Array 12 (2021) 100101

Contents lists available at ScienceDirect

Array

journal homepage: www.elsevier.com/locate/array

Sparse and dense matrix multiplication hardware for heterogeneous multi-precision neural networks☆

Jose Nunez-Yanez [a,*], Mohammad Hosseinabady [b]

[a] Electrical and Electronic Engineering, University of Bristol, Bristol, UK
[b] Consultant in the Area of High-Level Synthesis and FPGA Design

ARTICLE INFO

Keywords:
Neural network
FPGA
Quantization
Pruning
Matrix multiplication acceleration
Convolution
LSTM

ABSTRACT

In this paper, we present hardware accelerators created with high-level synthesis techniques for sparse and dense matrix multiplication operations. The cores can operate with different precisions and are designed to be integrated in a heterogeneous CPU-FPGA system for Edge AI applications. The methodology involves quantization-sparsity aware training and it is applied to a case study consisting of human activity classification. We initially investigate the effects of quantization and sparsity on the accuracy of neural networks with convolution, dense and recurrent layers observing better tolerance to pruning when recurrent layers are present. Then, we propose the hardware accelerators that can switch precision at run-time and work with any matrix size up to a maximum configured at compile time. We compare the performance of these accelerators at different levels of precision and sparsity levels and create a performance model to enable workload balancing. The results show that the proposed sparse matrix multipliers can outperform dense multipliers when sparsity levels are higher than 70% and the improvements are more evident when higher precision arithmetic or structural pruning is used. Additionally, sparsity levels as high as 99% can maintain the level of accuracy required in the network especially when recurrent layers are deployed. Overall, the balance between sparse and dense performance depends on matrix shape, precision, structural pruning and sparsity levels and performance modelling can be used to balance concurrent execution in a heterogeneous configuration.

1. Introduction

Over the last few years, novel hardware for deep-learning in AI from well-known companies and start-ups have entered the market, focusing on high energy-efficiency/performance and low cost. These devices generally focus on supporting network inference, and they analyse data produced near the sensor locations under strict constraints of performance and power. On the other hand, GPUs remain dominant in the area of training [1], although some hardware is optimized for both training and inference such as the Graphcore Intelligence Processing Unit (IPU) [2]. These IPUs are massively-parallel, mixed-precision floating-point processors made available as PCIe cards. The cost and power of these devices mean that they are more suitable for cloud deployments (as done with Google TPUs) rather than at the network edge. At the network edge, real-time inference of deep neural networks (DNNs) on custom hardware has become increasingly relevant with low-precision arithmetic and training frameworks such as the 8 bit EdgeTPU Google devices and TensorFlow Lite [3]. In addition to custom hardware, configurable solutions based on FPGAs have shown significant promise thanks to their ability to fit the architecture to the problem and exploit arbitrary precision arithmetic and sub-byte precision. Very high-performance solutions on FPGAs have been obtained using dataflow architectures and binary arithmetic [4]. The challenge is that such extreme quantization levels can result in a precision degradation, and the dataflow architectures tend to be very resource-intensive for complex neuron models. This limits the size of the neural networks that can be deployed in small FPGA devices. More recently [5], it has been argued that some neuron layers may be more accommodating for aggressive quantization, whereas others may require more expensive arithmetic advocating the advantages of multi-precision arithmetic. A well known approach to accelerate neural network computation is to map the neural network layer arithmetic into large matrix multiplication operations [6] via data duplication and reorganization. Matrix multiplication acceleration based on combined sparse and dense arithmetic with multi-precision arithmetic as proposed in this research could select the optimal hardware configuration depending on the task. Motivated by these observations, this paper makes the following contributions:

☆ Communicated by Diane Dunham Drexle & Hamido Fujita.

* Corresponding author.
E-mail addresses: j.l.nunez-yanez@bristol.ac.uk (J. Nunez-Yanez), mohammad@hosseinabady.com (M. Hosseinabady).

https://doi.org/10.1016/j.array.2021.100101
Received 15 August 2021; Accepted 8 October 2021
Available online 2 November 2021

- We investigate the effects of deep quantization and pruning on accuracy with convolutional and recurrent layers targeting a motion detection application.
- We create high-performance hardware for multiple-precision arithmetic for adaptive dense and sparse matrix calculations and develop simple performance models for run-time use.
- We investigate threshold levels in function of sparsity, precision, structural pruning and matrix shape in which the sparse hardware is faster or slower than the dense.
- We release the IP open-source to facilitate further research in this field at https://github.com/eejlny/gemm_spmm

This paper is organized as follows. Section 2 reviews state-of-the-art hardware for edge deployments showing that current state-of-the-art custom hardware focuses on 8 bit precision and dense matrix operations. It also reviews mixed and arbitrary precision which is more suitable for reconfigurable hardware and concludes that sparse operators are an area open to new research. Section 3 presents our methodology and application problem. Section 4 investigates the effects of accuracy of deep network pruning and quantizing with and without recurrent layers. Section 5 propose the hardware for high-performance dense and sparse multi-precision matrix multipliers. Section 6 compares the performance and complexity of this hardware mapped to a low-cost edge device in the form of a Zynq 7020 device. Finally, Section 7 concludes this paper and proposes future work.

2. Background and related work

In the following sections we present and overview of state-of-the-art hardware for neural network hardware and matrix accelerators.

2.1. 8/16 bit precision hardware

Edge computing is seen as a solution to the latency and privacy issues associated with cloud computing, but a major challenge is running AI workloads on computing resources limited by power and capabilities [7]. Typically, edge devices are specialized for inference and support low-precision arithmetic such as 8 bit integers. For example, Gyrfalcon's matrix processing engine (MPE) uses processor-in-memory techniques to compute neural network models. The 2803 chip [8] delivers 24TOPS/W and consumes a minimum power of 700 mW. Similarly, Google offers a low-cost and low-power version of the Google TPU called EdgeTPU [9] that can run dedicated convolutional neural networks with 8 bit precision. The systolic array size in the EdgeTPU is much smaller than the cloud configuration at around 64 × 64 mult-add cells, resulting in 4TOPS at 480 MHz. FPGA vendors such as Xilinx have also focused on inference, with the Xilinx DPU [10] unit. This unit contains register configuration, the data controller, and convolution computing modules optimized for the FPGA hardware resources. An encapsulated scheduler can assign tasks to multiple DPUs and tools are available, such as Dsight, to monitor performance. It uses a systolic array architecture for matrix multiplications and packs 8 bit operators into the device DSP blocks. Intel also offers FPGA-based solutions that can be targeted with their OpenVino [11] toolkit that has, as main components, the model optimizer and inference engine. OpenVino takes as input a trained network, using a framework such as TensorFlow/Keras, and optimizes it to target Intel hardware that can be their own CPUs, GPUs, FPGAs, or devices such as the NCS2, based on the Myriad Vision Processing Unit (VPU). The VPU has 16 VLIW general-purpose programmable cores, optimized for vision processing workloads, and includes a neural compute engine to accelerate tensor matrix calculations. The VLIW cores also enable other algorithms, unrelated to deep learning, although the current OpenVINO toolset focuses just on supporting recurrent and convolutional neural networks. The VPU uses 16 bit floating-point arithmetic instead of integer arithmetic. However, these more complex data types, compared with fixed-point arithmetic, also imply that performance must be lower at the same silicon size.

