

The Serialised Cyclic RAM Multiply-Accumulate approach heavily reduces resource utilisation by use of floating point accumulators and cyclic RAM blocks; in some cases, a reduced latency is also achievable. The resource utilisation saving can be invested in instance duplication as to scale up throughput. A cyclic RAM block can be envisioned as a fixed size RAM whereby elements are read consecutively, cycling back to the start once the end is reached.

The architecture of the forward inference is depicted in Figure 1. The computationally intensive sections of the model are the convolutional and dense layers. The method proposed

in this section can be employed for both convolutional and dense layers as they can both be defined in terms of matrix multiplications; for brevity, the method will be demonstrated for the convolutional layers only, but is trivially transferable.

Each convolutional kernel accepts two matrix inputs (input data and weight data) which are to be convolved to produce an output matrix. A standard implementation is to instantiate a bank of multipliers, followed by an adder tree, to compute one output value from one input matrix row and one weight matrix row. These compute units can be instantiated multiple times such that the total convolution can be performed in parallel, adhering to the "compute in space" paradigm associated with FPGA development. The most unrolled implementation that exploits the maximum amount of parallelism does not achieve the lowest latency among the architectures considered due to the complex interconnects that cause higher propagation delays, thus limiting the maximum frequency in which the design can operate at, and thus its effective throughput.

By parallelising over either the input matrix, or weight matrix, but not both, as to generate one output row as opposed to the entire output matrix, will yield a lower resource utilisation due to the duplication of resources over only one output dimension, and will reduce the complexity of the interconnects. Despite this complexity reduction, designs of this nature are still taxing on the FPGAs resources, especially DSPs (Digital Signal Processors).

To circumvent the inherent issue of inputs fully connected to multiple compute units, the input interface was reduced to a single word for the inputs and weights. By employing a cyclic RAM element (RAM with read address port connected to a repeating counter) for the input and weight data, the complexity in presenting contiguous memory locations is reduced. This data configuration is coupled with a floating point accumulator, preceded by a two input multiplier, which can encompassingly be rendered as a multiply-accumulate (MAC) operation. In the case of the LLP Jet Tagging Algorithm, the weights remain constant and are stored in a constant valued cyclic RAM element (analogous to a ROM), negating the need for an input weight data port. A smaller cyclic RAM is used to store only a single row of input data, which is repeatedly iterated over until the weight RAM outputs its final value, after which the process repeats with a new input row flushed to the input RAM (Figure 2).

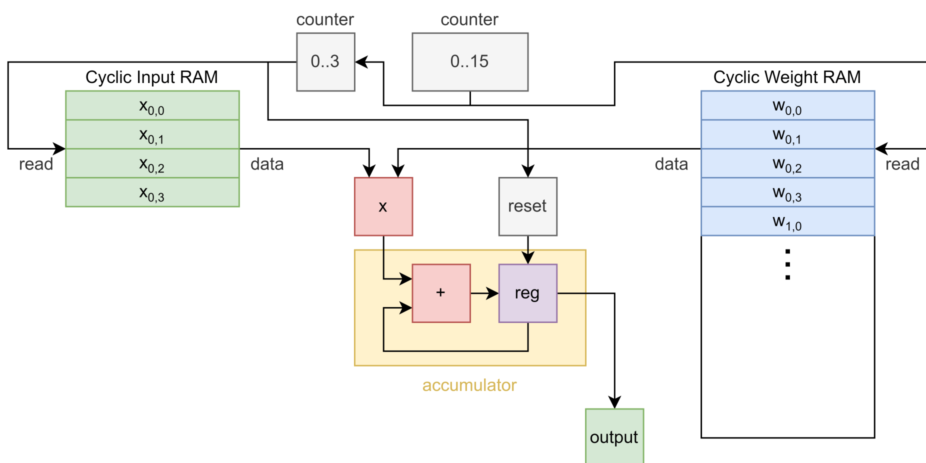

Figure 2. Input and Weight Cyclic RAMs Chained to Dual Input MAC Unit

Due to the reduction in computational parallelism, the latency for producing the output matrix data is increased, however, this design can be clocked at a higher frequency due to the elimination of high fan-in/fan-out nets, thus heavily simplifying dataflow within the design.

