Variable Precision Multiplication for Software-Based Neural Networks

Richa Singh
Virginia Tech
richas@vt.edu

Thomas Conroy
Virginia Tech
tconroy@vt.edu

Patrick Schaumont
WPI
pschaumont@wpi.edu

***Abstract*—As the number of applications of neural networks continues to grow, so does the need to efficiently perform inference computations on highly constrained devices. In this paper, we propose a methodology to accelerate neural networks in software. We exploit the limited-precision requirements of typical neural networks by formulating recurring operations in a bit-slice computation format. Bit-slice computation ensures that every bit of an M-bit processor word contributes useful work even while computing a limited-precision n-bit (with $n < M$) operation. This paper brings the following contributions. We first present an environment to efficiently create bitslice descriptions in software, by synthesizing them from Verilog. We then develop bitsliced designs of matrix multiplication and evaluate their performance.**

Our target is a small microcontroller, and we rely solely on software optimization. Our driving application is a neural network classifier for the MNIST database. Range-Based Linear Quantization in symmetric mode quantizes pre-trained 32-bit floating point weights and activation to low-precision data-widths. Experiments on RISC-V with varying levels of hardware-support show that for data-widths common to neural network applications, the bit-sliced code produces a speedup over traditional methods, which leads to faster and efficient inference without incurring significant loss in accuracy. For example, 8-bit matrix multiplications are sped up by a factor of 2.62× when compared with non-bitsliced *rv32i* ISA implementation with no hardware multiplier.

***Index Terms*—Bitslice compilation, Neural networks, Software Acceleration**

I. Introduction

The advent of the neural network computing paradigm in embedded context has opened a wealth of applications in inference and feature extraction from complex data. A Neural Network implements progressive feature extraction of data using layers of neurons. Each neuron requires the multiplication of input data with coefficients to obtain the output. The number of layers and the size of the dot products varies according to the type of neural network (NN). A common observation in most neural networks is that the required numerical precision for each layer is limited. Previous work in hardware-accelerated NN has used this property to create optimized fixed-point hardware implementations, such as 16-bit operations in DaDiannao [4] and variable-precision bitserial operations in Stripes [7]. The Stripes project demonstrated that many NN can operate at reduced precision (3 to 13 bits) with negligible loss in classification accuracy. A custom-hardware implementation with reduced computation accuracy may improve performance and reduce

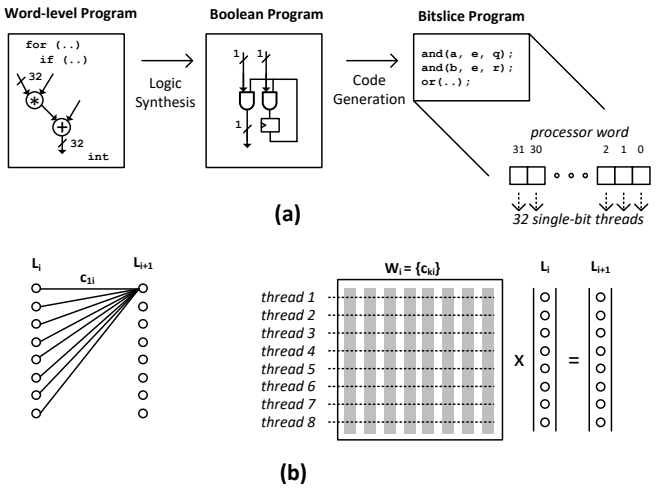


Fig. 1: (a) Bitslice Software Generation and Execution (b) Layers in a neural network are evaluated using matrix operations

energy consumption. Moreover, this improvement is achieved with every additional bit of precision reduction. Converting the precision of computations in a hardware design from 16-bit precision to p-bit precision may provide a performance improvement of up to 16/p times for computation-dominated designs [7].

In software implementations, adapting the precision of computations to the application requirements is not as straightforward. Standard processor architectures do not offer the same per-bit performance improvement as in hardware. Processor architectures are optimized for a few different wordlengths (say 32-bit, 16-bit and 8-bit, for a 32-bit embedded processor). Therefore, the arbitrary shortening of the wordlength in fixed-point precision applications does not offer a commensurate performance improvement. Some processors provide sub-vector operations (e.g. ARM NEON). However, such SIMD extensions are generally unavailable on small microcontrollers.