2.2. Sub-byte precision hardware

The neural network hardware presented so far does not support sub-byte precision, but this has received significant attention especially in FPGA solutions configured with arbitrary precision types. FINN [4] and hls4 ml [12] are two well-developed frameworks supporting arbitrary precision hardware in FPGAs. Hls4lm has been shown to achieve very high performance on very low latency problems such as particle colliders. It uses quantization-aware training and pruning based on the QKERAS library and exports the resulting configuration as a C++ Vivado HLS description ready to be implemented in the FPGA. The proposed example neural network in [12] consists of 3 dense layers and a softmax layer with a precision of 14 bits with 6 integer bits. Pruning introduces a lot of weights set to zero that do not need hardware resources, reducing the hardware complexity of the dataflow architecture significantly. Pruning does not affect latency because the non-zero weights define the longest path so the depth of the network remains unchanged. FINN also uses this concept of dataflow computing with Vivado HLS C++ descriptions. It focused originally in supporting binarized neural networks on FPGAs and it has now been extended to arbitrary precision. The binarized FINN obtains high performance and low memory cost using XNOR-popcount-threshold data-paths with all the parameters stored in on-chip memory. FINN has a streaming multilayer pipeline architecture where every layer is composed of a compute engine surrounded by input/output buffers. A FINN engine implements the matrix–vector products of fully-connected layers or the matrix–matrix products of convolution operations. Each engine computes binarized products and then compares against a threshold for binarized activation. A similar binarized solution is used in [6] where it initially transfers the convolution operations into matrix multiplication by unfolding and duplicating inputs and rearranging the weights. The system uses a dataflow architecture to map full networks to a large Zynq device 7z100. The dataflow architecture uses a balanced pipeline, with buffers to smooth out the flow of computations, and achieves very high throughput. The limitation in this previous work is the complexity of dataflow computation means that only small networks with few layers and filters can be mapped before hardware sharing is required. An alternative to reduce complexity without the extreme quantization used in binary networks is presented in [13]. This assigns variable bit widths to layers with a general reduction in precision for deeper layers. The PEs (processing elements) support bit-serial multiplication, and the weight bits are sent serially with additional PEs used to compensate for the lower throughput of the bit-serial architecture, which is much simpler than bit-parallel multiplication hardware. The concept of using mixed-precision hardware is also explored in [14] which proposes a scheduling strategy to distribute real-time tasks associated with sensor data acquisition, inference and action on a heterogeneous system that combines an FPGA and CPU. The FPGA inference accelerator supports precisions from 8 to 64 bits, resulting in different computation times that the scheduler needs to consider. Sub-byte neural networks have also been explored in general purposed microprocessor architectures suitable. 3pxnet [15] considers binarized networks so inputs and weights are binary values and can be −1 0 1. It explores a concept called sparse binarized for weights that consists on initially training a ternary network (so values are −1, 1 and 0) and then sparsified so it increases the number of zeros. Then it groups 1, −1 and 0 so the active weights (−1 and 1) are aligned in 32 bit words and the zeros are together in other 32 bit words. Then the active 32 bit words are formed by −1 and 1 and need to be multiplied while the zero words can be ignore. So the ternary matrix has now become a sparse binary matrix. The multiplication with 1 and −1 can be done with XNOR and bitcounting coding (instead of proper multiplication) 1 as 1 and −1 as 0. The overhead is that the active weights aligned and grouped in 32 bit words is a constraint that means that the accuracy of the network suffers. They try to use permutation techniques to compensate for this. In Tfnet [16] focuses on ternary weights and 4 bit inputs and the main

idea is to proposed efficient ways to performance multiple sub-byte multiply accumulate operands with a single 32 bit mult instruction. The approach packs the sub-byte operands into 32 bit operands and swaps the bytes of one of them. The result of performing a standard 32 bit multiplication is that as long as there are no overflows from the low-significant bits the upper 8 bits contain the result of the dot-product operation of the sub-byte components. The limitations of this approach is that functionality is presented using unsigned numbers which could affect the accuracy of the neural network that benefits from using positive and negative weights and the effort of the packing and unpacking operations could dilute the reduce number of mac instructions.

2.3. Sparse/dense matrix hardware

The application of sparse matrix multiplication is not that frequent in FPGA-based neural network hardware. The irregular memory access represents an overhead for hardware acceleration compared with dense hardware, and pruning-aware training must be able to insert a large number of zeros without affecting accuracy for this approach to be worthwhile. Recently, new implementations have been proposed [17] to support these operations in the machine learning field. These algorithms mainly utilize multi-core CPUs or many-core GPUs [18,19]. Several studies have investigated the optimization of SpMV on hardware and FPGAs [20–22]. This research has focused on high-end HPC FPGA-based systems optimizing the irregular memory access to external DRAM data. None have investigated multi-precision support and its impact on accuracy, performance and complexity compared with dense hardware as done in this paper. We call our approach multi-precision because the accelerators are designed to be controlled by a CPU such as the ARM processor available in the Zynq device. In this scenario the CPU offloads matrix multiplication tensors serially to the accelerator and switches on a layer per layer basis between different precisions. This is useful when large models need to be deployed in small devices that cannot support hardware for each layer simultaneously.

High-end cards are also targeted by general libraries for matrix processing created by Xilinx (https://github.com/Xilinx/gemx/) that are specialized for PCIe accelerator cards and large FPGA families (e.g. Xilinx Virtex Utrascale, Xilinx Alveo) and do not support low-end devices such as the Zynq 7000 considered in this research. Another approach targeting high-end accelerator cards is presented in [23] that compares different parallel strategies such as pipelining-dataflow, tiling, vectorization and systolic architectures applied to a matrix multiplication problem. The paper investigates these techniques using a high-end Xilinx Alveo accelerator card and shows the potential of systolic architectures to deliver very high throughput with a large resource usage of logic and memory.

Other previous work has aimed at creating optimized arithmetic circuits specialized on certain types of neural networks such as CNN [24] or LSTM [25]. In this research, we focus on creating a hardware library with dense and sparse hardware multiplier accelerators suitable for FPGA deployment that can then be used with a standard framework such as Tensorflow Lite running on ARM CPUs. The idea is that the proposed hardware library can be used as a compute resource of dense/sparse operators moving execution of the matrix operations from the ARM device to the FPGA on hybrid devices such as Zynq. Although, it is clear that the achievable performance will be lower than customized or pure hardware solutions, we aim at ease of use and general usability. We use pruning/quantization algorithms part of Tensorflow rather than trying to create our own.

3. Methodology and case study

The methodology is based on the recent support of quantization-sparsity aware training added to Keras and Tensorflow. Keras is a popular high-level neural network API written in Python. It is very well integrated with TensorFlow to facilitate neural network development. Recently, to simplify the procedure of quantizing Keras models, QKeras [26] has been proposed as a quantization extension to Keras that provides a drop-in replacement for layers performing arithmetic operations. This allows for efficient training of quantized versions of Keras models. In addition, modern Tensorflow libraries also support adaptive pruning of a neural network using techniques such as polynomial decay where the network is pruned in steps towards a final sparse value and training allows to recover accuracy as pruning is performed to deeper levels. Combining the quantization capabilities of QKERAS and the pruning of Tensorflow enables the study of highly optimized neural models with a reduced number of parameters and bits per parameter. This quantization-pruning aware training libraries and tools are a critical enabling technology to understand the level these optimizations can be deployed without hurting accuracy.

The use case application under consideration consists of an activity detection system using accelerometer and gyroscope sensors to classify user activity into 6 different classes: walking, walking upstairs, walking downstairs, sitting, lying down, and running. All data are normalized to the range [−1, 1]. The accelerometer sensor produces triaxial acceleration (total acceleration) and the estimated body acceleration, while the gyroscope produces triaxial angular velocity [27]. The model consists of convolutional, max pooling, LSTM (Long Short-Term Memory), concatenation, and finally, fully-connected layers, as shown in Fig. 1 and originally proposed in [28]. LSTM are recurrent layers capable of learning long-term dependencies, and they avoid the vanishing gradient problem during training typical of standard recurrent layers. They are not very well supported in edge deep learning accelerators, with support either being on-going work or not possible due to the accelerators' specialized pipelines. The data for the model has been obtained with a sampling rate of 50 Hz and, according to the data set documentation, each of the data samples corresponds to 2.56 s of activity. Consequently, there are 50*2.56 = 128 samples per sensor and axis. There are three axes, so a total of 128 × 3 values form an input into the neural network. There are about 25 samples in each activity window (another activity replaces an activity after 25 samples) in the data set, so each activity window corresponds to approximately 25*2.56 = 64 s.

4. Pruning and quantization accuracy analysis

The combination of quantization and pruning has the potential for significant performance improvements, but any possible impacts on accuracy should be minimized. In this section, we are considering quantization from float to 8 bit, 4 bit (quad) and 2 bit (ternary). We focus on power of 2 precision because our hardware accelerators support a general processor interface with AXI channels that have power of two bit-widths. This simplifies how weights and activations are packed for data transfer. We compare the results with the original float precision, and we also consider binary precision. Binarized neural networks have received significant attention over the last years [29] since as previously indicated the multiplication and additions operations gets reduce to simple XORing and bit counting which can be implemented very efficiently both in hardware and software. Efficient implementations in software require ISAs that support bit counting instructions which is not the case on some popular ISAs such as ARMv6 used in Cortex M microcontrollers but are available in other Cortex A class processor or RISCV. Although this is an area of active research and significant advances have been made on binarized neural network training, it is important to note that previous work has shown that binarized networks affect accuracy significantly especially for more complex problems [16]. This is something we have also observed in the following experiments for this case study.