The convolution between two matrices of dimension 16x4, requires 4 multiplications for each of the unique pairings of rows between the two matrices, thus resulting in 16x16x4 = 1024 multiplications. Due to the transition from 1024 multipliers in the fully unrolled implementation to just 1, the resource utilisation is heavily reduced (Section 5.1). This presents the opportunity to employ multiple MAC instances to decrease the latency. The single MAC implementation accepts an input row (element by element to decrease the input interface width), and convolves it with each weight row, in order to produce an output row. This process is repeated for all input rows, with there being no inter-dependency between any computation between two different input rows. Therefore, this design can be parallelised by instantiating a MAC unit for each row of the input, for each of the three input types, with the implementation of each MAC being identical to one another, allowing for shared control logic (Figure 3). Note that despite requiring the input matrix to be ingested one column at a time, the design is still far from bandwidth limited.

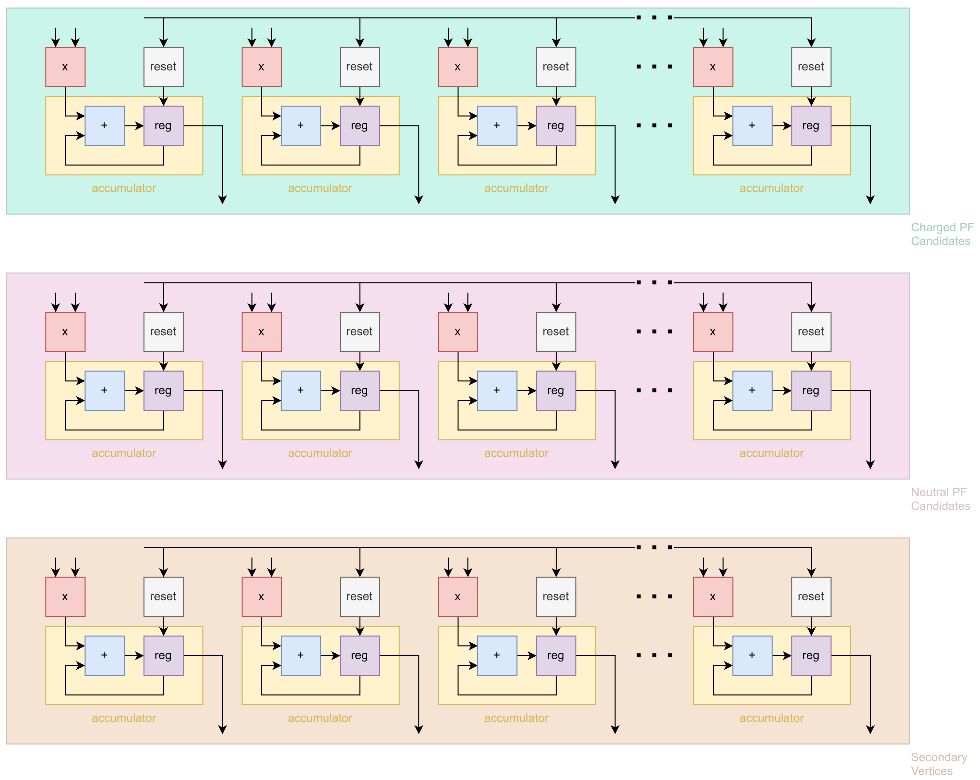

Figure 3. MAC Units Parallelised Across Input Rows and Input Channels

The approach of using MAC units offers a far simpler implementation and correspondingly reduced latency (when clocked at a higher frequency), indicating the advantage of inputting rows in a serial fashion and using an accumulators, as opposed to inputting a row at once, and using a bank of multipliers, followed by a tree of adders. For convolutions

whereby the common dimension is very large, a serialised accumulate method may not offer a latency reduction as compared to a parallel bank of multipliers, followed by an adder tree. This is because the depth of the adder tree scales logarithmically with the common dimension size, whereas the number of cycles for which the accumulator must run scales linearly with the common dimension size. In the case of the block sizes used in this implementation, the common dimension is small, thus the MAC implementation will achieve a higher throughput.

