A Variable Bitwidth Asynchronous Dot Product Unit

Jonny Edwards[1], Adrian Wheeldon[2], Rishad Shafik[2], and Alex Yakovlev[2]

[1]Temporal Computing Ltd, Newcastle upon Tyne, UK
[2]μSystems Group, Newcastle University, Newcastle Upon Tyne, UK

Abstract— **We demonstrate a many operand asynchronous dot product with adjustable bit width for one of the multiplication tuple operands. The generalised algorithm called MADD relies on the commutative aspects of both the addition order and multiplication tuple operands. The approach is deeply wedded to asynchrony due to optimisation of the variable bitwidth memory architecture. Although specialised, this algorithm has a significant practical application in deep learning models particularly in managing the variable operand size due to the data variability, layering and optimisation process.**

I. Introduction and Prior Work

Dot products are the workhorse of many signal and data processing methods. Latterly they have gained prominence as the main distributed processing element within deep learning architectures, with much new accelerator design based around systolically organised Multiply Accumulate units [1]. It is worth noting that these designs tend to focus on memory localisation thus minimising the energy drain of moving large amounts of data to their processing. Specifically we treat a dot product as this:

$$S = \sum_{i}^{I} w_i a_i \qquad (1)$$

II. Asynchronous Dot Product

Under a simple scheme, a *variable number* of (w, a) tuple pairs can be added *arbitrarily* to the array at their w index point. A dot product on this multiplexed array can be evaluated efficiently and in a systolic way using the **Multiplicative ADD** (MADD) algorithm (Algorithm 1). Here (w, a) is mapped such that $a = \texttt{memory}[w]$.

The crucial asynchronous aspect of this work is to note that the iterating `While` loop is conditional on the `MAXVAL`, (which is established when the data is loaded into the memory structure) so the array *clock cycles are dependent on the size of the indexing variable*. Given this advantage, the operations cannot be clocked with a known cycle size. This necessitates an asynchronous control signal, which effectively notifies communicating circuits that the operation is finished.

A. MADD Algorithm: Hardware Implementation

A shift-register holds the array of multiplicands. The MADD algorithm processes one multiplicand per clock cycle and takes M clocks to compute the dot-product of M multiplicand-multiplier pairs. The multipliers are represented by the index positions of the multiplicands in the MADD array (starting with a multiplier of one). Although the multipliers are fixed constants, by using a large MADD array we can emulate multiply-accumulate of variables. For example, a MADD array of size 255 with 8-bit multiplicands can emulate an 8×8 variable multiply-accumulate.

Algorithm 1 Dot-product calculation with MADD

```
 1: procedure DOTPRODUCT(memory)
 2:     $\mathrm{acc}, \mathrm{height} \leftarrow 0$
 3:     $i \leftarrow MAXVAL$
 4:     while $i > 0$ do                ▷ Iterate over the whole memory
 5:         $\mathrm{height} \leftarrow \mathrm{height} + \mathrm{memory}[i]$
 6:         $\mathrm{acc} \leftarrow \mathrm{acc} + \mathrm{height}$
 7:         $i \leftarrow i - 1$
 8:     end while
 9:     $control \leftarrow 1$
10:     return acc                      ▷ Contains the dot product
11: end procedure
```

Critical path length can be reduced by using the output from the *height* register rather than taking the result from the first adder. However, this would result in one extra cycle to complete the calculation.

III. Comparison with existing MACs

Two additional designs were created for comparison with the MADD algorithm. Firstly, a Multiply Accumulator *MAC* consisting of a conventional multiplier with accumulator and secondly, a *MAC with memory* consisting of a conventional multiplier with accumulator, whereby the input operands are held in a memory array. Each design takes m clock cycles to compute the sum of m numbers of $a \times b$ tuples.

All designs were synthesized using Synopsys Design Compiler for a 90 nm low-power process. Results were recorded using post-synthesis simulation without performing layout.

Power figures obtained from this work in Table I use real switching rates derived by performing 10 random dot product computations – each consisting of eight terms. MADD algorithm offers decreased propagation delay at the same power point as the conventional MAC with memory, whilst computing in the same number of cycles.

Figure 1 shows power scaling based on the number of tuples summed for MADD and MAC with memory. As the number of tuples increases, so does the power, since more memory locations are being written to. As the number of tuples approaches 255 (the maximum possible with the memory sizes tested), the MADD algorithm becomes more power efficient.