In a binarized network, only two values are present −1 (coded as 0) and 1 coded as 1, and this implies that pruning is not applicable since there are no 0s in the weights or activations. On the other hand, ternary

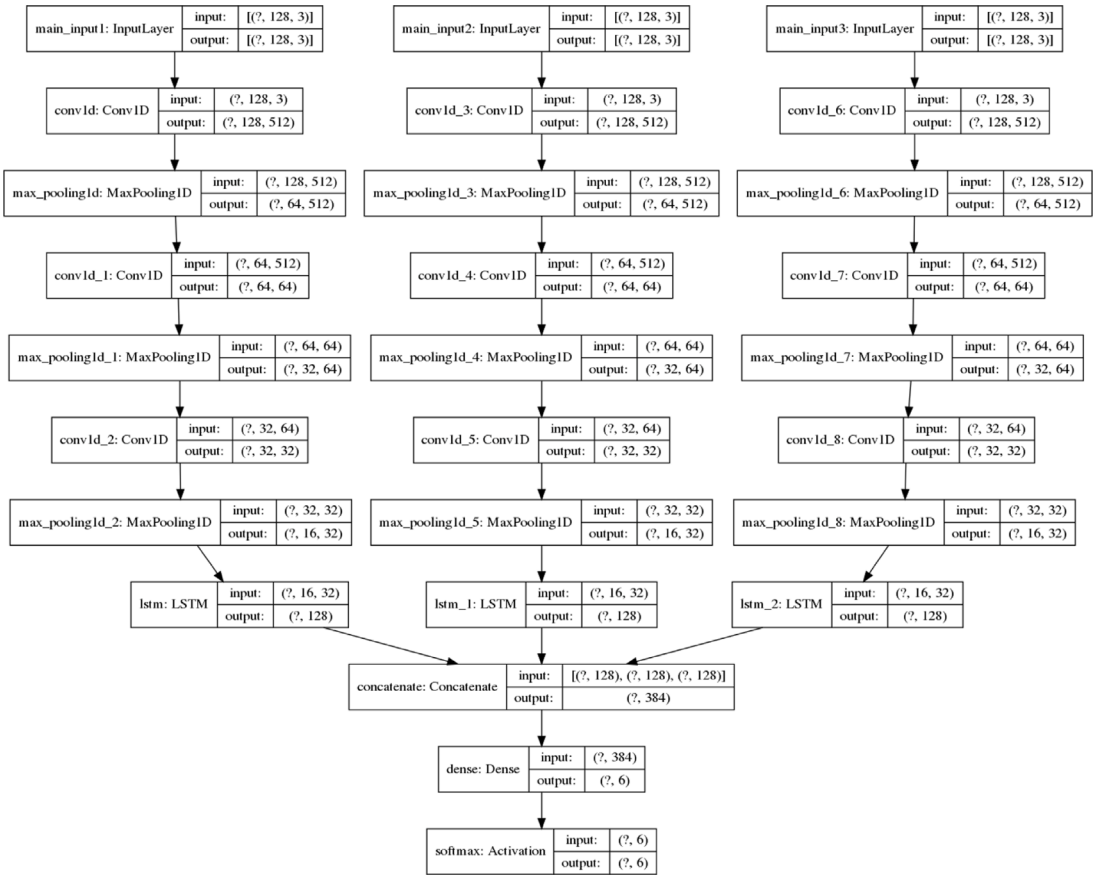

Fig. 1. Motion detection neural network.

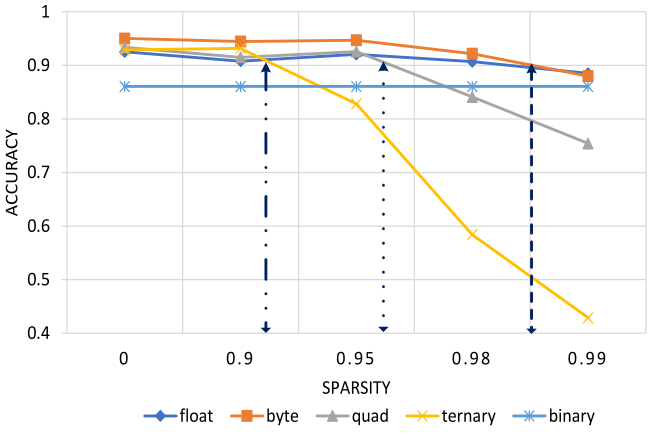

Fig. 2. Sub-byte convolutional accuracy analysis.

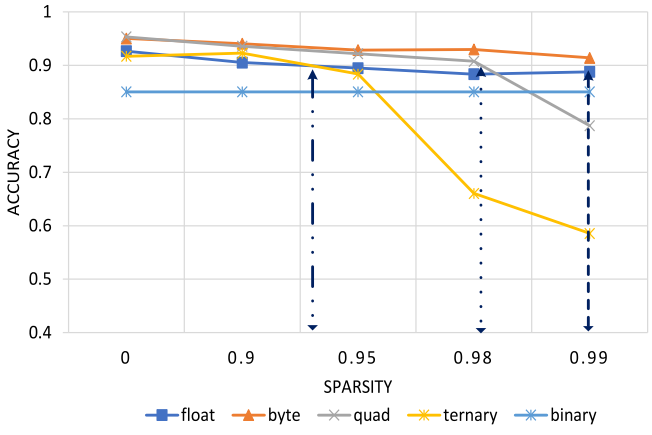

Fig. 3. Sub-byte LSTM https://www.overleaf.com/projectaccuracy analysis.

or 2 bit networks include −1, 0 and 1 and consequently it is possible to prune and increase the number of 0s in the weights. Similar to binary networks, ternary networks do not need multiplication hardware since only subtractions and additions are required. However, the operations require additions and subtractions and are more complex than simple XORing an bit-counting. Additionally, the memory requirements basically double from 1 bit to 2 bits. Figs. 2 and 3 show the accuracy of the neural network of Fig. 1 configured with LSTM layers enabled and disabled. The X-axis shows pruning increasing from 0% (dense) to a maximum possible of 99% (99% of weights are set to zero). The Y-axis shows the accuracy for each of the possible arithmetic precision under consideration. As expected, the binary configuration is not affected by pruning, but it is also evident that the accuracy is significantly lower compared with the other investigated precisions. We have set 90% as the lower accuracy we want to obtain, and the figure shows that this is not achievable by the binarised network. The vertical arrows represent the points at which accuracy degrades below 90% for the 2 bit, 4 bit and 8 bit precisions. This occurs at 90% pruning for 2 bit, 95% pruning for 4 bit and 98% pruning for 8 bit accuracy and the crossing point for 8 bit accuracy is comparable to the floating-point case. It is clear that deeper quantization results in a lower level of acceptable pruning. This is expected since we are removing information from two parameters. Fig. 3 shows the same analysis when the LSTM layers shown in Fig. 1 are enabled. The figure shows a clear displacement of the vertical lines

towards the right, which indicates that the addition of the LSTM layer means that a higher level of pruning can be supported at the same accuracy. This is particularly interesting if we observe the effects of the LSTM layers for the cases with 0% pruning and with floating-point precision. In both cases, we see that for this application, the positive effects of the LSTM layers at full precision and no pruning are very small. On the other hand, the prune and low precision networks show the positive effects of the LSTM layers in a much clearer way. For example, with LSTM layers, the 8 bit precision network can support pruning to 99% and maintain the required accuracy higher than 90%. We can conclude that the addition of LSTM layers helps compensate for the loss of information caused by pruning and quantization.

5. GEMM and SPMM accelerator hardware

Fig. 4 compares the parallelism exploited in the GEMM (General matrix multiplication) and SPMM (SParse Matrix dense Matrix Multiplication) hardware which use tiling, vectorization and streaming.