Another advantage offered by the MAC unit is that bias addition can be incorporated into the unit for no additional latency cost. An accumulator, when reset, defaults to storing zero in its accumulated value, but can instead be set to the corresponding bias associated with the input row index, prefetched from a BRAM block holding bias values. The bias addition is usually deferred to another kernel, but by integrating the bias addition into the convolution kernel, this reduces the number of kernels in the dataflow design and the additional inter-kernel hardware required for the orchestration of data between kernels.

4 Elementwise RAM Storage

The purpose of these optimisations are to simplify read logic by opting for a alternative storage approach, yielding a reduced latency.

The primary data structure that propagates through a DNN is the matrix. Conceptually, it is envisioned as a two dimensional grid of numbers, and it is common to implement their underlying storage structure to resemble said grid to some degree. In many High Level Synthesis (HLS) toolchains, one can instantiate a two dimensional memory element by declaring a two dimensional array, however, it is worth considering what FPGA resources that maps to.

A common method of multiplying matrices of arbitrary size using fixed hardware is to partition the two input matrices into smaller blocks of fixed size (with zero padding applied where necessary), and passing these blocks to a convolver, after which the intermediary convolution results are summed and merged back into the desired output matrix.

In order to reuse the convolution, bias addition and activation function kernels, a buffer is required to store matrix data and send blocks to their respective kernels. Instantiating a two dimensional array of matrix elements will yield the employment of BRAM (Block RAM) modules, as opposed to LUT-based memories, except in the case where the matrix dimensions are very small. The matrix may possibly be partitioned over many BRAMs, as their capacities are limited, incurring an additional processing overhead.

The nature in which (16x4) blocks are derived from a matrix is such that 4 consecutive elements are extracted across 16 consecutive columns. If the matrix (when flattened), is stored in row major order, reading the block from a BRAM module would result in 16 bursts of 4 cycle reads. This can be made slightly more efficient by selecting wide dimensions for the block size (e.g. 4x16), resulting in less read bursts, but of longer duration, for which the efficiency benefit is only obtainable if BRAM blocks are configured with larger widths, to allow for the effectively parallel acquisition of consecutive element data in one read cycle. Nonetheless, this approach introduces additional complexities when extracting one block from the storage structure used for the matrix, which is a fundamental function required for the orchestrated marshalling of fixed size data throughout the design.

The double buffer method employs a primary and secondary buffer of equal sizes, allowing the secondary buffer to be written to whilst the primary buffer is still being read from. This method can be used to alleviate the complexity in the temporary storage of output matrix blocks, which cannot be otherwise written to the primary buffer as the region in which it spans contains input data that is still required for subsequent block convolutions. The double buffer method does incur double the amount of BRAM and some additional complexity in the

paging mechanism, however the inefficiently ordered data still presents an inherent complexity overhead. There is the additional issue that this logic must be instantiated for all potential blocks, from which manifests a degree of inefficient utilisation of storage elements and read complexity overhead as not all intermediary matrices (between layers) are of the same size and thus do not partition into the same number of blocks.