In this contribution, we describe an acceleration technique that depends only on software to improve performance of the dot product, the most common operation in a NN. We propose a bitslice formulation of an optimized fixed-point implementation of the NN. Bitslice programming is a technique that was originally proposed in the context of cryptographic software acceleration [2]. It has been applied to a wide range

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

of ciphers [9], [1] and countermeasures [12], [6]. Figure 1a illustrates the design steps leading to a bitslice program. A standard program with word-level operations is first converted using logic synthesis into a Boolean program, a description in terms of logic gates. Boolean programs can be created using RTL synthesis from a hardware description [14], or else through dedicated logic synthesis tools [10]. The complexity of a Boolean program, i.e. the number of logic gates, is proportional to the complexity of the word-level program.

A bitslice program is created out of a Boolean program as a sequential execution of the gate-level description using bitwise instructions (`and`,`or`,...). A bitslice program on an N-bit processor thus consists of *N* parallel copies of the Boolean program. The N-fold parallel nature of bitslice programs is of great benefit to computing NN layers. Figure 1b illustrates the evaluation of layer L_{i+1} of an NN as a matrix multiplication of a vector L_i with a coefficient matrix W_i. To map the operations efficiently to a bitslice program, the elements of L_{i+1} are evaluated in parallel by columnwise multiply-accumulating W_i with L_i.

In this paper, we describe a tool to generate bitslice programs from RTL word-level programs, and we apply the tool to demonstrate acceleration of a quantized NN. The remainder of the paper is organized as follows. In the next section, we discuss related work in software acceleration of NN. In Section III, we summarize an open-source tool to generate bitslice software. Section IV describes how we implemented bitsliced matrix multiplication, and section V summarizes the results achieved. We then conclude the paper.

II. Related Work

There is a rich body of work on neural network accelerator architectures for the edge computing environment [11]. Our efforts concentrate on software-only optimization for small micro-controllers. Our key contribution is a technique to exploit the limited precision of multiply-accumulate operations, the most frequently occurring operation in neural networks.

Earlier research demonstrated that full floating point precision is not required for most neural network architectures. The precision of input weights and coefficients can be reduced to fixed point precision with a limited range of n bits. This leads to a smaller hardware footprint, as well as an improved energy efficient for hardware and software. The reduction in numerical complexity is potentially significant. In XOR-Net [13], for example, the multiply-accumulate operation is replaced with a bitwise XNOR operation and a population bitcount. However, the challenge of such single-bit precision network is to maintain sufficiently high classification accuracy. The authors of the STRIPES accelerator architecture evaluated the minimum required wordlength for several popular neural network architectures to maintain their classification accuracy [7]. They find that typical network architectures require wordlengths of between 3 bit (LeNet) and 10 bit (GoogleNet) for full baseline accuracy. This is still much less than the precision of the standard integer instruction set of a 32-bit micro-controller (32-bit addition, 32-bit multiply).

Therefore, it remains a challenge to map these small-wordlength neural network architectures efficiently into the standard wordlength of a microcontroller. Clearly, a straightforward mapping is inefficient, as most of the bits in the 32-bit datapath would remain unused in an 8-by-8 bit multiply-accumulate.

A recent solution to this problem was realized in TF-Net [15], which aims at small microcontrollers. The authors propose a neural network architecture with two-bit weights and four-bit activations. To optimize the multiply-accumulate operation, TF-Net proposes a packed-MAC operation as follows. TF-Net arranges four two-bit weights on the byte boundaries of a word, and then multiplies these weights with a single four-bit input to obtain four six-bit products aligned on the byte boundaries of a word. These products can then be accumulated (using standard integer `add`) with other products resulting from the same packed-MAC. Thus, TF-Net exploits a parallelism factor of four to compute the output activations. In combination with other optimizations, the authors of TF-Net show that this technique improves performance by a factor of 1.83 compared to the baseline [15]. Although TF-Net demonstrates how the micro-controller architecture can be operated more efficiently, the solution is still limited to specific wordlength combinations of weights and inputs.