5.1. GEMM hardware

GEMM is well suited for parallel computing since all the multiplications of the row and column elements can be done in parallel. A naïve implementation in hardware will simply buffer matrix A and B in the FPGA, and it will then compute the multiplication. This is effective, but it is very resource-intensive and can quickly overwhelm the BRAM and DSP resources available in a small device such as the zynq7020 considered in this work. Our preliminary studies show that the maximum matrix size possible is limited to around 256 × 256 elements. An alternative is to buffer only one row of A and a number of B columns and then work in the multiplication in parallel as shown in 4 for the GEMM case. The amount of parallelism is defined by two values BW and BH with higher values meaning that more DSP blocks work in parallel. In this approach a tile is formed by one row of A and a number of columns of B with BW and BH defining the amount of coarse vectorization while finer vectorization is done at the 32 bit word level depending on precision. So if the BW value is 16 and BH is 2 then the level of coarse vectorization is 32 and the level of fine vectorization depends on the precision and can be 4, 8 or 16. In each clock cycle a number of elements from the A row defined by BH are multiplied in parallel by a number of elements from B_block defined by BW*BH producing BW partial results as shown in Fig. 5. A larger BH implies that more accesses are done in parallel to the internal memory buffers so additional memory partitioning is needed. We create hardware using Vivado HLS, and C++ in which the maximum matrix sizes possible is defined by the values SN*SM and SM*SP, and the hardware supports the multiplication of any combination of matrix shapes up to these values. The main constrain is that memory allocated in the FPGA needs to be sufficient to store the area BW* SM. There is a trade-off between the largest matrix supported, and the amount of parallelism exploited that depends on the value of BW which is defined at compile-time. At run-time, the user can deploy any matrix size up to the maximum values choosing any matrices below these maximum dimensions. This is important in neural network applications since as it can be seen in the different layers of Fig. 1, the matrix size will change depending on the active layer. Packing weights and activations in GEMM is simple, and we define all interfaces at 32 bit widths for all matrices so in a word there are 4 8 bit, 8 4 bit and 16 2 bit values. For example, setting the maximum matrix size supported to 2048 × 2048 32 bit words allows a value of BW of 32, and this means that 128, 256 and 512 multiplications are performed in parallel for the different precision widths of 8 bit, 4 bit and 2 bit, respectively. Notice that in the 2 bit ternary implementation the values of −1 and 1 can be handle with subtractions and additions and the value of zero can be ignored but in our FPGA hardware, the synthesis engine obtains a more efficient implementation without DSPs using simply the multiplication operator instead of using an if-else approach to define subtractions and additions. This approach supports large matrix sizes with feasible implementations in small FPGA devices as shown in Section 7. An alternative to the proposed architecture are the systolic approaches proposed in [23] that can deliver very high throughput but they tend to be optimized for larger devices. For example, tests of the systolic approach used in [23] on a Zynq Ultrascale MPSOC ZU9 device which is approximately 5–10 larger than the Zynq 7020 used in this research reveals that a matrix size of 2048 × 2048 integers requires 400% percent of the available memory resources. Tiling strategies could be used to reduce these requirements however this is not readily available in [23] to evaluate overheads. Fig. 6 shows the interfaces used by the GEMM HLS block. A 2 bit input precision switches between 8 bit, 4 bit and 2 bit operating modes. 32 bit inputs are used to bring packed data from memory for input matrices A and B while outputs are generated non-packed in output C with a precision of 16 bits to avoid possible overflows. Additional inputs are available to define the shape of the matrices being multiplied.

5.2. SPMM hardware

The SPMM hardware is based on our previous work of an SPMV accelerator presented in [30] that has been extended to support multiple precisions and full matrix multiplication of a sparse matrix (weights) and one dense matrix (activations). The SPMV accelerator has been designed to support the popular CSR (Compressed Sparse Row) format to store sparse matrices, and it avoids having to buffer full chunks of the input matrices using a streaming dataflow architecture. It consists of an input stage, streaming mapping layers, compute stage, streaming mapping layers and output stage. Fig. 8 shows a simplified view of hardware corresponding to one thread and how the stream buffers provide sequential access to the col_index, values and row_index data, representing the sparse matrix. The computational engine reads the row_index data and calculates the number of elements in each row. Note that the number of elements in each row is the difference between two adjacent indexes in the row_index stream. Using this number of elements in each row, the compute engine reads data from col_index and values. The col_index content is used to access the proper element in x vector (i.e., the dense matrix vectors) pre-fetched into the local memory. Then the x element is multiplied to the values data and accumulates in a register inside the compute engine. After finishing the computation associated with a row, the result is sent out as the corresponding y value. Then, the compute engine reads the next element from the row_index stream, ready for calculating the next y element. Dataflow optimizations are also used to overlap how the multiple columns of the activation matrix are processed in SPMM.

The streaming mapping layers reformat and distribute the data received from its input buffers among its output buffers. The compute stage consists of a number of parallel processes or threads that work in parallel in different areas of the non-zero input values. In this work, the number of processes or hardware threads is 4, that is the maximum number that can be supported with the memory and logic resources available in the Zynq 7020 device. Fig. 4 shows for SPMM how each hardware thread works on a tile that is formed by the full matrix B and variable size A depending on how many non-zeros are present in each block. Each hardware thread is responsible for processing the same number of non-zeros and the distribution of work among the hardware threads is done automatically by the hardware. The original hardware targets 32 bit floating-point precision and all the interfaces for matrix A, B, and C had 32 bit widths. We are now deploying integer precision of 8, 4 and 2 bits and this implies that an important consideration is bit packing. The FPGA implementation tools do not support hardware AXI interfaces with sub-byte width. Observing sparse matrix A, we could pack our sub-byte elements into 32 bit elements, but this results in an important trade-off. The CSR format we use means that any non-zero sub-byte element will be packed into a non-zero 32 bit word that will

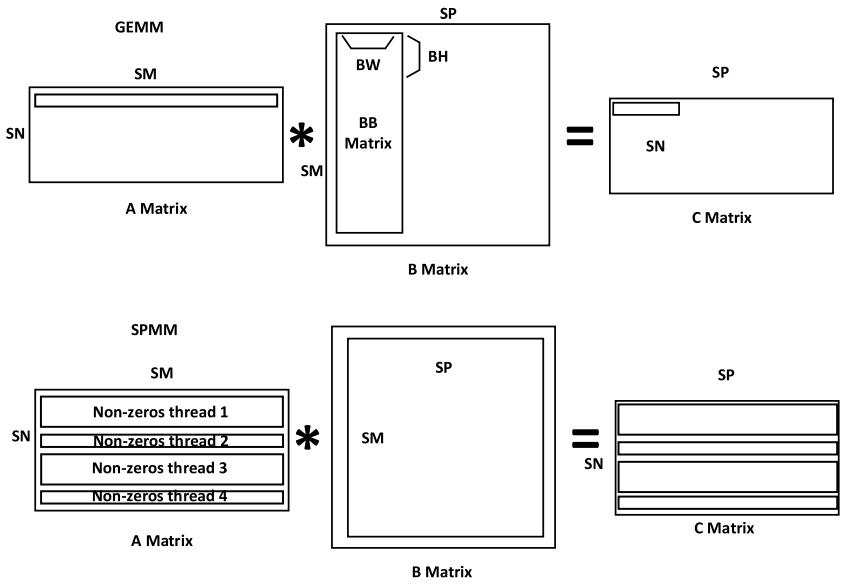

Fig. 4. GEMM tiled-vectorized and SPMM tiled-streaming hardware.

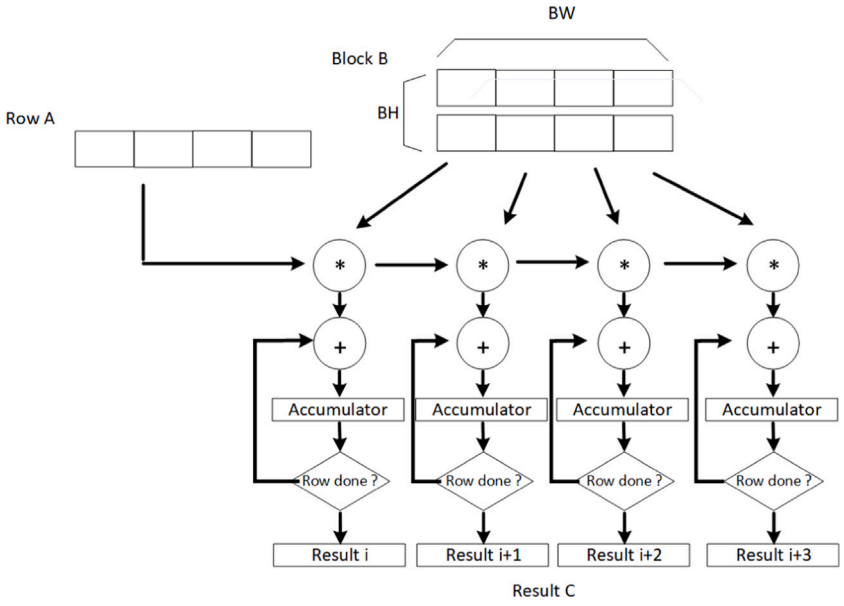

Fig. 5. Main GEMM kernel.

