Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

HOBFLOPS for CNNs: Hardware Optimized Bitslice-Parallel Floating-Point Operations for Convolutional Neural Networks

James Garland (jgarland@tcd.ie)
Trinity College Dublin: The University of Dublin Trinity College https://orcid.org/0000-0002-8688-9407
David Gregg
Trinity College Dublin: The University of Dublin Trinity College

Research Article

Keywords: neural network, floating point, FPGA, HOBFLOPS, CNN

Posted Date: September 2nd, 2021

DOI: https://doi.org/10.21203/rs.3.rs-866039/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Soft Computing for Edge-Driven Applications manuscript No.
(will be inserted by the editor)

HOBFLOPS for CNNs: Hardware Optimized Bitslice-Parallel Floating-Point Operations for Convolutional Neural Networks

James Garland · David Gregg

Received: date / Accepted: date

Abstract Low-precision floating-point (FP) can be highly effective for convolutional neural network (CNN) inference. Custom low-precision FP can be implemented in field programmable gate array (FPGA) and application-specific integrated circuit (ASIC) accelerators, but existing microprocessors do not generally support fast, custom precision FP. We propose hardware optimized bitslice-parallel floating-point operators (HOBFLOPS), a generator of efficient custom-precision emulated bitslice-parallel software (C/C++) FP arithmetic. We generate custom-precision FP routines, optimized using a hardware synthesis design flow, to create circuits. We provide standard cell libraries matching the bitwise operations on the target microprocessor architecture and a code-generator to translate the hardware circuits to bitslice software equivalents. We exploit bitslice parallelism to create a novel, very wide (32—512 element) vectorized CNN convolution for inference. On Arm and Intel processors, the multiply-accumulate (MAC) performance in CNN convolution of HOBFLOPS, Flexfloat, and Berkeley's SoftFP are compared. HOBFLOPS outperforms Flexfloat by up to 10× on Intel AVX512. HOBFLOPS offers arbitrary-precision FP with custom range and precision, *e.g.*, HOBFLOPS9, which outperforms Flexfloat 9-bit on Arm Neon by 7×. HOBFLOPS allows researchers to prototype different levels of custom FP precision in the arithmetic of software CNN accelerators. Furthermore, HOBFLOPS fast custom-precision FP CNNs may be valuable in cases where memory bandwidth is limited.

This research is supported by Science Foundation Ireland, Project 12/IA/1381. We thank the Institute of Technology Carlow, Carlow, Ireland for their support.

J. Garland
School of Computer Science and Statistics, Trinity College Dublin, Dublin, Ireland
E-mail: jgarland@tcd.ie

David Gregg
School of Computer Science and Statistics, Trinity College Dublin, Dublin, Ireland
E-mail: david.gregg@cs.tcd.ie

1 Introduction

Many researchers have shown that CNN inference is possible with low-precision integer [18] and floating-point (FP) [5,9] arithmetic. Almost all processors provide support for 8-bit integers, but not for bit-level custom-precision FP types, such as 9-bit FP. Typically processors support a small number of relatively high-precision FP types, such as 32- and 64-bit [16]. However, there are good reasons why we might want to implement custom-precision FP on regular processors. Researchers and hardware developers may want to prototype different levels of custom FP precision that might be used for arithmetic in CNN accelerators [25,12,7]. Furthermore, fast custom-precision FP CNNs in software may be valuable, particularly in cases where memory bandwidth is limited.

To address custom-fp in central processing units (CPUs), FP simulators, such as the Flexfloat [23], and Berkeley's SoftFP [13], are available. These simulators support arbitrary or custom range and precision FP such as 16-, 32-, 64-, 80- and 128-bit with corresponding fixed-width mantissa and exponents respectively. However, the simulators' computational performance may not be sufficient for the requirements of high throughput, low latency arithmetic systems such as CNN convolution.

We propose hardware optimized bitslice-parallel floating-point operators (HOBFLOPS). HOBFLOPS generates FP units, using software *bitslice parallel* arithmetic, efficiently emulating FP arithmetic at arbitrary mantissa and exponent bit-widths. We exploit bit-slice parallelism techniques to pack the single instruction multiple data (SIMD) vector registers of the microprocessor efficiently. Also, we exploit

bitwise logic optimization strategies of a commercial hardware synthesis tool to optimize the associated bitwise arithmetic. A source-to-source generator converts the netlists to the target processor's bitwise SIMD vector operators.

To evaluate performance we benchmark HOBFLOPS8 through to HOBFLOPS16e parallel MACs against arbitrary-precision Flexfloat, and 16- and 32-bit Berkeley's SoftFP, implemented in CNN convolution with Arm and Intel scalar and vector bitwise instructions. We show HOBFLOPS offers significant performance boosts compared to Flexfloat, and SoftFP. We show that our software bitslice parallel FP is both more efficient and offers greater bit-level customizability than other software FP emulators.

We make the following contributions:

- We present a full design flow from a VHDL FP core generator to arbitrary-precision software bitslice parallel FP operators. HOBFLOPS are optimized using hardware design tools and logic cell libraries, and domain-specific code generator.
- We demonstrate how 3-input Arm NEON bitwise instructions *e.g.*, SEL (multiplexer), and AVX512 bitwise ternary operations are used in standard cell libraries to improve the efficiency of the generated code.
- We present an algorithm for implementing CNN convolution with the very wide vectors that arise in bitslice parallel vector arithmetic.
- We evaluate HOBFLOPS on Arm Neon and Intel AVX2 and AVX512 processors. We find HOBFLOPS achieves approximately 3×, 5×, and 10× the performance of Flexfloat respectively.
- We evaluate various widths of HOBFLOPS from HOBFLOPS8–HOBFLOPS16e. We find *e.g.*, HOBFLOPS9 outperforms Flexfloat 9-bit by 7× on Intel AVX512, by 3× on Intel AVX2, and around 2× Arm Neon. The increased performance is due to:
  - Bitslice parallelism of the very wide vectorization of the MACs of the CNN;
  - Our efficient code generation flow.

The rest of this article is organized as follows. Section 2 highlights our motivation and gives background on other CNN accelerators use of low-precision arithmetic types. Section 3 outlines bitslice parallel operations and introduces HOBFLOPS, shows the design flow, types supported and how to implement arbitrary-precision HOBFLOPS FP arithmetic in a convolution layer of a CNN. Section 4 shows significant increases in performance of HOBFLOPS8–HOBFLOPS16e compared with Flexfloat, and SoftFP on Intel's AVX2 and AVX512 and Arm Neon processors. We conclude with Section 5.

2 Background and Motivation

Arbitrary precision floating-point computation is largely unavailable in CPUs. Soft FP simulation is available but typically lacks the computation performance required of low latency applications. Researchers often reduce the precision to a defined fixed-point basis for CNN inference, potentially impacting CNN classification accuracy [20].

Reduced-precision CNN inference, particularly CNN weight data, reduces computational requirements due to memory accesses, which dominate energy consumption. Energy and area costs are also reduced in ASICs and FPGAs [22].

Johnson [17] suggests that little effort has been made in improving FP efficiency so proposes an alternative floating-point representation. They show that a 16-bit log float multiply-add is 0.68× the integrated circuit (IC) die area compared with an IEEE-754 float16 fused multiply-add, while maintaining the same significand precision and dynamic range. They also show that their reduced FP bit precision compared to float16 exhibits 5× power saving. We investigate if similar efficiencies can be mapped into software using a hardware tool optimization flow.

Kang *et al.*, [19] investigate short, reduced FP representations that do not support not-a-numbers (NANs) and infinities. They show that shortening the width of the exponent and mantissa reduces the computational complexity within the multiplier logic. They compare fixed point integer representations with varying widths up to 8-bits of their short FP in various CNNs, and show around a 1% drop in classification accuracy, with more than 60% reduction in ASIC implementation area. Their work stops at the byte boundary, leaving us to investigate other arbitrary ranges.

For their Project Brainwave [5,9], Microsoft proposes MS-FP8 and MS-FP9, which are 8-bit and 9-bit FP arithmetic that they exploit in a quantized CNN [5,9]. Microsoft alters the Minifloat 8-bit that follows the IEEE-754 specification (1-sign bit, 4-exponent bits, 3-mantissa bits) [15]. They create MS-FP8, of 1-sign bit, 5-exponent bits, and 2-mantissa bits. MS-FP8 gives a larger representative range due to the extra exponent bit but lower precision than Minifloat, caused by the reduced mantissa. MS-FP8 more than doubles the performance compared to 8-bit integer operations, with negligible accuracy loss compared to full float. To improve the precision, they propose MS-FP9, which increases the mantissa to 3 bits and keeps the exponent at 5 bits. Their later work [9] uses a shared exponent with their proposed MS-FP8 / MS-FP9, *i.e.*, one exponent pair used for many mantissae, sharing the reduced mantissa multipliers. We do not investigate shared exponent. Their work remains at 8- and 9-bit for FPGA implementation and leaves us to investigate other bit-precision and ranges.