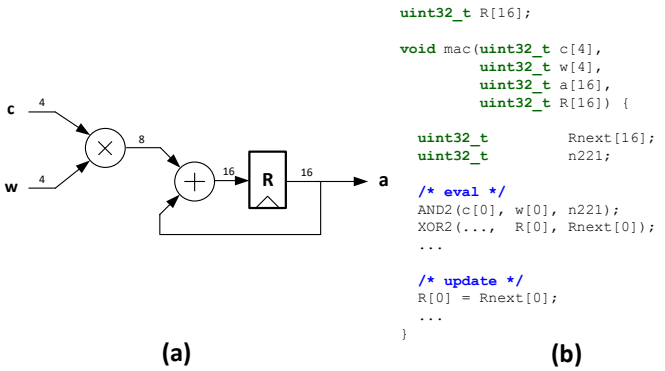
CMIX-NN is another solution inspired by Digital Signal Processing, and demonstrates a library optimized for short-wordlength weights and activitations of 8, 4, 2 and 1 bit [3]. By developing optimized custom-instruction designs, CMIX-NN minimizes the overhead of packing and unpacking smaller wordlengths (1, 2, 4 and 8 bit) into regular wordlengths (16 bit). However, the computations are done using a standard `MAC16` multiply-accumulate, which means that performance stays roughly constant when going to smaller wordlengths.

The bit-serial software implementations by Cowan *et al.* are most similar to our proposed approach [5]. The authors hand-optimize a matrix multiplication for low-precision evaluation, and implement each multiply-accumulate operation as a bit-serial design. Bits are then packed into processor words. Next, they hand-optimize the schedule, for a matrix multiplication based on these bit-serial designs. The main difference of this approach with ours is that we automate the conversion to the bit-serial implementation.

Our proposed solution, using bitslicing, is aimed at small microcontrollers. We propose a bitslice version of the multiply-accumulate (MAC). We create the bitslice program automatically using logic synthesis from Verilog and code generation in C [8]. Another solution, Usuba, is a recent effort that offers a high-level specification and synthesis of bitsliced software [10].

Bitslicing is a popular technique in the cryptographic community to build high-performance implementations of algorithms [2], [9], as well as to build side-channel-protected and fault-protected implementations [12], [6]. We are not aware of its application to matrix multiplication and neural network (NN) computation.

978-1-7281-9219-2/20/$31.00 ©2020 IEEE




```
uint32_t R[16];

void mac(uint32_t c[4],
         uint32_t w[4],
         uint32_t a[16],
         uint32_t R[16]) {

  uint32_t         Rnext[16];
  uint32_t         n221;

  /* eval */
  AND2(c[0], w[0], n221);
  XOR2(...,  R[0], Rnext[0]);
  ...

  /* update */
  R[0] = Rnext[0];
  ...
}
```

Fig. 2: (a) Multiply-accumulate in Verilog (b) Generated bitsliced C code

III. Bitslice Compilation

In this section, we describe our solution to create bitslice software from a standard word-level description. We rely on a technique that uses logic synthesis to create a bitsliced version of the implementation [8].

Figure 2 illustrates an example. The input in Figure 2a is a register-transfer level description of a multiply-accumulate (MAC) function. The MAC multiplies 4-bit weights and inputs, and accumulates the 8-bit result using a 16-bit accumulator. This MAC is a building block for the matrix multiplication used in NN. Using logic synthesis, the MAC is converted into a netlist of primitive gates including AND, OR, NOT, XOR as well as a state element. The netlist is then converted into C code by topologically sorting of the gates and replacing each gate with an equivalent bitwise operation. Our C code generation is implemented as a backend in the YOSYS open-source synthesis environment [14]. Figure 2b illustrates the generated C code. Each input or output is mapped into a corresponding `uint32_t` representing 32 parallel slices. In other words, the C program implements 32 parallel copies of Figure 2a. The state element is mapped into a global variable to preserve state across `mac` invocations. The function computes synchronously and each call corresponds to one logical clock cycle of the input description in Figure 2a.