need to be processed by the accelerator. If we follow this strategy, then a sub-byte matrix sparsified to 99% will be converted into a packed 32 bit word matrix where most elements are non-zero, and the effects of pruning will be diluted. Two possible alternatives are to lower the level of packing for matrix A (for example, from 32 to 8 bit) or to use structural pruning techniques to constrain pruning so non-zero values are group together. This research uses Tensorflow that includes partial support of structural pruning in Model Optimization extensions and the details of how structural pruning can be used to increase the width of the datapath and improve performance is investigated in Section 7. Matrix B is simpler to deal with since it is a dense matrix and all data has to be processed, and consequently we can simple packed into 32 bit widths to optimize the data transfers over the AXI interfaces similarly to the GEMM hardware. Fig. 7 shows the interfaces used by the SPMM HLS block using a 32 bit internal architecture. A 2 bit input precision switches between 8 bit, 4 bit and 2 bit operating modes. Each hardware thread has a set of interfaces to access A matrix data in CSR format. One interface is shared between the hardware threads to B dense data. Additional interfaces are used to define the matrix shape and the number of non-zeros present in the sparse matrix A. Outputs are generated independently by each thread with a 16 bit precision to avoid possible overflows in hardware.

6. Performance and complexity analysis

To have a fair comparison between the SPMM and GEMM hardware we have set the maximum matrix size for both accelerators to 4096*4096 32 bit words and obtain hardware that uses up the available resources in the selected FPGA Zynq 7020. This means that further parallelism is not possible due to either logic, DSP or memory limitations. The resources that limit further parallelism can be either DSP blocks, LUTs or BRAMs and will depend on the design. This maximum matrix sizes limit the value of BW to 16 for GEMM. The value of BH is set to 2 since a higher BH although it does not increase the amount of BRAM needed, it fails to meet the timing constraints. The number of parallel processes is set to 4 for SPMM. This immediately indicates an initial

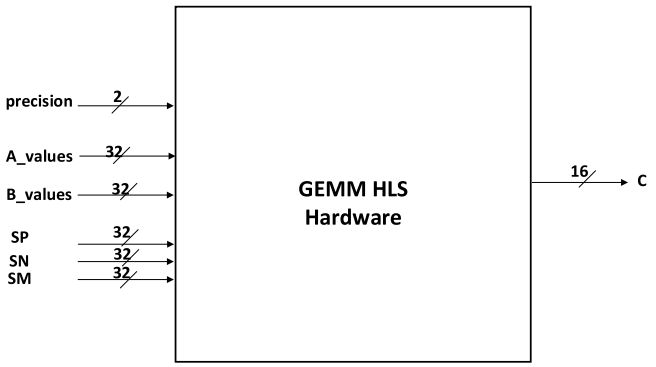

Fig. 6. GEMM hardware interfaces.

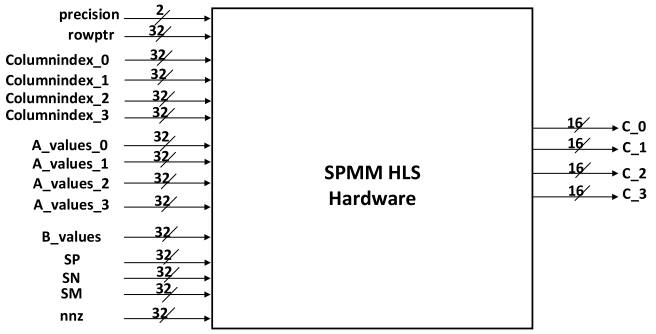

Fig. 7. SPMM hardware interfaces.

Table 1
Maximum matrix size supported.

Precision	32 bit	8 bit	4 bit	2 bit
Size	4096 × 4096	16384 × 16384	32768 × 32768	65536 × 65536

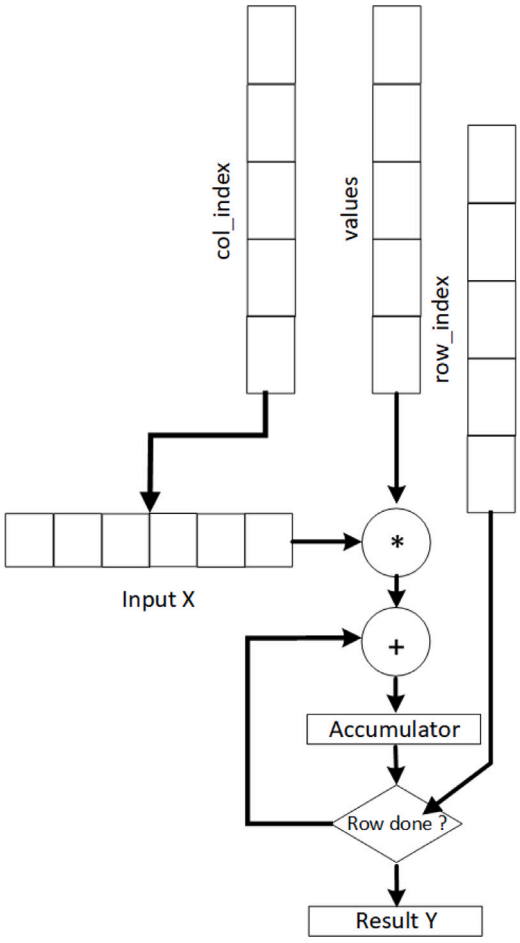

Fig. 8. Thread SPMM hardware.

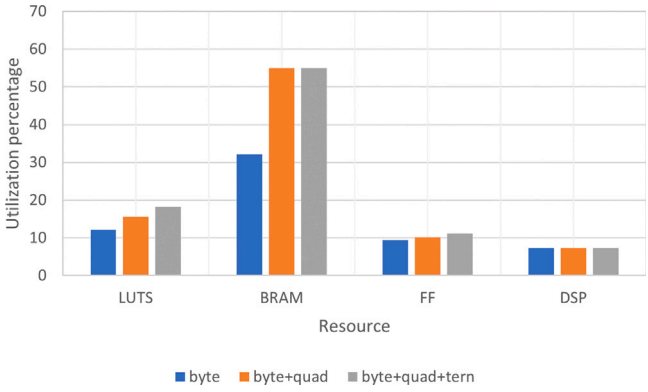

Fig. 9. GEMM resource analysis.

advantage for GEMM since it will be able to perform 16*2=32 parallel 32 bit operations. In comparison, SPMM is limited to 4 operations with multiple data. Each of these operations multiplies 4, 8 or 16 values depending on precision, so it is clear than lower precision hardware results in higher parallelism and consequently higher performance.

Figs. 9 and 10 compares the resource requirements of GEMM and SPMM for the multi-precision implementations. The GEMM implementation is BRAM limited and to increase the level of parallelism will result in not enough BRAM resources available to store the active chunk of matrix B. BRAM and LUTs limit SPMM, and adding additional execution units is not possible due to these two constraints. GEMM shows that BRAM utilization increases significantly when support for 4 bit and 2 bit arithmetic is needed, and this is the result of mapping values to memory blocks, so that parallel reads are possible. The final addition of 2 bit arithmetic support does not result in significant complexity increase. The complexity of SPMM is dominated by the data reading and preparation stages, and the complexity of the computation stage remains largely invariant with the multi-precision hardware. The maximum matrix sizes that can be multiplied for both SPMM and GEMM depend on the active precision and are illustrated in Table 1.

To analyse performance we have obtained quantized and pruned weight and activation data from the convolutional and LSTM layers of the network of Section 3. We use QKERAS quantizers functions quantized_bits for 8 and 4 bit precision and ternary for 2 bit and sparsity is deployed with a polynomial decay starting at 50% and with a first step at 0.2*end_step. The end_step depends on the size of the training data, the batch size and the number of training epochs. The final sparsity is defined to values 90%, 95% and 99%. We then use Python to transpose, reshape the weight data to obtain the full matrices. These full matrices are packed into matrices with 32 bit values containing a variable number of weights depending on precision and also processed to obtain the corresponding CSR formatted matrices used by SPMM. Fig. 11 compares the performance of GEMM and SPMM hardware as a function of the hardware precision and the level of sparsity. The circles indicate the crossing points between SPMM and GEMM. It can be seen that SPMM outperforms GEMM for sparsity levels higher than around

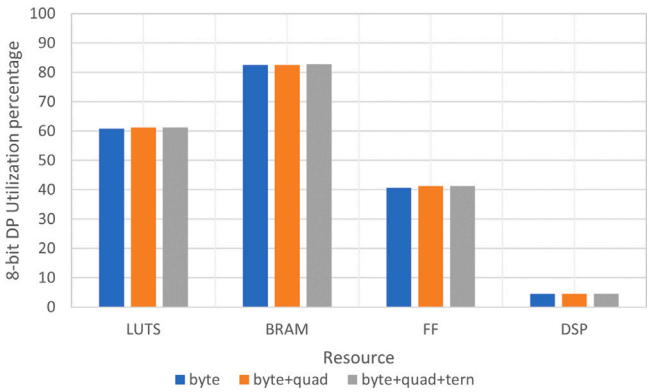

Fig. 10. SPMM resource analysis.