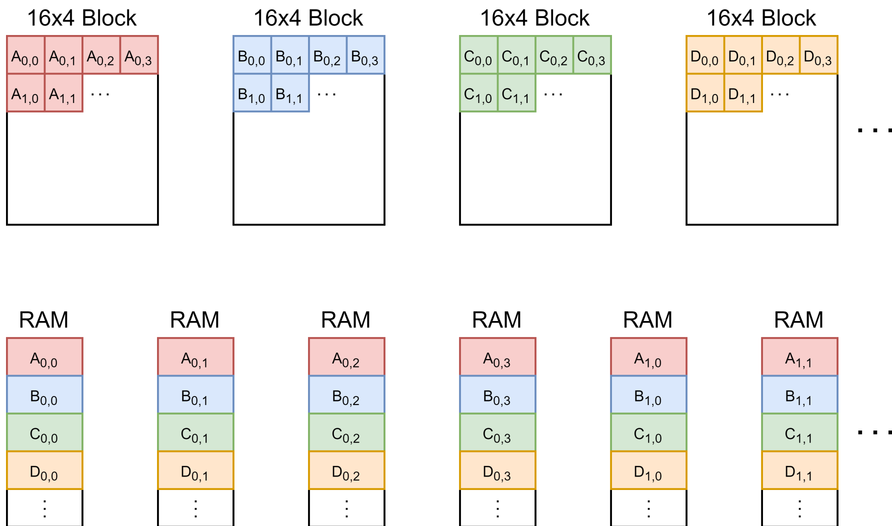

Figure 4. Transition from Block Form to Elementwise RAM Storage

The proposed method of elementwise RAM block indexing transposes the natural view which is taken upon the envision of a partitioned matrix. As opposed to grouping elements based on proximity to one another (row wise, or even block wise), matrix elements can be grouped by position in their block (see Figure 4). In doing so, one would need to instantiate precisely 64 (16x4) RAM modules, one for each of the index positions in a 16x4 block. One efficiency that arises is due to the number of storage elements remaining fixed between convolutional layers, irrespective of the dimensions of the input and weight matrices. The most valuable computational and resource saving derives from the heavily simplified read mechanism. In order to obtain a 16x4 block, all 64 RAMs should be addressed with the same value (block index) and the contents of the block will be presented at the read data ports of the 64 RAM blocks. For convolution, there are only read operations to the RAM blocks, and only writes to the input data matrix for each execution of a forward inference, hence why simplification of the read mechanism presents significant computational and resource savings.

5 Evaluation and Analysis

The methods outlined in sections 3 and 4 were evaluated for their latency and resource utilisation. All implementations were written using MaxJ [5], a Java-like dataflow oriented HLS

language by Maxeler Technologies. All testing was performed on a Xilinx Virtex Ultra-Scale+ VU9P, connected via a PCIe interface to a Intel Xeon Gold 6154 host CPU clocked at 3.00GHz.

5.1 Serialised Cyclic RAM Multiply-Accumulate

A convolution kernel accepting two 16x4 inputs and yielding one 16x16 output was instantiated using each of the four methods shown in the table below. All latency values are expressed as the time taken for output matrix to be obtained.

Hardware Design	**Fully Unrolled**	**Partial Unroll**	**Single MAC Unit**	**Column MAC Units**
f (MHz)	100	100	350	250
Latency (ns)	**80.00**	**210.00**	**140.00**	**24.00**
LUT (%)	26.86	13.24	1.08	17.31
FF (%)	21.61	16.88	1.07	19.22
DSP (%)	29.94	1.86	0.06	0.94
BRAM (%)	8.24	3.11	2.36	2.98
RU (%)	**21.66**	**8.77**	**1.14**	**10.11**

The implementations employing MAC units were able to meet timing for higher clock speeds than the unrolled implementations. The increased latency for the partially unrolled implementation compared against the fully unrolled implementation arises from the serial fashion in which input rows are ingested; despite pipelining the design, the latency of the fully unrolled design cannot be matched. The MAC based implementations both offer very low resource utilisation, especially DSP usage, as a single floating point MAC unit requires 2 DSPs. This reduced resource utilisation is justified by the ability to share resources over rows and channels. For the LLP Jet Tagging Algorithm, three columnwise MAC units should be instantiated (see Figure 3), each with 25, 25, and 4 rows respectively, which when individually instantiated, yield a total resource utilisation of 41.23%. The additional inter layer logic for the activation functions, flattening of the convolutional output layers, and instantiation of the dense layers, further increases the resource utilisation from 41.23% to 69.51%.