Before a standard n-bit input can be processed by a bitsliced function, the input has to be transposed. We convert 32 n-bit inputs into n 32-bit inputs for the bitsliced function. When bitslice processing is completed, the reverse operation is done: m 32-bit outputs are reverse-transposed into 32 m-bit outputs. Transposition introduces overhead similar to packing and unpacking in other low-precision approaches [15]. However, within the bitsliced domain, bitsliced functions are compatible with each other. The transposition overhead can be ammortized over multiple layers of a neural network.

IV. Bitsliced Matrix Multiplication

Matrix-vector Multiplication forms the core of computational operation in the fully-connected (FC) layers of a NN. In this section, we apply the bitsliced design of a MAC operation to implement an optimized version of the matrix-vector multiplication. We assume the case of a standard fully-connected multi-layer network.

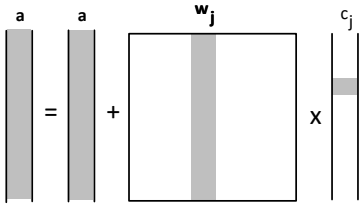


Fig. 3: parallel MAC to compute the matrix multiplication.

We first explain how the matrix multiplication is mapped into parallel MAC operations. Next, we present a storage technique that minimizes address calculation overhead during matrix multiplication. Our results, presented in the next section, are based on this optimized implementation.

A. Mapping of matrix multiplication into parallel MAC operations

A bitslice design needs fewer operations for low-precision operations because a bitslice program is formulated in terms of bit-operations. When fewer bits have to be computed, a commensurate reduction in bit-operations can be expected. However, the Boolean programs at the basis of bitslice designs compute on only one single bit. Bitslice designs still have to agressively parallelize the Boolean program over every slice of the processor. We discuss how this is achieved for matrix multiplication. The matrix multiplication takes coefficients w_{ij} and multiplies them with inputs c_j to obtain outputs a_i.

$$a_i = \sum_{j=0}^{31} w_{ij} \cdot c_j \tag{1}$$

This loop can be implemented as a multiply accumulate operation over the column index j. To parallelize the MAC operation over the matrix multiplication, all a_i are computed in parallel. We thus compute 32 MACs at the same time. Figure 3 visualizes the parallel MAC as it iterates through each column of the weights w_{ij}. We obtain the following formulation, where $\mathbf{a}$ indicates the result vector and $\mathbf{w}_j$ column j from the weight matrix.

$$\mathbf{a} = \mathrm{MAC}_{j=0}^{31}(\mathbf{a}, \mathbf{w}_j, c_j) \tag{2}$$

This formulation computes 32 NN activations in parallel. Large dimensions are supported through repeated application of this 32-parallel MAC.

B. Memory organization

An important consideration in the implementation of bitslice design is the memory organization of inputs. Bitslicing tends to be demanding on memory, because these programs handle parallel copies of a Boolean program. Thus, for an n bit processor, the storage requirements tend to increase n fold. We applied two optimizations to implement the scheme of Figure 3 in a performance-effective way. Both optimizations exploit the fact that the data organization of coefficients can

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

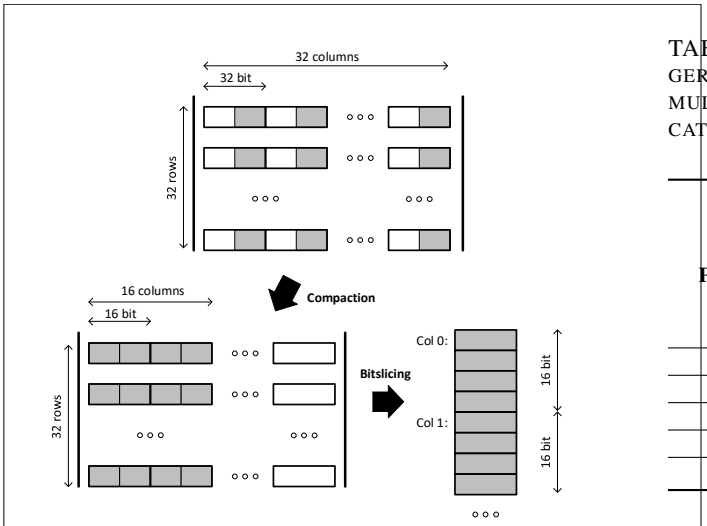


Fig. 4: Compaction to minimize storage cost of bitslice designs; example illustrates case of 16-bit coefficients.

be optimized at compile time, and with the full knowledge of coefficient widths.