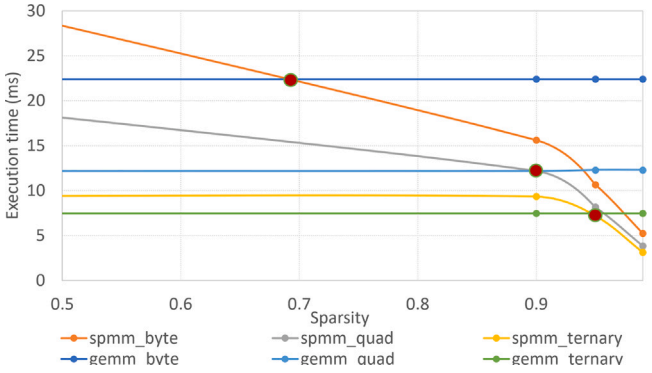

Fig. 11. GEMM vs SPMM hardware performance in Zynq7020.

70%, 90% and 95% for 8 bit, 4 bit and 2 bit precision respectively. This shows that as the precision decreases the performance of the SPMM hardware reduces compared with GEMM hardware, which is a consequence of packing overheads. The reason is that in SPMM if 1% of the numbers are non-zeros after packing, it is unlikely that these non-zeros are packed together, so the number of non-zero values remains largely constant and independent of the precision. The effect is that the hardware efficiency with lower precision benefits from fewer transfers of matrix B that packs values into 32 bit words but not from a lower number of operations. In Section 7 we investigate how this can be improved deploying the structural pruning available in the Tensorflow model optimization extensions. On the other hand, GEMM can easily increase the number of packed values in the 32 bit words, and this means that with lower precision more values are packed together without overheads. The efficiency of the GEMM hardware increases with both a reduction in the number of transfers and a reduction in the number of arithmetic operations.

In GEMM, we can reason that as the size of the maximum matrix supported increases, the required BRAM increases proportionally and this results in having to reduce the value BW that defines the amount of compute parallelism present in the accelerator. On the other hand, the architecture of SPMM means that performance is not affected by the maximum size supported and it is just limited by the requirement to buffer one full column of matrix B and matrix A values. This organization means that SPMM is more effective with large matrix sizes. Large matrix sizes are being shown relevant in the neural network field especially with new architectures such as transformers [31]. These advanced models try to solve the problem of long-term dependencies using convolutional neural networks together with attention models and could benefit from sparse arithmetic.

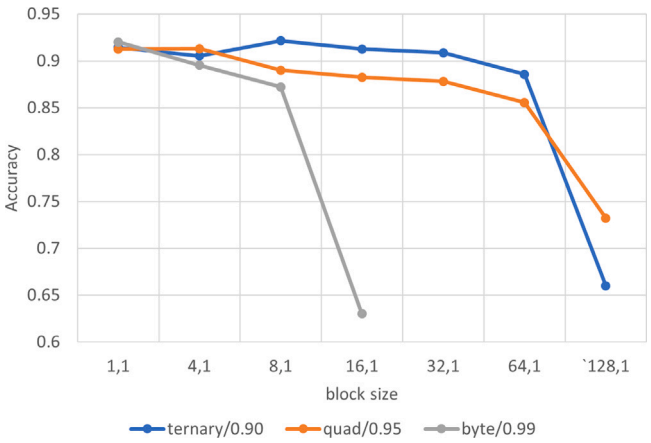

Fig. 12. Effects of structural pruning in network accuracy.

7. Structural pruning optimization

In the previous section we have identified that the sparse hardware performance degrades compared with dense hardware with lower precision arithmetic due to packing overheads. The quantized 2 bit and 4 bit values are packed into 32 bit values and the nature of the packing and pruning strategy means that in many cases only one active non-zero value is present in the 32 bit pack and the rest of the pack components are set to zero and do not contribute to the computation. This hardware limitation has also been observed in GPUs and structural pruning or block sparse techniques have been proposed to alleviate it. In block sparse, pruning is applied in coarser block sizes that can be defined by the designer. The version of Tensorflow 2.4.1 used in this work supports block sparsity for two dimensional weights used by the dense layers of the network. In this section we investigate the accuracy and performance impact of structural pruning. We consider a block_size value that defines the height and width of the block that should be used during pruning. For the proposed SPMM architecture the optimal block sizes are one dimensional since they ensure that the processing pipeline is fill with active non-zero values. In the ternary 2 bit configuration and with a 32 bit datapath width, the optimal block_size is (16, 1). Similarly the optimal block_size for the quad and byte precisions are (8, 1) and (4, 1) respectively.

Fig. 12 investigates the possible effects on network accuracy introduced by the additional block sparsity constraint for different levels of sparsity . We observe that for the block_sizes that the hardware can benefit from, there is no impact on accuracy although this can degrade significantly for larger block_sizes that will constraint the network excessively. For example, we can observe in the figure that for the byte precision blocking should not go higher than (4, 1) without negatively affecting accuracy.

Figs. 13 and 14 compare the number of packed non-zero values that must be processed by the hardware considering 32 bit and 8 bit packed values. A lower value means that there will be fewer values that need to be processed by the sparse multiplier hardware resulting in better performance. The (1, 1) case corresponds to a configuration that does not use block sparsity. In this scenario the number of active values for the byte precision is comparable for the 32 bit and 8 bit packed cases. This means the wider data-path will not result in performance improvements. On the other hand, Figs. 13 and 14 show that block sparsity at values of (4, 1) or higher significantly reduces the number of values that must be processed by the SPMM hardware. If we compare the number of active values for the 8 bit and 32 bit packing strategies, we can observe that 32 bit packing clearly reduces the number of values to be processed by a factor of 3–4 depending with block sizes larger than (1, 1).

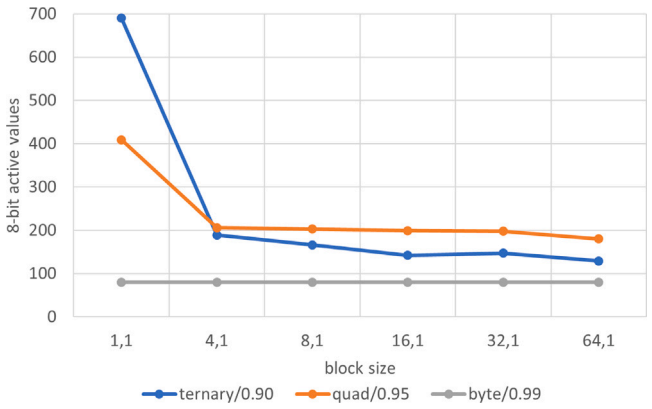

Fig. 13. Effects of structural pruning in active 8 bit packed values.

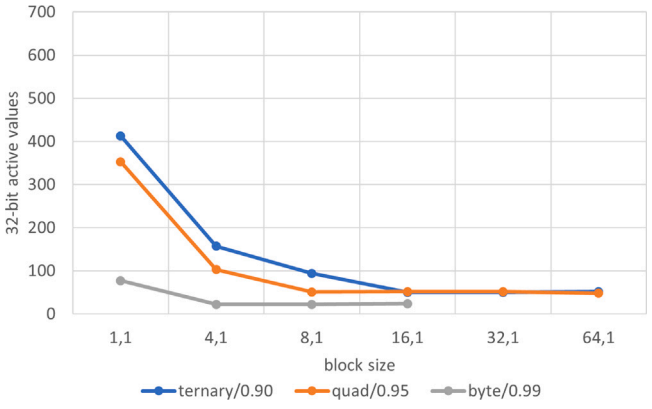

Fig. 14. Effects of structural pruning in active 32 bit packed values.