Rzayev *et al.'s*, Deep Recon work [21] analyzes the computation costs of deep neural networks (DNNs). They

propose a reconfigurable architecture to efficiently utilize computation and storage resources, thus allowing DNN acceleration. They pay particular attention to comparing the prediction error of three CNNs with different fixed and FP precision. They demonstrate that FP precision on the three CNNs is 1-bit more efficient than fixed bit-width. They also show that the 8-bit FP is around 7× more energy-efficient and approximately 6× more area efficient than 8-bit fixed precision. We further the area efficiency investigation.

Existing methods of emulating FP arithmetic in software primarily use existing integer instructions to implement the FP computation steps. This works well for large, regular-sized FP types such as FP16, FP32, or FP64. Berkeley offer SoftFP emulation [13] for use where, *e.g.*, only integer precision instructions are available. SoftFP emulation supports 16- to 128-bit arithmetic and does not support low bit-width custom precision FP arithmetic or parallel arithmetic.

The Flexfloat C++ library of Tagliavini *et al.*, [23], offers alternative FP formats with variable bit-width mantissa and exponents. They demonstrate Flexfloat is up to 2.8× and 2.4× faster than MPFR and SoftFloat, respectively for various benchmarks. They do not explore arbitrary bit-precision or other optimization techniques on the proposed number formats, which our work does. Both Flexfloat and SoftFP simulators operate at a much lower performance than that of the hardware floating-point unit (FPU).

Other researchers investigate optimizing different representations of FP arithmetic. Xu *et al.*, [24] propose bitslice parallel arithmetic and present FP calculations undertaken on a fixed point unit. Instead of storing vectors in the traditional sense of storing 17-bit vectors inefficiently in a 32-bit register, they instead store thirty-two 17-bit words transformed into bitslice parallel format. Xu *et al.*, manually construct bitwise arithmetic routines to perform integer or FP arithmetic, while the vectors remain in a bitslice parallel format. When coded in C/C++ and AVX2 SIMD instructions, they demonstrate this approach is efficient for low-precision vectors, such as 9-bit or 11-bit arbitrary FP types. We investigate beyond 11-bit and AVX2 implementation by generating and optimizing the process using ASIC design flow tools and applying HOBFLOPS to CNN convolution on Arm Neon, Intel AVX2 and AVX512.

Researchers investigate different bit precision and representations of the FP number base. Google's tensor processing unit (TPU) ASIC [18] implements brain floating point 16-bit (bfloat16) [11], a 16-bit truncated IEEE-754 FP single-precision format. Bfloat16 preserves dynamic range of the 32-bit format due to the 8-bit exponent. The precision is reduced in the mantissa from IEEE's 24-bits down to 7-bits.

Nvidia has proposed the new tensor float 32 (TP32) number representation [4], which is a sign bit, 8-bit exponent and 10-bit mantissa. This 19-bit floating-point format is an input format which truncates the IEEE FP32 mantissa to 10-bits. TP32 still produces 32-bit floating-point results to move and store in the graphics processor unit (GPU). Similar to Google, Nvidia does not investigate other bit-widths of FP range and precision, something our work demonstrates.

```
1 flopoco FPMult pipeline=0 useHardAdd=0
      frequency=1 plainVHDL=1 target=Virtex6 wE
      =5 wF=2 wFOut=3 outputFile=FPMult_5_2_3.
      vhd
2 flopoco FPMult pipeline=0 useHardAdd=0
      frequency=1 plainVHDL=1 target=Virtex6 wE
      =5 wF=2 wFOut=5 outputFile=FPMult_5_2_5.
      vhd
3 flopoco FPAdd  pipeline=0 useHardAdd=0
      frequency=1 plainVHDL=1 target=Virtex6 wE
      =5 wF=3 outputFile=fpAdd_5_3.vhd
4 flopoco FPAdd  pipeline=0 useHardAdd=0
      frequency=1 plainVHDL=1 target=Virtex6 wE
      =5 wF=5 outputFile=fpAdd_5_5.vhd
```

Listing 1 Example FloPoCo Script for standard and extended precision Multipliers and Adders.

3 Approach

In this section, we present how to produce HOBFLOPS arithmetic units. HOBFLOPS generates efficient software emulation parallel FP arithmetic units optimized using hardware synthesis tools. HOBFLOPS investigates reduced complexity FP, [19,17] that is more efficient than fixed-point [21]. HOBFLOPS considers alternative FP formats [23] and register packing with bit-sliced arithmetic [24]. We demonstrate HOBFLOPS in a CNN convolution layer.

Figure 1 outlines our flow for creating HOBFLOPS arithmetic units. We generate the register transfer level (RTL) representations of arbitrary-precision FP multipliers and adders using the FP unit generator, FloPoCo [6]. We configure the FP range and precision of HOBFLOPS adders and multipliers, Listing 1. Note the option *frequency=1* is set so FloPoCo relaxes timing constraints to produce the smallest asynchronous gate count design. The low number of gates will 1-1 map to bitwise logic SIMD vector instructions.

We produce standard cell libraries of logic gate cells mapped to bitwise logic instructions of the target microprocessor architecture. For example, AND, OR, XOR, the SEL (multiplexer) bitwise instructions are supported on Arm Neon. The 256-ternary logic look up table (LUT) bitwise instructions of the Intel AVX512 are supported. NOTE: due to the lack of availability of Arm Scalable Vector Extension (SVE) devices, we show results for an Arm Neon device. The same approach can be applied to creating a cell library for an Arm SVE device or other SIMD vector instruction processor, when available.

We use the ASIC tool, Cadence Genus with our standard cell libraries, Listing 2, and small gate area optimization script, Listing 3. Genus synthesizes the adders and multi-

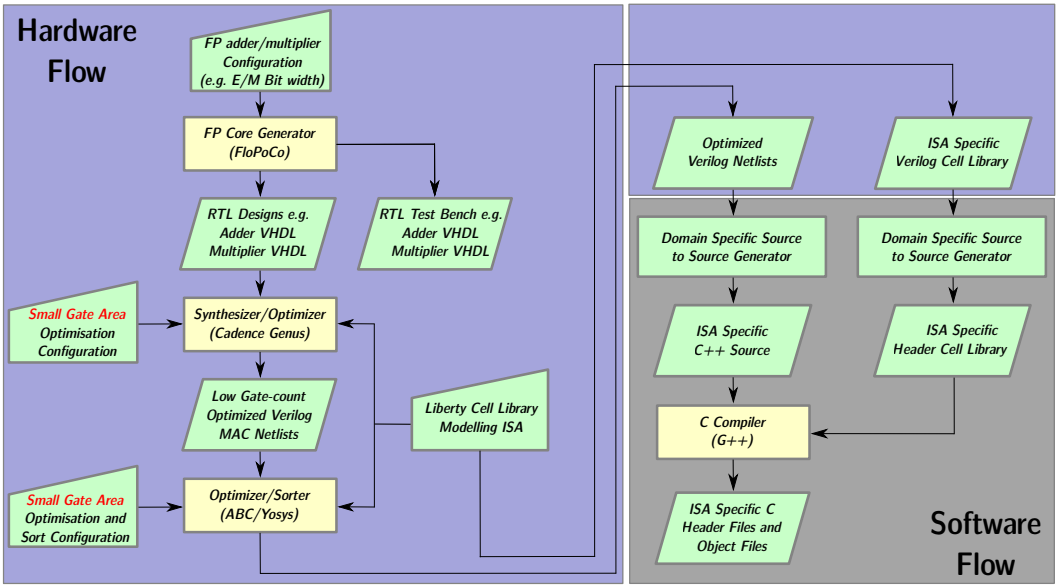


Fig. 1 Flow for Creating HOBFLOPS Bitwise Intrinsic Operations. Yellow signifies third party tools. We start in the hardware domain and cross into the software domain.

```
library(avx512) {
cell (LUT000X1) {
area : 1.0;
  cell_leakage_power : 0.0;
  pin(Y)  {
    direction : output;
    capacitance : 0;
    rise_capacitance : 0;
    fall_capacitance : 0;
    max_capacitance : 0;
    function : "0";
  }
}
}
```

Listing 2 Snippet Showing One Cell of AVX512 Standard Cell Library.

```
proc runFlow {}  {
  read_libs cellLib/avx.lib
  read_hdl -library work -f filesList.txt
  set top FPMult_5_2_5_2_5_3
  elaborate
  ungroup -all -flatten -force
  syn_generic -effort high
  syn_map -effort high
  syn_opt -effort high
  set netsList [get_nets]
  sort_collection $netsList name
  writeReps reports netlists $top
}
proc writeReps {_repsDir_ _netsDir_ _top_} {
  report gates >  ${_repsDir_}/${_top_}
    _gates.rpt
  report area [get_designs $_top_] > ${
    _repsDir_}/${_top_}_area.rpt
  write_hdl -mapped > ${_netsDir_}/$_top_.v
  write_sdc > ${_netsDir_}/constraints.sdc
}
file mkdir reports
file mkdir netlists
runFlow
```

Listing 3 Example Cadence Genus Script.

pliers into Verilog netlists. The synthesis and technology mapping suite, Yosys ASIC [3] and ABC optimizer [2], Listing 4, allows further optimization and topologically sort of the netlists with the same cell libraries.