First, we produce compact representations of the bitsliced coefficients as follows (Figure 4). A given 32-by-32 array with k bit coefficients is compacted into a k-by-32 array. Assuming $k \in \{2, 4, 8, 16, 32\}$, this will allocate an integer multiple coefficients in each `uint32_t`. Next, the resulting array is bitsliced. This leads to a compact representation with efficient address computation. The bitsliced target address for bit $v1$ of coefficient $v2$ will be $v1 + v2 * k$ rather than $v1 + v2 * 32$, leading to contiguous memory use and simplified addressing.

Second, one can observe that in the parallel MAC scheme of Figure 3, the elements of the weight matrix are accessed *columnwise*. The traditional storage order of arrays (eg. in C) follows a row-major storage order. This makes the address computation challenging, since there will be row-sized jumps to go from one element in column w_j to the next one below it. To avoid this addressing overhead, we adopted a column-major storage order of coefficients.

The combination of the two optimizations leads to a very compact and efficient form for the bitsliced matrix multiplication. The following shows the code that is used for a k-bit bitsliced multiply accumulate loop. In this code, `trans_coef` and `trans_c` are bitsliced inputs, `trans_mac_out` is the bitsliced output, and `state` is the bitsliced MAC state.

```
uint32_t trans_coef[32 * k];
uint32_t trans_c[32 * k];
uint32_t trans_mac_out[32];
uint32_t state[16];
for (j=0; j<32; j++)
    mac_top(&(trans_coef[j*k]),
            trans_c[j*k],
            trans_mac_out,
           &state);
```

V. Results

a) Experimental Setup: In our experiments, we programmed an FPGA Zedboard with 20MHz RISC-V core-based PULPissimo SoC. We generated the bitslice program from the Verilog code using a bitslice generator integrated in YOSYS [14]. Our driver design is a small four-layer network with a 784-32-32-10 structure trained for the MNIST database. We generate bitsliced C code for MAC at a particular bit-length by synthesising the Verilog model with a `BITWIDTH` parameter appropriately chosen using YOSYS. The C code is compiled using the open-source RISC-V GNU Compiler Toolchain for PULP.

TABLE I: Performance of Bitsliced, Non-Bitsliced Integer without Hardware multiplier and with Hardware multiplier implementations of matrix-vector multiplication at different bit precisions

Bit Precision	Number of cycles in matrix-vector multiplication		
	Bitsliced	Non-Bitsliced Integer without Hardware Multiplier	Non-Bitsliced Integer with Hardware Multiplier
2-bit	2613	119711	10418
4-bit	12309	149667	10418
8-bit	66870	175594	10418
16-bit	345174	212845	10418
32-bit	1492374	282649	10418

For the comparison between bitsliced and non-bitsliced implementations of matrix multiplication, we generated pseudorandom coefficients matrixes and input vectors, having various bit-lengths (using the Mersenne Twister pseudorandom number generator). The performance was measured in CPU cycles by reading the hardware performance counters on the CPU before and after the matrix multiplication. Each experiment for the non-bitsliced integer implementation was repeated 100 times (randomizing the input arrays each iteration), and the performance was determined as the average cycle count for matrix multiplication. For bitsliced experiments, only one iteration was needed since the bitsliced design's performance is data-independent due to its branch-free code structure. The bitsliced MAC function can compute 32 multiply-accumulate operations in parallel and internal accumulate registers are zeroed on reset. We have evaluated the different implementations of Matrix-vector multiplication using a fixed 32x32 dimension matrix and 32x1 vector. This implies that 32 iterations of bitsliced MAC function will traverse all the 32 columns of matrix with each iteration being computed in parallel. The cycle count for the bitsliced implementation is calculated over 32 iterations of this bitsliced MAC function.

b) Performance Analysis: In Table I, we report the performance numbers of the proposed bitslice implementation of matrix-vector multiplication against two non-bitsliced integer implementations. The first non-bitsliced integer case is generated using the *rv32i* RISC-V base integer ISA (no hardware multiplier). The second case is generated for the *rv32im* RISC-V ISA, which uses a hardware multiplier. As the bit precision increases, the number of cycles in bitsliced version increases (roughly quadratically) since the number of logic gates in the the gate-level synthesis of Verilog code for MAC are proportional to the required bit precision. The