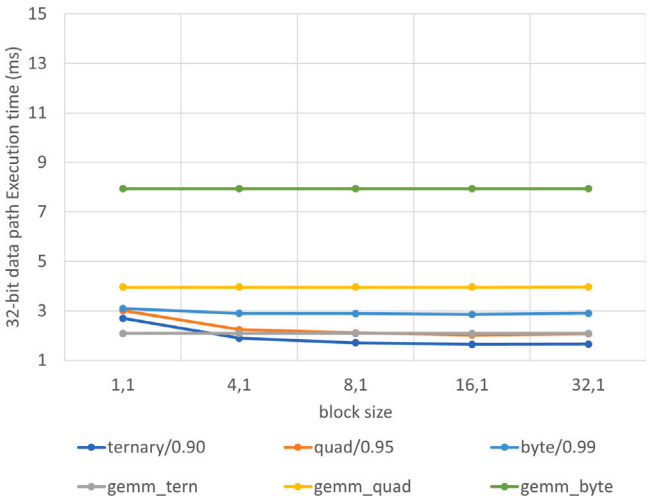

Fig. 15. Structural pruning performance with 32 bit hardware.

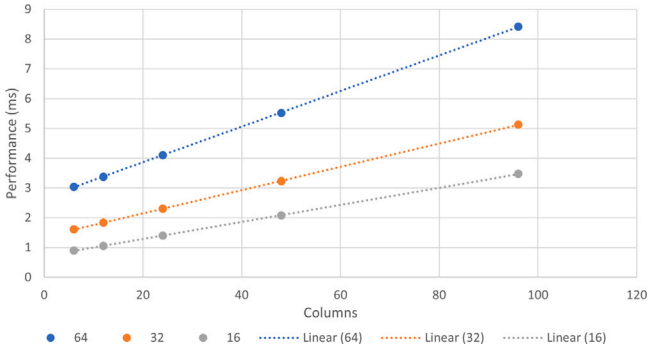

Fig. 16. Dense performance dependencies.

Finally, Fig. 15 shows the performance and dense and sparse hardware when this additional constrain of block sparsity is deployed. Currently, structural pruning in Tensorflow only supports Dense 2D layers and work is on-going to extend this feature to other layers. The sparsity levels considered are of 90% for ternary, 95% for quad and 99% for byte. Fig. 15 shows that despite the significant pruning, for the (1, 1) case the sparse hardware outperforms the dense hardware for the quad and byte cases but for the ternary case this is not the case. Once structural pruning is introduced all the sparse configurations perform better than the dense hardware. The performance improvement is much more modest than the reduction in number of active values. This can be explained due the dataflow nature of the architecture that overlaps computation with loading of the dense matrix. The interface used to load the dense matrix is 32 bit and for higher levels of sparsity it becomes the dominant factor limiting performance.

8. Performance modelling

In the previous section we have proposed two optimized architectures for GEMM and SPMM and analyse their performance when deploying them in the same device. The architectures exploit coarse and fine grain level parallelism and multiple precision levels. Overall, we have observed that SPMM can outperform GEMM with levels of sparsity higher than 70% (depending on arithmetic precision). We have also seen that deploying structural pruning on the proposed architecture can benefit SPMM performance significantly while it should have no effect on GEMM. In this section we consider that an array of sparse and dense accelerators are available in the SoC and we need to distribute the workload so each computing resource can contribute to finalize the task as early as possible. The workload consists of multiple network layer with different shapes, sparsity levels and arithmetic precisions so the preferred configuration depends on the layer. The variables that will affect this workload balancing are: matrix size and shape, sparsity level, structural pruning and arithmetic precision. To achieve workload balancing we derive performance models that can be used by the scheduler to calculate an optimal subdivision of work. We aim at obtaining simple models that can be used at run-time to divide the work. Analytical models can be derived using the performance estimation available in the Xilinx toolset. However, these models tend to underestimate the effects of memory accesses. An alternative is to obtain measurements and derive use multiple linear regressions where the multiple independent variables.

8.1. GEMM model

The performance model is defined by the sizes of the matrices being multiplied. Different arithmetic precisions result in different levels of packing into the 32 bit values so the effects of precision in performance are already accounted for in the packed matrix size. To derive the performance model we plot the execution time with different matrix data in Fig. 16. The figure does not show any curvatures suggesting that the relations are linear. On the other hand, the non-parallel lines suggests product terms or interactions between the explanatory or independent variables. The general equation is shown in Eq. (1). We take multiple measurements and solve a linear regression problem to obtain the coefficient values A, B and C and the constant D obtaining a percentual error of 2.86%.

$$p = A \times rows + B \times columns + C \times rows \times columns + D \quad (1)$$

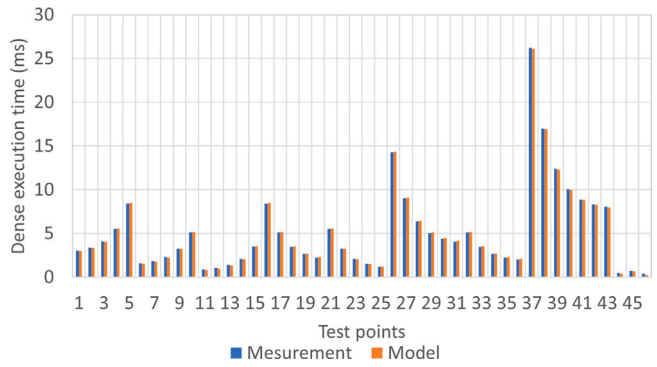

Fig. 17. Dense model performance error.

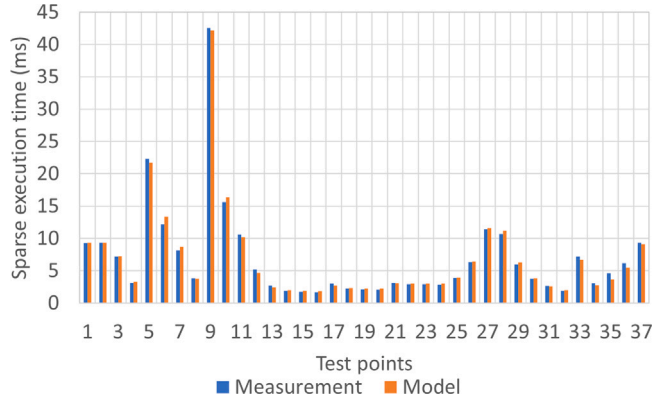

Fig. 18. Sparse model performance error.

$$p = 0.020063584 \times rows + 0.041137792 \times columns + 0.000482719 \times rows \times columns - 0.043234957 \quad (2)$$

The performance model does not compute an absolute value of performance since it ignores the number of columns of matrix B that is a dense matrix for both operators. This is because the number of columns of matrix B will scale the obtained performance index by the same amount for sparse and dense so they will not vary the performance relations of both operators that are required to achieve the workload balancing. Fig. 17 shows the accuracy of this simple model for different test points that correspond to matrix of different sizes.

8.2. SPMM model

The A matrix in SPMM is sparse and the number of rows and columns are replaced by the number of non-zeros present in the CSR structure. The number of columns in matrix A defines the vector length from Matrix B that must be loaded into memory before computation with non-zeros starts. The number of vectors present in matrix B is defined by the width of Matrix B and it does not need to be considered in the performance index value since it will affect in the same way the performance of dense and sparse operators. The general equation is shown in Eq. (3) where active_values are 32 bit values with non-zero components. We follow the same process as the dense operator to solve the linear regression problem and obtain the corresponding coefficients.

$$p = A \times active\_values + B \times columns + D \quad (3)$$

$$p = 0.001587 \times active\_values + 0.004734 \times columns + 1.356 \quad (4)$$

Fig. 18 shows the accuracy of this simple model for different test points that correspond to matrix of different sizes. The average percental error observe is 5.5%.