Our custom domain-specific source-to-source generator, Listing 5, converts the topologically sorted netlists into a C/C++ header of bitwise operations. In parallel, the generator converts the standard cell libraries into equivalent C/C++ cell library headers of SIMD vector extension instructions of the target processor instruction set architecture (ISA).

We create a CNN convolution layer to include the HOBFLOPS adder, multiplier and cell library headers corresponding to the target ISA, e.g., Listing 6, and compile with G++.

3.1 HOBFLOPS Cell Libraries for Arm Neon, Intel AVX2 and AVX512

We create three Liberty standard cell libraries mapping hardware bitwise representations to Arm Neon Intel X86_64, AVX, AVX2 and AVX512 SIMD vector intrinsics. The Arm

Table 1 HOBFLOPS Cell Libraries' Support for Arm Neon and Intel Intrinsic Bitwise Logic Operations.

Arm (64-bit)		Arm Neon (128-bit) [1]		Intel (64-bit)		Intel AVX2 (128-, 256-bit)		Intel AVX512 (512-bit) [16]	
AND	A & B	AND	A & B	AND	A & B	AND	A & B	LUT000	0
OR	A | B	OR	A | B	OR	A | B	OR	A | B	LUT001	(A | (B | C)) ^ 1
XOR	A ^ B	XOR	A ^ B	XOR	A ^ B	XOR	A ^ B	LUT002	~ (B | A) C
NOT	~A	NOT	~A	NOT	~A	NOT	~A	LUT003	(B | A) ^ 1
ORN	A & (~B)	ORN	A | ~B			ANDNOT	~A & B	LUT004	~(A | C) B
		SEL	(~((S & A) | (~S & B)))					LUT005	(C | A) ^1
								...	... *(truncated)*
								LUT253	A | (B | (C ^ 1))
								LUT254	A | (B | C)
								LUT255	1

```
read_liberty -lib ../cellLib/avx2.lib
read_verilog FPMult_5_2_5_2_5_3_comb_uid2.v
hierarchy -check -top
    FPMult_5_2_5_2_5_3_comb_uid2
synth -top FPMult_5_2_5_2_5_3_comb_uid2
abc -liberty ../cellLib/avx2.lib
proc; opt; flatten; proc; opt; pmuxtree; opt;
    memory -nomap; opt; clean;
techmap; opt
clean
torder -noautostop
write_verilog
    FPMult_5_2_5_2_5_3_comb_uid2Topo.v
```

Listing 4 Example Yosys-ABC script.

```
cd $1/netlists
for i in $(find . -name "*.v");
do
  TOP_MODULE=$(basename $i .v)
  cp template.h ${TOP_MODULE}.h
  sed -i "s/TEMPLATE_H_/${TOP_MODULE}_H_/g" ${
    TOP_MODULE}.h
  sed -i "s/void template/void ${TOP_MODULE}/g
    " ${TOP_MODULE}.h
  sed -e '1,11d' < ${TOP_MODULE}.v > ${TOP_
    MODULE}_1.h
  sed -i 's/wire/BSFP_T/g' ${TOP_MODULE}_1.h
  sed -i "s/.A (/\&/g" ${TOP_MODULE}_1.h
  sed -i "s/.B (/\&/g" ${TOP_MODULE}_1.h
  sed -i "s/.C (/\&/g" ${TOP_MODULE}_1.h
  sed -i "s/.Y (/\&/g" ${TOP_MODULE}_1.h
  sed -i "s/) ,/,/g" ${TOP_MODULE}_1.h
  sed -i "s/));/);/g" ${TOP_MODULE}_1.h
  sed -i "s/ g.*(/(/g" ${TOP_MODULE}_1.h
  sed -i "s/endmodule/\}/" ${TOP_MODULE}_1.h
  sed -e "\$a#endif  \/\/ ${TOP_MODULE}_H_" ${
    TOP_MODULE}_1.h > ${TOP_MODULE}_2.h
  cat ${TOP_MODULE}.h ${TOP_MODULE}_2.h > ${
    TOP_MODULE}_3.h
  mv ${TOP_MODULE}_3.h ${TOP_MODULE}.h
  rm ${TOP_MODULE}_1.h ${TOP_MODULE}_2.h
done
```

Listing 5 Example Source-to-Source Generator.

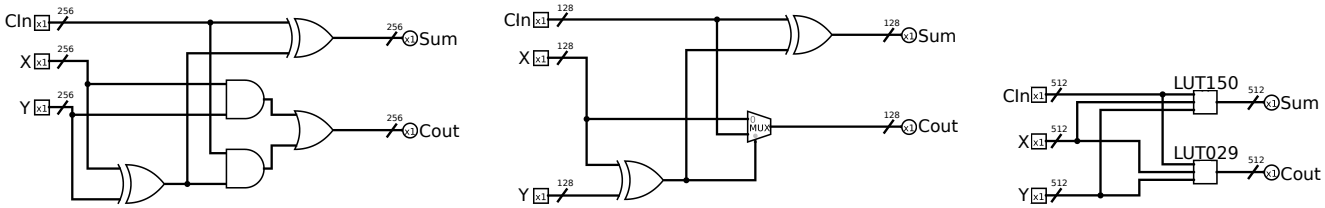
```
void AND2X1(u256 *A, u256 *B, u256 *Y) {
  *Y = _mm256_and_si256(*A, *B);}
void NOTX1(u256 *A, u256 *Y) {
  // Inverter could be implemented in many
    ways
  *Y = _mm256_xor_si256(*A, _mm256_set1_epi32
    (-1));}
void OR2X1(u256 *A, u256 *B, u256 *Y) {
  *Y = _mm256_or_si256(*A, *B);}
void XOR2X1(u256 *A, u256 *B, u256 *Y) {
  *Y = _mm256_xor_si256(*A, *B);}
void ANDNOT2X1(u256 *A, u256 *B, u256 *Y) {
  *Y = _mm256_andnot_si256(*A, *B);}
```

Listing 6 Macros for AVX2 Cell Library Bitwise Operator Definitions

Neon SEL (multiplexer) bitwise multiplexer instruction is a 3-input bitwise logic instruction, whereas all other Neon bitwise logic instructions modeled in the cell library are 2-input. The ternary logic LUT of the Intel AVX512 is a 3-input bitwise logic instruction that can implement 3-input boolean functions. An 8-bit immediate operand to this instruction specifies which of the 256 3-input functions should be used. We create all 256 equivalent cells in the Liberty cell library when targeting AVX512 devices. Table 1 lists the bitwise operations supported in the cell libraries for each architecture. The AVX512 column shows a truncated example subset of the available 256 bitwise logic instructions [16] in both hardware and software cell libraries.

To demonstrate the capabilities of the cell libraries, we show an example of a full adder. Figure 2a shows a typical 5-gate full adder implemented with our AVX2 cell library.

The same full adder can be implemented in three Arm Neon bitwise logic instructions, Figure 2b, one of which is the SEL bitwise multiplexer instruction. Intel AVX512 intrinsics can implement the full adder in two 3-input bitwise ternary instructions, Figure 2c. While the inputs and outputs of the hardware gate level circuits are single bits, these inputs and outputs are parallelized by the bit-width of the SIMD vector registers. The bitwise instructions, converted



(a) Intel AVX2 Bitwise Operations. (b) Arm Neon Bitwise Operations. (c) Intel AVX512 3-Input LUT Bitwise Operations.

Fig. 2 Full Adders Implemented in Intel's AVX2 and AVX512 and Arm's Neon Intrinsic Bitwise Operations.

Table 2 Bit Width Comparisons of Existing Custom FP.