Figure 2 shows how the average power of the MADD algorithm scales with bitness. Here the (w_n, x_n) tuples are kept square such that w_n and x_n are the same bitness. This is achieved by using a 2^n array size for an n-bit operand.

Work	Process (nm)	Size (bit)	Power (mW)	Delay (ns)
MAC	90	8	0.04	0.80
MAC w/mem	90	8	1.36	1.36
MADD	90	8	1.36	1.02
[2] mul	65	4	18	1
[3] MAC	180	8	1.9	2.34

TABLE I
COMPARISON OF DESIGNS FROM THE LITERATURE

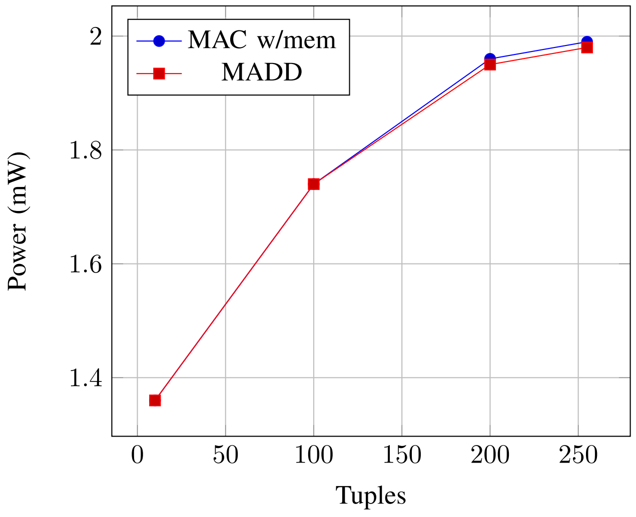

Fig. 1. Comparison of average power consumption versus number of tuples computed.

IV. CONCLUSIONS

The MADD algorithm's asynchronous index approach offers a novel and viable in-memory solution to the design of a dot product operator, with dynamically variable bitwidth for the indexing operand. The architectural use of asynchronous control is imperative in this design and has significant potential in the area of deep learning neural architectures. It is well known that bitwidth variations exists between neural models, within neuron models and even during training [4], and here, we offer a method that connects an asynchronous approach that can cope with this architectural inefficiency.

Our future aims are to explore the effects of this algorithm on large scale neural models, in particular during the training phase, to demonstrate increased efficiencies when the weight distributions become sparse.

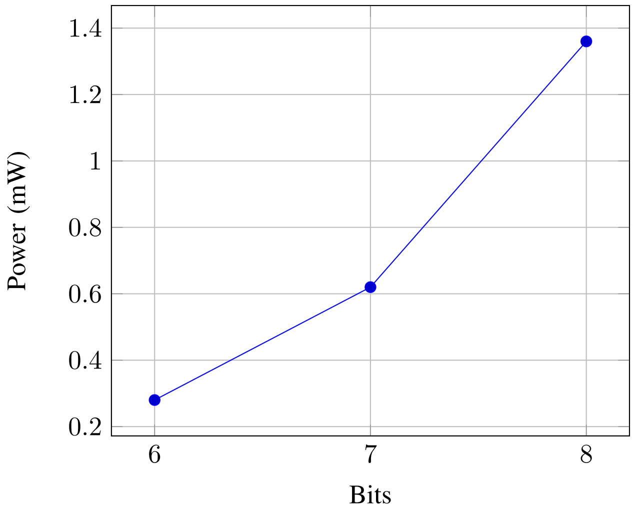

Fig. 2. MADD power scaling with bitness. n bit means n bit operands with a 2^n array size

REFERENCES

[1] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 1–12, ACM, 2017.
[2] Aurangozeb, A. D. Hossain, C. Ni, Q. Sharar, and M. Hossain, "Time-domain arithmetic logic unit with built-in interconnect," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, pp. 2828–2841, Oct 2017.
[3] S. Deepak and B. J. Kailath, "Optimized mac unit design," in *2012 IEEE International Conference on Electron Devices and Solid State Circuit (EDSSC)*, pp. 1–4, Dec 2012.
[4] E. Wang, J. J. Davis, R. Zhao, H. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where we've been, where we're going," *CoRR*, vol. abs/1901.06955, 2019.