9. Conclusions and future work

In this paper, we have investigated the effects of deep pruning and quantization on a motion detection application that uses convolution, LSTM and dense layers with multiple heads and data capture from accelerometers. The initial results have shown that under deep quantization and pruning LSTM layers are very effective at limiting accuracy degradation. We have then proposed acceleration hardware for low-cost FPGAs for dense and sparse matrix multi-precision multipliers suitable for performing dense, LSTM and convolutional operations. The results show that the sparse hardware cannot deploy the same level of parallelism as the dense hardware at the same level of resource usage. However, pruning enables the sparse operator to outperform the dense multipliers. These hardware accelerators are relevant to deploying large neural networks in resource constraint devices that will be less suitable for dataflow models. In the dataflow model, dedicated hardware exists for individual neurons allowing very high performance but with high hardware requirements. Also in a dataflow model if certain layers require longer processing time than others (e.g. recurrent versus convolutional) the whole pipeline could stall. Our approach is suitable for hybrid software–hardware neural networks where a standard microprocessor chooses which hardware engine GEMM or SPMM to use depending on matrix size, precision and level of sparsity. For example, small matrices with no pruning will favour GEMM while larger matrices with pruning levels higher than 70%–80% can be mapped to the SPMM accelerator. One of the advantages of this hybrid approach is that novel techniques such as transformers can be easily deployed in software without affecting the hardware accelerators. Our future work involves deploying an array of dense and sparse accelerators as a part of hardware library and use the performance models to schedule operators to these accelerators to improve performance compared with a homogeneous systems where a single type of accelerator is available.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research was funded by the Royal Society Industry fellowship, INF\R2\192044 Machine Intelligence at the Network Edge (MINET), EPSRC HOPWARE EP\RV040863\1, EPSRC ENEAC EP\N002539\1.

References

[1] Chen J, Ran X. Deep learning with edge computing: A review. Proc IEEE 2019;107(8):1655–74.
[2] Jia Z, Tillman B, Maggioni M, Scarpazza DP. Dissecting the graphcore IPU architecture via microbenchmarking. 2019.
[3] David R, Duke J, Jain A, Reddi VJ, Jeffries N, Li J, Kreeger N, Nappier I, Natraj M, Regev S, Rhodes R, Wang T, Warden P. TensorFlow lite micro: Embedded machine learning on TinyML systems. 2020.
[4] Umuroglu Y, Fraser NJ, Gambardella G, Blott M, Leong P, Jahre M, Vissers K. Finn. In: Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays. ACM; 2017.
[5] Dong Z, Yao Z, Cai Y, Arfeen D, Gholami A, Mahoney MW, Keutzer K. HAWQ-V2: Hessian aware trace-weighted quantization of neural networks. 2019.
[6] Xu Z, Cheung R. Binary convolutional neural network acceleration framework for rapid system prototyping. J Syst Archit 2020;109:101762.
[7] Zhou Z, Chen X, Li E, Zeng L, Luo K, Zhang J. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. Proc IEEE 2019;107(8):1738–62.

[8] Sun B, Yang L, Dong P, Zhang W, Dong J, Young C. Ultra power-efficient CNN domain specific accelerator with 9.3TOPS/Watt for mobile and embedded applications. 2018.
[9] Murshed MGS, Murphy C, Hou D, Khan N, Ananthanarayanan G, Hussain F. Machine learning at the network edge: A survey. 2020.
[10] Zhu J, Wang L, Liu H, Tian S, Deng Q, Li J. An efficient task assignment framework to accelerate DPU-based convolutional neural network inference on FPGAs. IEEE Access 2020;8:83224–37.
[11] Liu Y, Wang Y, Yu R, Li M, Sharma V, Wang Y. Optimizing CNN model inference on CPUs. 2019.
[12] Duarte J, Han S, Harris P, Jindariani S, Kreinar E, Kreis B, Ngadiuba J, Pierini M, Rivera R, Tran N, et al. Fast inference of deep neural networks in FPGAs for particle physics. J Instrum 2018;13(07):P07027.
[13] Hsu L-C, Chiu C-T, Lin K-T, Chou H-H, Pu Y-Y. ESSA: An energy-aware bit-serial streaming deep convolutional neural network accelerator. J Syst Archit 2020;111:101831.
[14] Jiang W, Song Z, Zhan J, He Z, Wen X, Jiang K. Optimized co-scheduling of mixed-precision neural network accelerator for real-time multitasking applications. J Syst Archit 2020;110:101775.
[15] Romaszkan W, Li T, Gupta P. 3PXNet: Pruned-permuted-packed XNOR networks for edge machine learning. ACM Trans Embed Comput Syst 2020;19(1).
[16] Yu J, Lukefahr A, Das R, Mahlke S. TF-Net: Deploying sub-byte deep neural networks on microcontrollers. ACM Trans Embed Comput Syst 2019;18(5s).
[17] Sun S, Monga M, Jones PH, Zambreno J. An I/O bandwidth-sensitive sparse matrix-vector multiplication engine on FPGAs. IEEE Trans Circuits Syst I Regul Pap 2012;59(1):113–23.
[18] Liang Y, Tang WT, Zhao R, Lu M, Huynh HP, Goh RSM. Scale-free sparse matrix-vector multiplication on many-core architectures. IEEE Trans Comput-Aided Des Integr Circuits Syst 2017;36(12):2106–19.
[19] Tang WT, Tan WJ, Goh RSM, Turner SJ, Wong W. A family of bit-representation-optimized formats for fast sparse matrix-vector multiplication on the GPU. IEEE Trans Parallel Distrib Syst 2015;26(9):2373–85.
[20] Li S, Wang Y, Wen W, Wang Y, Chen Y, Li H. A data locality-aware design framework for reconfigurable sparse matrix-vector multiplication kernel. In: Proceedings of the 35th international conference on computer-aided design. New York, NY, USA: Association for Computing Machinery; 2016.
[21] Umuroglu Y, Jahre M. An energy efficient column-major backend for FPGA SpMV accelerators. In: 2014 IEEE 32nd international conference on computer design. 2014, p. 432–9.
[22] Fowers J, Ovtcharov K, Strauss K, Chung ES, Stitt G. A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication. In: 2014 IEEE 22nd annual international symposium on field-programmable custom computing machines. 2014, p. 36–43.
[23] de Fine Licht J, Besta M, Meierhans S, Hoefler T. Transformations of high-level synthesis codes for high-performance computing. 2020.
[24] Lentaris G, Chatzitsompanis G, Leon V, Pekmestzi K, Soudris D. Combining arithmetic approximation techniques for improved CNN circuit design. In: 2020 27th IEEE international conference on electronics, circuits and systems. 2020, p. 1–4.
[25] Ghasemzadeh SA, Bank-Tavakoli E, Kamal M, Afzali-Kusha A, Pedram M. BRDS: an FPGA-based LSTM accelerator with row-balanced dual-ratio sparsification. CoRR 2021;abs/2101.02667.
[26] Coelho Claudionor N, Kuusela Aki, Li Shan, Zhuang Hao, Ngadiuba Jennifer, Aarrestad Thea Klaeboe, Loncar Vladimir, Pierini Maurizio, Pol Adrian Alan, Summers Sioni. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. Nature Machine Intelligence 2021;3(8). URL http://dx.doi.org/10.1038/s42256-021-00356-5.
[27] Garcia-Gonzalez D, Rivero D, Fernandez-Blanco E, Luaces MR. A public domain dataset for real-life human activity recognition using smartphone sensors. Sensors 2020;20(8).
[28] Ordóñez FJ, Roggen D. Deep convolutional and LSTM recurrent neural networks for multimodal wearable activity recognition. Sensors 2016;16(1).
[29] Courbariaux M, Hubara I, Soudry D, El-Yaniv R, Bengio Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or −1. 2016.
[30] Hosseinabady M, Nunez-Yanez JL. A streaming dataflow engine for sparse matrix-vector multiplication using high-level synthesis. IEEE Trans Comput-Aided Des Integr Circuits Syst 2020;39(6):1272–85.
[31] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I. Attention is all you need. 2017.

Jose Nunez-Yanez is a Reader (associate professor) in adaptive and energy efficient computing at the University of Bristol and member of the microelectronics group. He holds a Ph.D. in hardware-based parallel data compression from the University of Loughborough. His main area of expertise is in the design of reconfigurable architectures for signal processing with a focus on run-time adaptation, parallelism and energy-efficiency. In 2006–2007 he was a Marie Curie research fellow at ST and in 2011 he was a Royal Society research fellow at ARM Ltd, Cambridge. He is currently an industrial research fellow with the Royal Society at Sensata technologies.

Mohammad Hosseinabady is currently a consultant working in the area of high-level synthesis and FPGAs. Prior to that he was a research fellow at the University of Bristol on high performance and energy efficient heterogeneous computing with CPUs, FPGAs and GPUs. He received the Ph.D. degree in computer engineering from the University of Tehran in 2006. His research interests include high-level synthesis for FPGA, high-level reliability and testability, reconfigurable architectures, dynamic resource management, and runtime power management. He has published more than 50 papers on these topics in journals and conference proceedings.