Sign	Exponent	Mantissa	Type, Availability, Performance
1	4	3	IEEE-FP8 (Slow in S/W) [15]
1	5	2	MS-FP8 (Fast in H/W) [5,9]
1	5	3	MS-FP9 (Fast in H/W) [5,9]
1	5	10	SoftFP16 (Slow in S/W) [13]
1	8	23	SoftFP32 (Slow in S/W) [13]
1	Arbitrary	Arbitrary	Flexfloat (Slow in S/W) [23]
1	Arbitrary	Arbitrary	HOBFLOPS (Fast in S/W) **(Ours)**

to bitwise software operations, produce extensive parallel scaling of the arithmetic.

3.2 HOBFLOPS Bitslice Parallel Operations

HOBFLOPS exploits bitslice parallel operations to represent FP numbers in a bitwise manner that are processed in parallel. For example, many 9-bit values are transformed to bitslice parallel representations, see Figure 3a. A simple example of how nine registers of 512-bit bitslice parallel data are applied to a 512-bit wide bitslice parallel FP adder is shown in Figure 3b. Each adder instruction has a throughput of between a 1/3 and 1 clock cycle [16,1]. In this example, the adder's propagation delay is related to the number of instruction-level parallelism and associated load/-store commands. The number of gates and thus SIMD vector instructions in the HOBFLOPS adder or multiplier is dependent on the required HOBFLOPS precision. See Table 3 for examples of HOBFLOPS MAC precision, and Section 4 for associated HOBFLOPS MAC SIMD vector instruction counts and performance.

3.3 HOBFLOPS Design Flow

Taking inspiration from Microsoft's MS-FP8 / MS-FP9 [5,9] and Minifloat 8-bit that follows the IEEE-754 2019 specification, we create single- and extended-precision HOBFLOPS adders and multipliers. For example, we create the single-precision HOBFLOPS8 multiplier to take two 5-bit exponent, 2-bit mantissa, 1-bit sign inputs and produce a single 5-bit exponent and 3-bit mantissa and 1-bit sign output. We also create extended-precision HOBFLOPS8e multiplier to take two 5-bit exponent, 2-bit mantissa, and 1-bit sign to produce a single 5-bit exponent, 5-bit extended mantissa and 1-bit sign output.

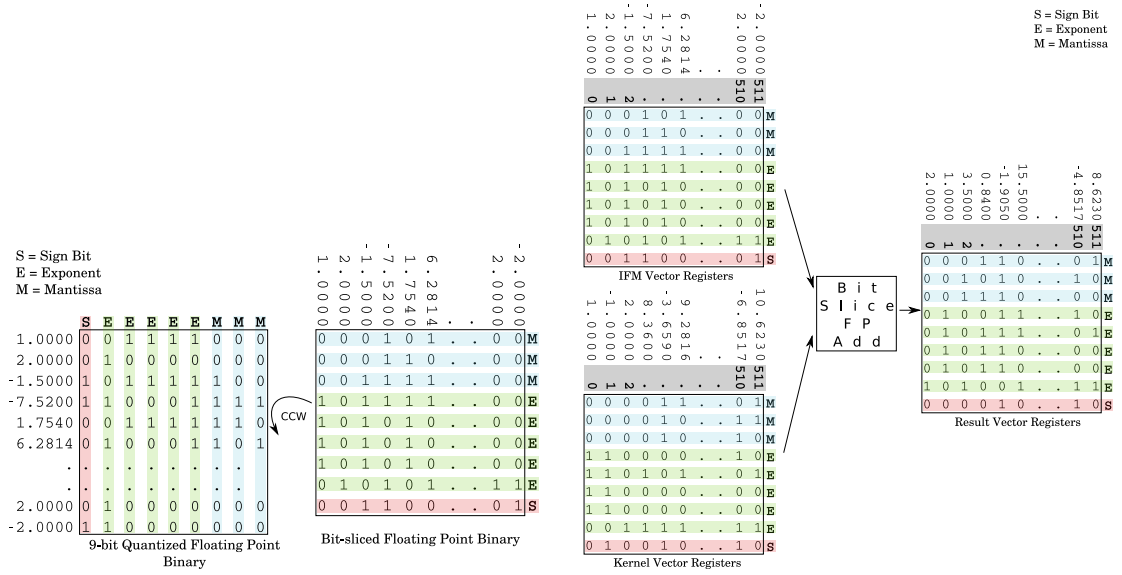
When using HOBFLOPS, any quantization method may be employed, such as those investigated by Gong *et al.*, [10]. These quantized values are stored and computed during inference mode. Therefore we set FloPoCo to the required range and precision. For comparison, Table 2 shows the range and precision of IEEE-FP8 of MS-FP8 and MS-FP9, respectively, available in simulated software and FPGA hardware. These FP solutions only support specific ranges and do not allow for arbitrary exponent and mantissa values.

Details of the evaluated HOBFLOPS are in Table 3, although, other arbitrary combinations of mantissa and exponent bit-widths are supported in the flow (see Figure 1).

FloPoCo [6], generates VHDL descriptions of FP adders and multipliers of varying exponent and mantissa bit-widths for standard and extended precision HOBFLOPS types, see Table 3. As a performance baseline, we produce the 16- and 16e-bit IEEE-754 FP versions of the multiplier and adder, for comparison against Flexfloat and SoftFP. FloPoCo automatically produces the corresponding VHDL test benches and test vectors required to test the generated cores. FloPoCo has a slightly different way of encoding the FP numbers when compared to the IEEE-754 2019 specification and does not support subnormal numbers. A FloPoCo FP number [6] is a bit-vector consisting of the following 4 fields: 2-bit exception field (*01* for normal numbers); a sign bit; an exponent field *wE* bits wide; a mantissa (fractional) field *wF* bits wide. The significand has an implicit leading *1*, so the fraction field *ff...ff* represents the significand *1.ff...ff*.

We configure FloPoCo to generate combinatorial plain RTL VHDL cores with a 1MHz frequency, no pipelining, and no use of hard-macro FPGA multipliers or adders, Listing 1. These settings ensure that FloPoCo generates reduced area rather than reduced latency multipliers and adders. We simulate the FP multipliers and adders in a VHDL simulator with the corresponding FloPoCo generated test bench to confirm that the quantized functionality is equivalent to IEEE-754 FP multiplier and adder.

We create Synopsys Liberty standard cell libraries to support the target processor architecture, e.g., Listing 6. Ca-



(a) Many 9-bit FP Values Transformed to 9-bit bitslice parallel FP Data Layout.

(b) 512 9-bit Bitslice Parallel FP Add Operation Using AVX512 Registers.

Fig. 3 Bit-sliced Parallel FP Transformation and Bit-sliced FP Add Operation.

Table 3 HOBFLOPS MAC Standard and Extended Range and Precision Types

hobflops(IEEE)*XX*	**Inputs Bit Width**		**Outputs Bit Width**		**hobflops(IEEE)*XXe***	**Inputs Bit Width**		**Outputs Bit Width**	
	Expo	**Mant**	**Expo**	**Mant**		**Expo**	**Mant**	**Expo**	**Mant**
HOBFLOPSIEEE8	4	3	4	4	HOBFLOPSIEEE8e	4	3	4	7
HOBFLOPS8	5	2	5	3	HOBFLOPS8e	5	2	5	5
HOBFLOPS9	5	3	5	4	HOBFLOPS9e	5	3	5	7
... (truncated)	...	...	...	...	***... (truncated)***	...	...	...	...
HOBFLOPS16	5	10	5	11	HOBFLOPS16e	5	10	5	21

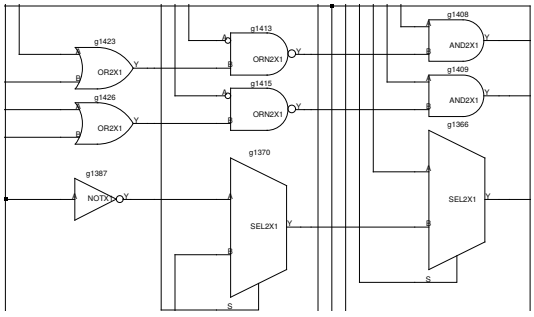


Fig. 4 Zoomed Snippet of An Example HOBFLOPS8 Multiplier Netlist Using Arm Neon Cells.

dence Genus (version 16.22-s033_1) the industry-standard ASIC synthesis tool, synthesizes the adder and multiplier VHDL cores with our standard cell libraries and configuration and small gate area optimization script, Listing 3, into a Verilog netlist of the logic gates. See Figure 4 for an example of the HOBFLOPS8 multiplier logic produced by FloPoCo when synthesized with our Arm Neon cell Library. Note how Genus has synthesized the design to include the 3-input SEL gate (multiplexer) supported by Arm Neon.