5.2 Elementwise RAM Storage

A block marshalling kernel accepting 16x4 blocks as input, storing a 25x17 data matrix and outputting a selected 16x4 block was instantiated using each of the four methods shown in the table below. All latency values are expressed as the time taken to update region of data matrix with inputted 16x4 block, and output a 16x4 block, after the block index has been presented.

A latency reduction can be obtained by opting for a double buffer mechanism as opposed to a single buffer, as the write method for the inputted 16x4 block is simplified (write to secondary buffer) as opposed to the complex method used for the single buffer. The latency for the elementwise RAM storage method significantly outperforms the other two, with a latency reduction by a factor of at least 4. The most valuable advantage of the elementwise RAM storage method lies in the heavily reduced resource utilisation, deriving predominantly from the reduction in LUTs and FFs, attributable to the heavily simplified read mechanism. The BRAM usage is only slightly less than the single buffer method, as the number of blocks

Hardware Design	**Single Buffer**	**Double Buffer**	**Elementwise RAM Storage**
f (MHz)	100	100	100
Latency (ns)	**100.00**	**80.00**	**20.00**
LUT (%)	1.65	3.38	0.71
FF (%)	6.43	13.22	2.90
DSP (%)	0.00	0.00	0.00
BRAM (%)	9.28	18.56	8.12
RU (%)	**4.34**	**8.77**	**1.14**

required for a full forward pass is constant irrespective of the implementation choice, and cannot be further reduced.

5.3 Comparisons to Prior Implementations

A C++ implementation (compiled using g++ with -O3 optimisations) of a convolution kernel accepting two 16x4 inputs and yielding one 16x16 output took 780ns to produce a result when running on a Intel Xeon Gold 6154 CPU clocked at 3.00GHz. Relative to the FPGA implementation of the same convolution kernel (outlined in section 5.1), a ~ 32.5x latency decrease is achieved, indicating the huge potential for FPGAs to be used to accelerate these large models such that they can be ran in real time, negating the need for offline post-processing.

6 Conclusions and Future Work

The approaches discussed in this paper propose alternate implementations to commonly performed operations within machine learning algorithms (matrix storage and convolution); operations used beyond the scope of the LLP Jet Tagging Algorithm, and outside the scope of high energy physics applications. By careful consideration of the nature of the resources available on an FPGA, it is possible to optimise designs to achieve performance enhancements of up to an order of magnitude, whilst still benefiting from the productivity of adopting HLS tools.

Future work involves further investigation of the duplication of MAC units for latency reduction at the expense of increased resource utilisation, and an investigation into quantifying such tradeoffs such that arbitrarily sized networks can be automatically configured as to minimise latency and/or maximise throughput by selecting design parameters derived from the matrix dimensions used in the network architecture.

References

[1] CMS Collaboration, Mach. Learn. Sci. Tech. **1**, 035012. 45 p (2019), 1912.12238

[2] M. Stoye (et al.), Tech. rep., CERN, Geneva (2017), https://cds.cern.ch/record/2293134

[3] G. Apollinari, I. Béjar Alonso, O. Brüning, M. Lamont, L. Rossi, *High-Luminosity Large Hadron Collider (HL-LHC): Preliminary Design Report*, CERN Yellow Reports: Monographs (CERN, Geneva, 2015), https://cds.cern.ch/record/2116337

[4] Tech. rep., CERN, Geneva (2020), final version, https://cds.cern.ch/record/2714892

[5] Maxeler, *Maxcompiler*, https:/www.maxeler.com/products/software/maxcompiler/