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

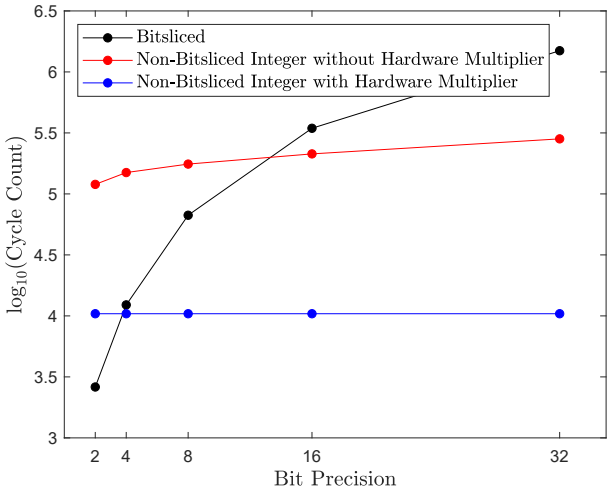


Fig. 5: Analysis of cross-over point in Matrix-Vector Multiplication

number of logic gates further translate into a proportional count of bitwise instructions in the bitsliced program, thereby, leading to slower performance at higher bit lengths. Cycle counts for the *rv32i* architecture have been measured as an average over 100 iterations because the software emulation of integer multiplication, *mulsi3*, has a data-dependent execution time. This multiplication technique is based on conventional shift and add algorithm. Performance comparison of the bitsliced implementation with *rv32i* case clearly shows that the bitsliced implementation is faster than the *rv32i* at 2-bit, 4-bit and 8-bit precisions. It also highlights that the bitsliced approach proves to be beneficial for the the most common low numerical precision formats used for weights and activation to maximize the throughput of matrix multiplication in a Neural Network (NN). When comparing the speed of the bitsliced implementation with the *rv32im* target architecture, the bitsliced implementation is faster only for 2-bit precision. This indicates that the architecture with built-in 32x32-bit hardware multiplier has greater design efficiency than the bitsliced software multiplication technique.

In Figure 5, CPU cycle count as a function of bit precision is shown in order to determine the cross-over point of our proposed bitsliced approach with different ISA implementations for matrix multiplication. We observe that bitsliced approach is faster than the other two implementations at smaller bit-lengths. And the cost of extra data movement instructions with the increase in bit-precision of bitsliced code imply a certain threshold where bitsliced approach starts to perform better. We get the first crossover point as approximately 13 bits below which bitsliced approach performs better than the non-bitsliced integer with no hardware multiplier support. Second crossover point is present at approximately 3 bits below which bitsliced approach is faster than the non-bitsliced integer with hardware multiplier enabled.

c) Overhead Analysis: Table II shows our analysis of bitsliced code at different bit precisions in terms of CPU cycle count, instruction breakdown, overhead of load-store instructions and code density. Bitslice code is compiled with optimization for size (`-Os`) enabled since we are programming for a small embedded microcontroller architecture. As a result, numbers reported in table for code size of bitsliced code at varying bit precision is small enough to fit in our benchmark architecture. Additionally, due to the linear structure of the bitsliced function, the smaller the code size, the fewer cycles it takes to execute. This meant size optimization was both smaller and performs better than the `-O3` GCC compiler option.

We observe that the number of `OR`, `NOT`, `XOR` and `AND` instructions which perform the MAC computation are increased by a factor of about 5× with doubling bit precision. Furthermore, we observe that moving data from memory to processor becomes expensive with increasing bit precision. The increased register pressure leads to a larger proportion of load-store instructions at 16-bit and 32-bit precisions, which explains the slower performance of the bitsliced approach at 16-bit and 32-bit precisions. The overhead related to spilling is about 30-50% in terms of instruction count.

The number and composition of logical bitwise instructions originates directly from the Boolean netlist generated from the the original description. The overhead of loads and stores, however, are introduced by the compiler because the width the netlist (i.e., in this case the number of active variables in the code) exceeds the number of registers available, causing register spilling into memory.