HOBFLOPS designs are combinatorial, so synthesis timing constraints are unnecessary. In the standard cell libraries Liberty file, we assign a value of 1.0 to *cell area* and *cell leakage power* of the cells. We configure the cell capacitance and timing values to zero. These values ensure the synthesizer assigns equal optimization priority to all gates and produces a netlist with the least number of logic gates rather than creating a netlist optimized for hardware timing propagation.

We further optimize the netlist using the open-source Yosys ASIC synthesizer [3] and ABC optimizer [2]. We use ABC's *strash* command to transform the current network into an AND-inverter graph (AIG) by one-level structural hash-

ing. We then use the *refactor* function to iteratively collapse and refactor the levels of logic and area of the netlist. We configure Yosys to produce a topologically sorted Verilog netlist of gates, Listing 4. The topological sorting is required as Cadence Genus writes the netlist file in an output port to input port order, whereas the C/C++ compiler requires the converted netlist to have input to output ordering. We formally verify the topologically sorted netlist against the original netlist with Yosys satisfiability (SAT)-solver. These netlists are re-simulated with the test bench used to simulate the FloPoCo generated VHDL designs and compared for correlation.

ALGORITHM 1: HOBFLOPS Code for a Simple AVX2 2-bit Binary Full Adder (unrolled by Synthesis) - 256 wide 2-bit adders in 12 Bitwise Operations.

```
Input: x[2] of SIMD width
Input: y[2] of SIMD width
Input: cin of SIMD width
Output: HOBFLOPS register sum[2] of SIMD width
Output: HOBFLOPS register cout of SIMD width
XOR2X1(x[1], y[1], n_5); // Wide XOR Operation
OR2X1(x[1], y[1], n_2); // Wide OR Operation
OR2X1(cin, x[0], n_1);
AND2X1(x[1], y[1], n_0); // Wide AND Operation
AND2X1(cin, x[0], n_3);
AND2X1(y[0], n_1, n_6);
OR2X1(n_3, n_6, n_8);
AND2X1(n_2, n_8, n_9);
OR2X1(n_0, n_9, cout);
XOR2X1(n_5, n_8, sum[1]);
XOR2X1(x[0], y[0], n_4);
XOR2X1(cin, n_4, sum[0]);
```

ALGORITHM 2: HOBFLOPS Code for a Simple AVX512 2-bit Binary Full Adder - 512 wide 2-bit adders in 4 Bitwise Operations.

```
Input: x[2] of SIMD width
Input: y[2] of SIMD width
Input: cin of SIMD width
Output: HOBFLOPS register sum[2] of SIMD width
Output: HOBFLOPS register cout of SIMD width
LUT232X1(cin, y[0], x[0], n_1); // (B&C)|A&(B ∧ C)
LUT232X1(x[1], n_1, y[1], cout);
LUT150X1(y[1], x[1], n_1, sum[1]); // A ∧ B ∧ C
LUT150X1(y[0], x[0], cin, sum[0]);
```

Our domain-specific source-to-source generator, Listing 5, translates the Verilog adder and multiplier netlists to Intel AVX2, AVX512, or Arm Neon bitwise operators with the correct types and pointers. Algorithm 1 shows the HOBFLOPS code for a simple 2-bit binary adder with carry, generated from 12 bitwise operations, referenced from the cell library of Listing 6. A single input bit of the hardware multiplier becomes the corresponding architecture variable type *e.g.*, uint64 for a 64-bit processor, an __mm256i type for an AVX2 processor, an __mm512i type for an AVX512 processor, uint32x4_t type for a Neon processor (see also Figure 2). Algorithm 2 demonstrates the efficiency of the AVX512 implementation of the same simple 2-bit adder, generated from 4 bitwise operations.

A HOBFLOPS8 multiplier targeted at the *e.g.*, AVX2 processor is generated in 80 bitwise operations, and a HOBFLOPS8 adder is generated in 319 bitwise operations. When targeted at the AVX512 processor, the HOBFLOPS8 multiplier is generated in 53 bitwise operations, and the HOBFLOPS8 adder is generated in 204 bitwise operations.

3.4 CNN Convolution with HOBFLOPS

We present a method for CNN convolution with HOBFLOPS arithmetic. We implement HOBFLOPS MACs in a CNN convolution layer, where up to 90% of the computation time is spent in a CNN [8]. We compare HOBFLOPS MAC performance to IEEE FP MAC and to Flexfloat [23] and SoftFP [13].

Figure 5 shows HOBFLOPS IFM and kernel values convolved and stored in the OFM. To reduce cache misses, we tile the IFM $H \times W \times C$ dimensions, which for *Conv dw / s2* of MobileNets CNN is $14 \times 14 \times 512 = 100,352$ elements of HOBFLOPS IFM values. We tile the M kernel values by $LANES \times NINBITS$, where $LANES$ corresponds to the target architecture registers bit-width, *e.g.*, 512-lanes corresponds to AVX512 512-bit wide register. The $LANES \times NINBITS$ tiles of binary values are transformed to $NINBITS$ of *SIMD width* values, where *SIMD width* correspond to the target architecture register width, *e.g.*, uint64 type for a 64-bit processor architecture, __mm512i type for AVX512 processor.

We broadcast the IFM channel tile of $NINBITS$ across the corresponding channel of all the kernels tiles of $NINBITS$ to convolve image and kernel values using HOBFLOPS multipliers, adders and rectified linear unit (ReLU) activation function. The resultant convolution *SIMD width* values of $NOUTBITS$ wide are stored in corresponding location tiles in the OFM. The HOBFLOPS IFM and kernel layout for single-precision, as defined by FloPoCo is outlined in Equation 1.

$$NINBITS=EXC+SIGN+EXPO\_IN+MANT\_IN \quad (1)$$

The HOBFLOPS OFM layout for single-precision is shown in Equation 2.

$$NOUTBITS=EXC+SIGN+EXPO\_IN+MANT\_IN+1 \quad (2)$$

and for extended-precision, see Equation 3.

$$NOUTBITS=EXC+SIGN+EXPO\_IN+(2 \times MANT\_IN)+1$$

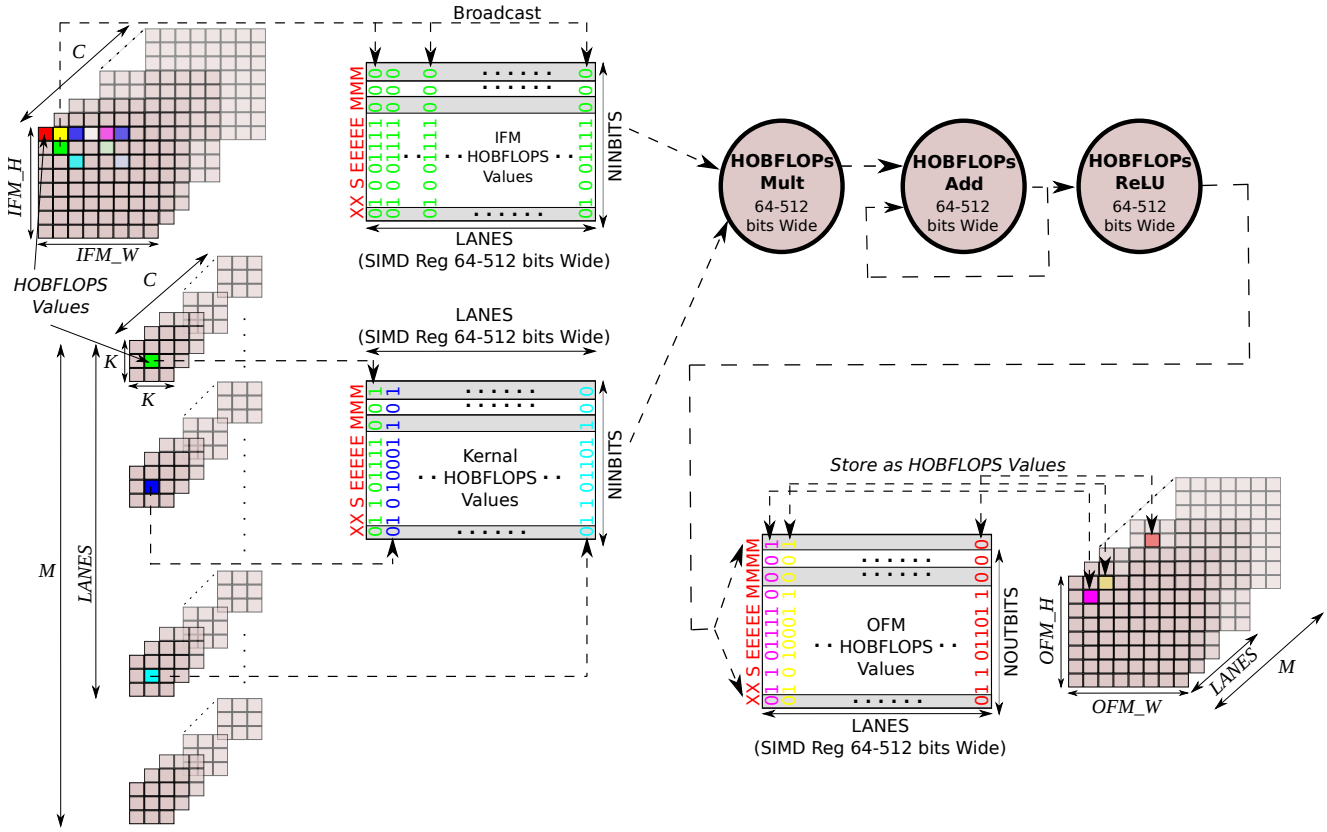


Fig. 5 HOBFLOPS CNN Convolution (IFM and Kernel data pre-transformed to HOBFLOPS, OFM remains in HOBFLOPS layout for use in the next layer. IFM values are broadcast across the kernel values. Arm Neon SIMD LANES are up to 128-bit wide and Intel SIMD LANES are up to 512-bit wide.)

(3)

For example, HOBFLOPS9, as can be seen in Table 3, the input layout $NINBITS$ has 2-bit exception EXC, 1-bit sign $SIGN$, 5-bit exponent $EXPO\_IN$, 3-bit mantissa $MANT\_IN$, which added comes to 11-bits. The single-precision $NOUTBITS$ would be 12-bits ($NINBITS + 1$). The extended-precision $NOUTBITS$ would be 15-bits.

If HOBFLOPS is implemented in a multi-layer CNN, the data between each layer could remain in HOBFLOPS format until the last convolution layer. The OFM at the last convolutional layer could be transformed from HOBFLOPS values to floats resulting in the transformation overhead only occurring at the first and last convolutional layers of the CNN model. An additional pooling layer could be developed in the HOBFLOPS format, for the interface between the last convolutional layer and the fully connected layer, not done in this work.

We repeat for MACs up to HOBFLOPS16e on each processor architecture up to the 512-bit wide AVX512 registers.

4 Evaluation

We implement each of the 8- to 16e-bit HOBFLOPS multipliers and adders in a convolution layer of the MobileNets CNN [14]. We use layer *Conv dw / s2* of MobileNets as it has a high number of channels C and kernels M, perfect for demonstrations of high-dimensional parallelized MAC operations. We compare the HOBFLOPS16 multipliers and adders round-to-nearest-ties-to-even and round-towards-zero modes performance to SoftFP16 rounding near_even and round min modes [13]. This 16-bit FP comparison acts as a baseline as Soft FP8 is not supported by Berkeley's emulation tool.

We target 32-bit to 512-bit registers for AVX2 and AVX512 processors and target 32- and 128-bit registers for the Cortex-A15 processor. We implement 32- to 512-lanes of HOBFLOPS multipliers and adders and capture each of the AVX2 32-, 64-, 128- and 256-bit, AVX512 32-, 64-, 128-, 256 and 512-bit, and Cortex-A16 32-, 64- and 128-bit results.

Three machine types are used to test the HOBFLOPS MAC:

- Arm Cortex-A15 Neon embedded development kit, containing ARMv7 rev 3 (v7l) CPU at 2GHz and 2GB RAM;
- Intel Core-i7 PC, containing Intel Core-i7 8700K CPU at 3.7GHz and 32GB RAM;
- Intel Xeon Gold server PC, containing Intel Xeon Gold 5120 at 2.2GHz and 256GB RAM.

Our cell libraries model-specific cells, see Table 1. We omit the bit clear (BIC) of the Arm Neon in our cell library. Inclusion of bit clear (BIC) prevents the synthesis tool, Cadence

Genus, from optimizing the netlist with SEL (multiplexer) units, leading to a less area efficient netlist.

To further decrease area and increase performance, we produce round-towards-zero versions of the HOBFLOPS8–HOBFLOPS16e adders as the rounding can be dealt with at the end of the layer in the activation function, assuming the non-rounded part of the FP value is retained through to the end of the layer.

The MACs per second of an average of 1000 iterations of a HOBFLOPS adders and multipliers are captured and compared. We use the GNU G++ compiler to optimize the code for the underlying target microprocessor architecture and numbers of registers. We compile the HOBFLOPS CNN code (see Figure 5) to include our HOBFLOPS adders and multipliers, and our cell library (*e.g.*, Listing 6 for AVX2) with G++ (version 8.2.1 20181127 on Arm, version 9.2.0 on Intel AVX2 and version 6.3.0 20170516 on Intel AVX512 machines). We target C++ version 17 and using *-march=native, -mtune=native, -fPIC, -O3* compiler switches with $-msse$ for SSE devices and *-mavx2* for AVX2 devices. When targeting an Intel AVX512 architecture we use the the *-march=skylake-avx512, -mtune=skylake-avx512, -mavx512f, -fPIC, -O3* switches. When targeting an Arm Neon device we use *-march=native, -mtune=native, -fPIC, -O3, -mfpu=neon* to exploit the use of Neon registers.

After the G++ compilation, we inspect the assembler object dump. Within the multiplier and adder units, we find an almost one-to-one correlation of logic bitwise operations in the assembler related to the gates modeled in the cell libraries, with additional loads/stores where the compiler has seen fit to implement.

4.1 Arm Cortex-A15 Performance

We configure an Arm Cortex-A15 Development kit with 2GB RAM, ARCH Linux version 4.14.107-1-ARCH installed, and fix the processor frequency at 2GHz. We run tests for 32-, 64- and 128-lanes and capture performance. We use *taskset* to lock the process to a core of the machine for measurement consistency.

Figure 6a shows 128-lane performance for all arbitrary-precision HOBFLOPS FP between 8- and 16e-bits, IEEE 8- and 32-bit equivalents, Flexfloat, and SoftFP versions. HOBFLOPS16 round-to-nearest-ties-to-even achieves approximately half the performance of Flexfloat and SoftFP 16-bit rounding near_even mode on Arm Neon. However, HOBFLOPS offers arbitrary precision mantissa and exponent FP between 8- and 16-bits, outperforming Flexfloat and SoftFP between HOBFLOPS8 and HOBFLOPS12 bits.

Similarly, HOBFLOPS16 round-towards-zero version shown in Figure 6b demonstrates a slight improvement in performance compared to Flexfloat and SoftFP rounding min mode. Figure 6b also shows HOBFLOPS round-towards-zero versions have an increased performance when compared to HOBFLOPS round-to-nearest-ties-to-even.

HOBFLOPS appears to exhibit a fluctuation around HOBFLOPS8e and HOBFLOPS9 in Figure 6. While there is 1-bit more in the input mantissa of HOBFLOPS9 compared to HOBFLOPS8e, which leads to HOBFLOPS9 containing larger adder/accumulators, Figure 6b shows the round-towards-zero HOBFLOPS9 functionality almost counter intuitively exhibiting slightly greater throughput than HOBFLOPS8e. The greater throughput of the round-towards-zero HOBFLOPS9 is due to the lack of rounding adder, reduced gate area and latency.

The low bit-width and thus reduced hardware synthesis gate count or area as seen in Figure 6 would benefit memory storage and bandwidth within the embedded system. This allows for reduced energy consumption, however, energy consumption is not measured here.