To investigate how effective the register use is, and to put it in perspective with spilling, we analyzed the lifetime of variables for a 4-bit bitsliced MAC (Figure 6) and for an 8-bit bitsliced MAC (Figure 7). In this figure, the X-axis indicates instruction count and the Y axis indicates storage resources. The top-half of the Y axis corresponds to the 27 general-purpose registers of RISC-V. The bottom-half of the Y axis corresponds to stack memory locations. Colored bars indicate active variables stored in a register or on the stack. Each color change indicates a different variable.

We conclude that the compiler has obtained a tight register allocation, as there is hardly any register space/time left unutilized. The stack utilization is less uniform. The deepest positions in the stack are used for callee-saved registers. The space utilization of the shallow stack positions is non-uniform over the execution time of the algorithm. Figure 7 reveals a faint inverted pyramid shape. This is expected as the number of live variables in a multiplication grows when partial products are created, and subsequently shrinks when the partial products are added and accumulated.

d) Classification Accuracy: Finally, we evaluated 8-bit precision bitsliced matrix multiplication on a 4-layer Neural Network architecture using hand-written digits classification dataset, MNIST, in terms of Top-1 accuracy as shown in the Table III. MNIST database consists of 60,000 training and 10,000 test images. We pre-trained the network with hard sigmoid activation function and during inference, linear symmetric Quantization scheme is applied to represent full-

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

TABLE II: EVALUATION OF MATRIX-VECTOR MULTIPLICATION AT DIFFERENT BIT-WIDTHS ON 20MHZ RISC-V SOFT CORE SOC. OVERHEAD IS CALCULATED AS THE NUMBER LW/SW INSTRUCTIONS AS A PERCENTAGE OF THE TOTAL

Bit Precision	Number of cycles in matrix-vector multiplication	Instruction Mix									Overhead (%)	Code Size (KB)
		OR	NOT	XOR	AND	ADD	ADDI	LUI	LW	SW		
2-bit	2613	5	6	9	15	0	0	0	17	8	41.67	9.932
4-bit	12309	77	23	36	100	0	2	0	82	43	34.44	10.98
8-bit	66870	451	119	196	489	0	2	0	605	192	38.80	16.52
16-bit	345174	2211	394	848	2109	0	2	0	3885	1227	47.88	47.664
32-bit	1492374	9499	1968	3009	8727	0	2	0	17369	5729	49.88	183.564

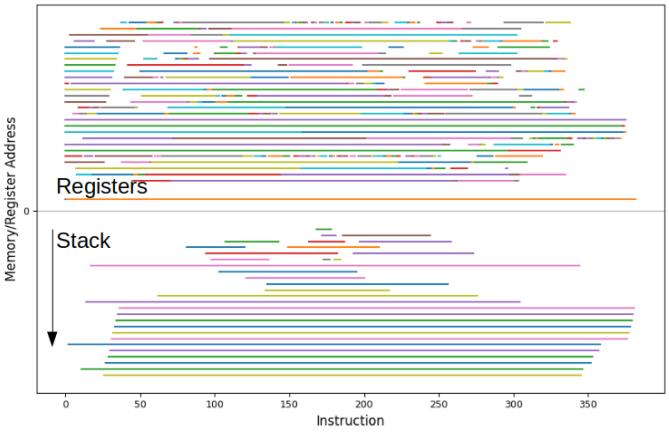


Fig. 6: Variable lifetime analysis over the registerfile and the stack for bitsliced 4-bit matrix multiplication

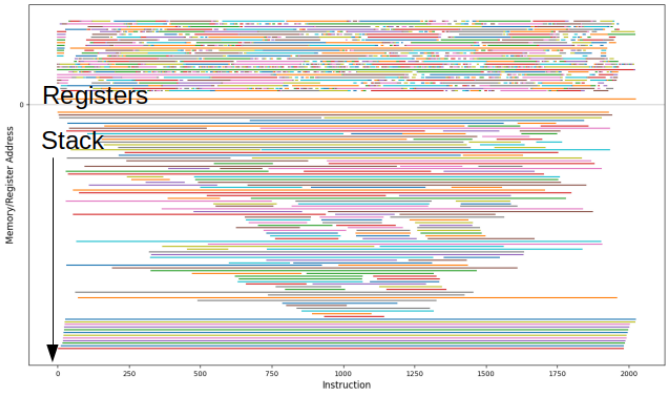


Fig. 7: Variable lifetime analysis over the registerfile and the stack for bitsliced 8-bit matrix multiplication