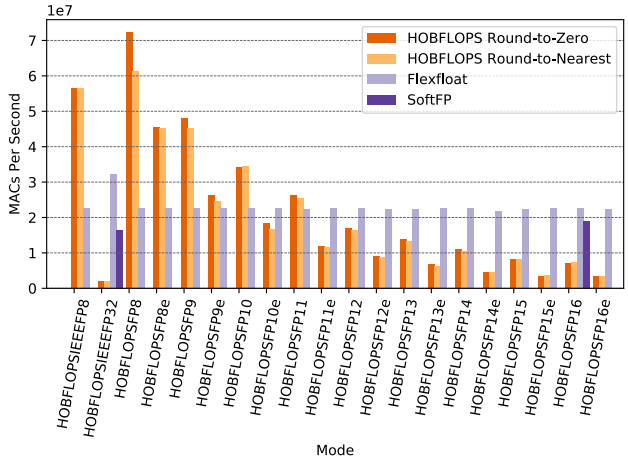
4.2 Intel AVX2 Performance

We configure an Intel Core i7-8700K desktop machine with 32GB RAM, and ARCH Linux 5.3.4-arch1-1 installed. For consistency of performance measurements of various HOBFLOPS configurations, within the BIOS we disable:

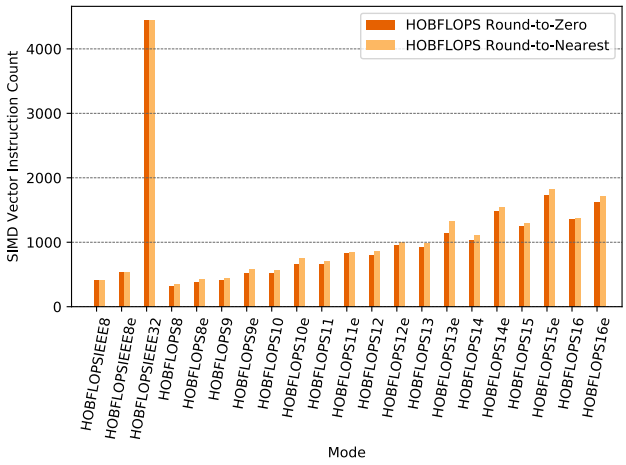
- Intel's SpeedStep (*i.e.*, prevent the CPU performance from ramping up and down);
- Multi-threading (*i.e.*, do not split the program into separate threads);
- TurboBoost (*i.e.*, keep all processor cores running at the same frequency);
- Hyperthreading Control (*i.e.*, keep one program on one processor core);
- C-States control (*i.e.*, prevent power saving from ramping down the core clock frequency).

We alter GRUB's configuration so *intel_pstate* (*i.e.*, lock the processor core clock frequency) and *intel_cstate* are disabled on both *GRUB_CMDLINE_LINUX* and *GRUB_CMDLINE_LINUX_DEFAULT*. This BIOS and Linux Kernel configuration ensures the processor frequency is fixed at 4.6GHz, no power-saving, and each HOBFLOPS instance running at full performance on a single thread and single CPU core. When executing the compiled code, *taskset* is used to lock the process to a single core of the CPU. These configurations allow a reproducible comparison of timing performance of each HOBFLOPS configuration against SoftFP16.

We run tests for 32-, 64-, 128- and 256-lanes and capture performance. Figure 7a shows 256-lane results for all arbitrary-precision HOBFLOPS FP between 8- and 16e-bits, IEEE 8- and 32-bit equivalents, Flexfloat, and SoftFP versions. HOBFLOPS16 performs over 2.5× higher MACs/second when compared to Berkeley's SoftFP16 *MulAdd* rounding near_even mode. The round-towards-zero version of HOBFLOPS16 performs at around 2.7× higher MACs/second

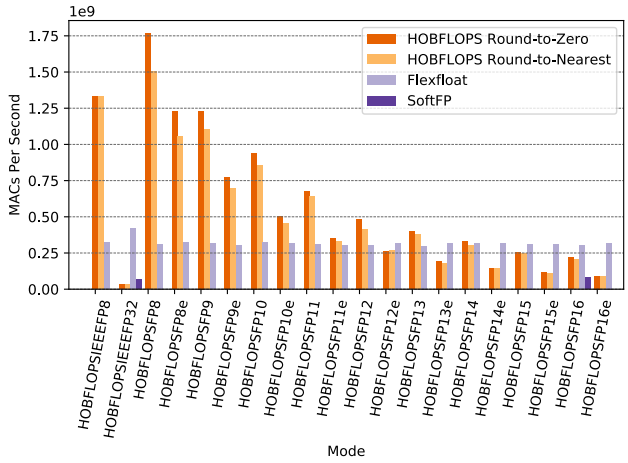


(a) Throughput of Arm Neon, HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs/s Compared to Flexfloat and SoftFP MACs/s - **higher throughput is better**.

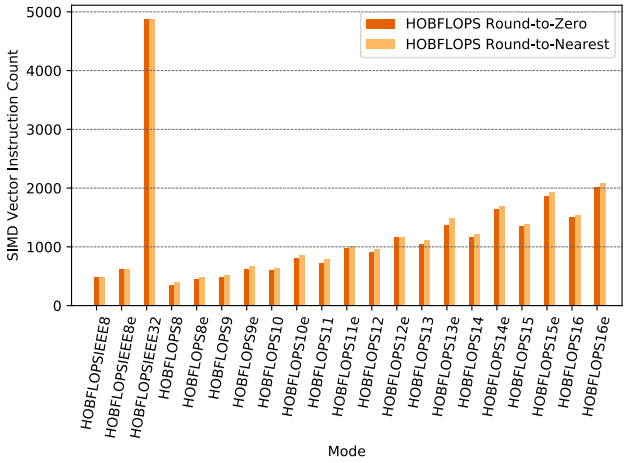


(b) Arm Neon SIMD Bitwise Vector Instruction Count HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs - **lower gate count is better**.

Fig. 6 Throughput and SIMD-Bitwise-Vector-Instruction Count of Arm Neon, HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs in Convolution of Two Rounding Methods.



(a) Throughput of Intel AVX2, HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs/s Compared to Flexfloat and SoftFP MACs/s - **higher throughput is better**.



(b) Intel AVX2 SIMD Bitwise Vector Instruction Count of HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs - **lower gate count is better**.

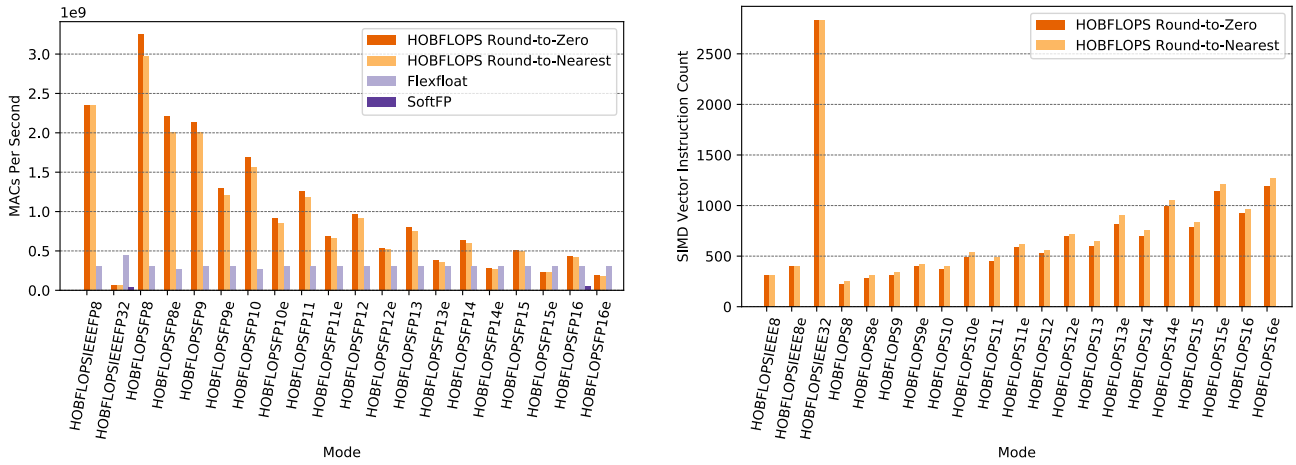
Fig. 7 Throughput and SIMD-Bitwise-Vector-Instruction Count of Intel AVX2, HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs in Convolution of Two Rounding Methods.

when compared to Berkeley's SoftFP16 *MulAdd* rounding min mode. In fact, HOBFLOPS outperforms SoftFP for all versions of between HOBFLOPS8 and HOBFLOPS16 for both round-to-nearest-ties-to-even and round-towards-zero rounding modes. HOBFLOPS outperforms Flexfloat up to HOBFLOPS12e for both rounding modes.

HOBFLOPS performance gain is due to reduced synthesis area of the HOBFLOPS units as seen in Figure 7b. Again, this reduction in area, also seen for round-towards-zero, is key to reduced SIMD bitwise operations being created for the HOBFLOPS MACs and therefore reduced latency through the software HOBFLOPS MACs.

4.3 Intel AVX512 Performance

We configure an Intel Xeon Gold 5120 server with 256GB RAM, and Debian Linux 4.9.189-3+deb9u2. This shared server-grade machine BIOS or clock could not be changed



(a) Throughput of Intel AVX512, HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs/s Compared to Flexfloat and SoftFP MACs/s - **higher throughput is better**.

(b) Intel AVX512 SIMD Bitwise Vector Instruction Count of HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs - **lower gate count is better**.

Fig. 8 Throughput and SIMD-Bitwise-Vector-Instruction Count of Intel AVX512 HOBFLOPSIEEE8-32 and HOBFLOPS8-16e MACs in Convolution of Two Rounding Methods.

as done for the AVX2-based machine. However, *taskset* is used to lock the process to a single CPU core.

We run tests for 32-, 64-, 128-, 256- and 512-lanes. Figure 8a captures 512-lane results for HOBFLOPS FP between 8- and 16e-bits, IEEE 8- and 32-bit equivalents and, Flexfloat, and SoftFP versions. HOBFLOPS16 performs with 8.2× greater MACs throughput than SoftFP16 *MulAdd* rounding near_even mode. For the 512-lane round-towards-zero results, HOBFLOPS16 performs at 8.4× the MACs throughput of SoftFP16 *MulAdd* rounding min mode. HOBFLOPS significantly outperforms Flexfloat for HOBFLOPS8 to HOBFLOPS16 for both round-to-nearest-ties-to-even and round-towards-zero. HOBFLOPS9 performs at approximately 2 billion MACs/second, around 7× the performance of Flexfloat 9-bit.

HOBFLOPS performance is due to HOBFLOPS lower hardware synthesis area. When the netlists are converted to software bitwise operations, HOBFLOPS translates to fewer SIMD 3-input ternary logic LUT instructions the MACs. Figure 8b shows HOBFLOPS16 area on the AVX512 platform is 38% smaller than HOBFLOPS16 area on AVX2. A further slight performance boost is seen for round-towards-zero.

5 Conclusion

Arbitrary precision floating-point computation is largely unavailable in CPUs or FPUs. FP simulation latency is often too high for computationally intensive systems such as CNNs. FP simulation has little support for 8-bit or arbitrary precision.

We propose HOBFLOPS, that generates fast custom-precision bitslice-parallel software FP arithmetic. We optimize HOBFLOPS using a hardware design flow, cell libraries and custom code-generator. We generate efficient software-emulated FP operators with an arbitrary precision mantissa and exponent. We pack the generated FP operators efficiently into the target processors SIMD vector registers. HOBFLOPS offers FP with custom range and precision, useful for FP CNN acceleration where memory storage and bandwidth are limited.

We experiment with large numbers of channels and kernels in CNN convolution. We compare the MAC performance in CNN convolution on Arm and Intel processors against Flexfloat, and Berkeley's SoftFP. HOBFLOPS outperforms Flexfloat by over 10×, 5×, and 3× on Intel AVX512, AVX2 and Arm Neon respectively. HOBFLOPS offers arbitrary-precision FP with custom range and precision, *e.g.*, HOBFLOPS9, which outperforms Flexfloat 9-bit by over 7×, 3×, and 2× on Intel AVX512, AVX2 and Arm Neon respectively.

The performance gains are due to the optimized hardware synthesis area of the MACs, translating to fewer bitwise operations. Additionally, the bitslice parallelism of the very wide vectorization of the MACs of CNN convolution contributes to the performance boost. While we show results for 8- and 16-bit with a fixed exponent, HOBFLOPS supports the emulation of any required precision of FP arithmetic at any bit-width of mantissa or exponent, *e.g.*, FP9 containing a 1-bit sign, 5-bit exponent and 3-bit mantissa, or FP11 containing a 1-bit sign, 5-bit exponent and 5-bit mantissa, not efficiently supported with other software FP emulation.

Other processors, *e.g.*, RiscV or Arm SVE can be supported by producing a standard cell library of the processors bitwise vector extensions. HOBFLOPS allows researchers to

prototype different levels of custom FP precision in the arithmetic of software CNN accelerators. Furthermore, HOBFLOPS fast custom-precision FP CNNs may be valuable in cases where memory bandwidth is limited.

6 Compliance with Ethical Standards

6.1 Ethical Approval

This article does not contain any studies with human participants or animals performed by any of the authors.

6.2 Funding Details

This research is supported by Science Foundation Ireland, Project 12/IA/1381. We thank the Institute of Technology Carlow, Carlow, Ireland for their support.

6.3 Conflict of Interest

The authors declare that they have no conflict of interest.

6.4 Informed Consent

As no human participants studies are contained in this work, no informed consent was required.

7 Authorship Contributions

David Gregg is the advisor and editor of this article. James Garland designed and tested the system and wrote the article.

References

1. Arm: Arm Neon Intrinsics Reference for ACLE Q2 2019 (2019). https://developer.arm.com/docs/101028/0010/advanced-simd-neon-intrinsics
2. Brayton, R., Mishchenko, A.: ABC: An Academic Industrial-Strength Verification Tool. In: International Conference on Computer Aided Verification, Computer Aided Verification, pp. 24–40. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
3. C. Wolf J. Glaser, J.K.: Yosys - A Free Verilog Synthesis Suite. In: Proceedings of the 21st Austrian Workshop on Microelectronics, Austrochip 2013. Springer, USA (2013)
4. Choquette, J., Gandhi, W., Giroux, O., Stam, N., Krashinsky, R.: NVIDIA A100 Tensor Core GPU: Performance and Innovation. IEEE Micro **01**(01), 1–1 (2021)
5. Chung, E., Fowers, J., Ovtcharov, K., et al.: Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. IEEE Micro **38**(2), 8–20 (2018). DOI 10.1109/MM.2018.022071131
6. de Dinechin, F., Klein, C., Pasca, B.: Generating high-performance custom floating-point pipelines. In: 2009 International Conference on Field Programmable Logic and Applications, pp. 59–64. IEEE, USA (2009). DOI 10.1109/FPL.2009.5272553
7. DiCecco, R., Sun, L., Chow, P.: FPGA-based training of convolutional neural networks with a reduced precision floating-point library. In: 2017 International Conference on Field Programmable Technology (ICFPT), pp. 239–242. IEEE (2017)
8. Farabet, C., Martini, B., Akselrod, P., et al.: Hardware accelerated convolutional neural networks for synthetic vision systems. In: Proceedings of 2010 IEEE International Symposium on Circuits and Systems, pp. 257–260. IEEE ISCAS, Paris, France (2010). DOI 10.1109/ISCAS.2010.5537908
9. Fowers, J., Ovtcharov, K., Papamichael, M., et al.: A configurable cloud-scale DNN processor for real-time AI. In: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pp. 1–14. IEEE ISCAS, Los Angeles, California, USA (2018)
10. Gong, Y., Liu, L., Yang, M., Bourdev, L.: Compressing deep convolutional networks using vector quantization. arXiv preprint arXiv:1412.6115 (2014)
11. Google: BFloat16: The secret to high performance on Cloud TPUs. Google Cloud Blog (2019)
12. Gupta, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P.: Deep learning with limited numerical precision. In: Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37. JMLR.org (2015)
13. Hauser, J.: The SoftFloat and TestFloat Validation Suite for Binary Floating-Point Arithmetic. University of California, Berkeley, Tech. Rep (1999)
14. Howard, A.G., Zhu, M., Chen, B., et al.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)
15. IEEE: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) pp. 1–84 (2019). DOI 10.1109/IEEESTD.2019.8766229
16. Intel: Intel Intrinsics Guide (2020). https://software.intel.com/sites/landingpage/IntrinsicsGuide/
17. Johnson, J.: Rethinking floating point for deep learning. arXiv preprint arXiv:1811.01721 (2018)
18. Jouppi, N.P., Young, C., Patil, N., et al.: In-Datacenter Performance Analysis of a Tensor Processing Unit. SIGARCH Comput. Archit. News **45**(2), 1–12 (2017). DOI 10.1145/3140659.3080246
19. Kang, H.J.: Short floating-point representation for convolutional neural network inference. IEICE Electronics Express p. 15.20180909 (2018)
20. Lo, C.Y., Lau, F.C., Sham, C.W.: Fixed-point implementation of convolutional neural networks for image classification. In: 2018 International Conference on Advanced Technologies for Communications (ATC), pp. 105–109. IEEE (2018)
21. Rzayev, T., Moradi, S., Albonesi, D.H., Manchar, R.: Deeprecon: Dynamically reconfigurable architecture for accelerating deep neural networks. In: 2017 International Joint Conference on Neural Networks (IJCNN), pp. 116–124. IEEE (2017)
22. Sze, V., Chen, Y.H., Yang, T.J., Emer, J.S.: Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE **105**(12), 2295–2329 (2017)
23. Tagliavini, G., Marongiu, A., Benini, L.: Flexfloat: A software library for transprecision computing. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(1), 145–156 (2018)
24. Xu, S., Gregg, D.: Bitslice Vectors: A Software Approach to Customizable Data Precision on Processors with SIMD Extensions. In: 2017 46th International Conference on Parallel Processing (ICPP), pp. 442–451. IEEE, USA (2017). DOI 10.1109/ICPP.2017.53
25. Zaidy, A.: Accuracy and Performance Improvements in Custom CNN Architectures. Purdue University (2016)