precision weights and activation using signed 8-bit integers. This quantization is performed for the matrix multiplication phase of NN layer in order to evaluate the bitsliced matrix multiplication of signed 8-bit precision weights matrix with 8-bit input activations. On comparison with 32-bit floating-point arithmetic, the evaluation demonstrated that there is an increase in model accuracy with reduced bit-precision as indicated by the accuracy corresponding to bitsliced and non-bitsliced integer-only arithmetic.

TABLE III: CLASSIFICATION ACCURACY OF NEURAL NETWORKS ON MNIST DATASET WITH BITSLICED MATRIX MULTIPLICATION

Neural Network Structure	Classification accuracy		
	32-bit floating-point matrix multiplication	8-bit integer non-bitsliced matrix multiplication	8-bit integer bitsliced matrix multiplication
784-32-32-10	93.38%	93.67%	93.67%
784-50-32-10	93.6%	94.12%	94.12%

VI. CONCLUSION

We demonstrated that bitslicing is a viable strategy to optimize the reduced precision needs in neural networking. The most important contribution of the bitslicing technique is that it removes the limitations of wordlength-specific instructions. On the other hand, we conclude that there is a significant amount of spilling created through bitslicing. Also, our current experiments have focused on the matrix multiplication. In our future work we plan to investigate the spilling overhead in further depth. We will extend bitslicing to the full neural network, and we plan to evaluate additional NN topologies and data-sets.

ACKNOWLEDGEMENTS

Support for this research was provided in part through National Science Foundation Award 1931639.

REFERENCES

[1] Zhenzhen Bao, Peng Luo, and Dongdai Lin. Bitsliced implementations of the PRINCE, LED and RECTANGLE block ciphers on AVR 8-bit microcontrollers. In *International Conference on Information and Communications Security*, pages 18–36. Springer, 2015.

[2] Eli Biham. A fast new DES implementation in software. In *International Workshop on Fast Software Encryption*, pages 260–272. Springer, 1997.

[3] Alessandro Capotondi, Manuele Rusci, Marco Fariselli, and Luca Benini. Cmix-nn: Mixed low-precision CNN library for memory-constrained edge devices. *IEEE Trans. Circuits Syst. II Express Briefs*, 67-II(5):871–875, 2020.

[4] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 609–622. IEEE Computer Society, 2014.

[5] Meghan Cowan, Thierry Moreau, Tianqi Chen, and Luis Ceze. Automating generation of low precision deep learning operators. *CoRR*, abs/1810.11066, 2018.

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

[6] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 567–597. Springer, 2017.
[7] Patrick Judd, Jorge Albericio, and Andreas Moshovos. Stripes: Bit-Serial Deep Neural Network Computing. *IEEE Computer Architecture Letters*, 16(1):80–83, January 2017.
[8] P. Kiaei and P. Schaumont. Synthesis of parallel synchronous software. *IEEE Embedded Systems Letters*, pages 1–1, 2020.
[9] Seiichi Matsuda and Shiho Moriai. Lightweight cryptography for the cloud: exploit the power of bitslice implementation. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 408–425. Springer, 2012.
[10] Darius Mercadier and Pierre-Évariste Dagand. Usuba: high-throughput and constant-time ciphers, by construction. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 157–173. ACM, 2019.
[11] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. Machine learning at the network edge: A survey, 2019.
[12] Conor Patrick, Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. Lightweight fault attack resistance in software using intra-instruction redundancy. In *International Conference on Selected Areas in Cryptography*, pages 231–244. Springer, 2016.
[13] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.
[14] Clifford Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.
[15] Jiecao Yu, Andrew Lukefahr, Reetuparna Das, and Scott A. Mahlke. Tf-net: Deploying sub-byte deep neural networks on microcontrollers. *ACM Trans. Embedded Comput. Syst.*, 18(5s):45:1–45:21, 2019.

978-1-7281-9219-2/20/$31.00 ©2020 IEEE