



# AI Engine Development

---

[See Vitis™ Development Environment on xilinx.com](#)

[See Vitis™ AI Development Environment on xilinx.com](#)

## Softmax Function

---

**Version: Vitis 2024.1**

### Table of Contents

- [1. Introduction](#)
- [2. Softmax Function Definition](#)
- [3. Computing the Exponential Function](#)
- [4. AI Engine Implementation](#)
- [5. AI Engine Kernel Processing](#)
- [6. Running the Example](#)
- [7. Analyzing Results](#)

[References](#)

[Support](#)

[License](#)

### Introduction

Machine Learning is pervasive in just about any modern application. Whereas machine learning tends to infer processing of structured data, the sub-class of deep learning is often applied to unstructured data where more abstraction is required to extract significant features from a data set. Some applications that have proven to benefit from the application of deep learning are natural language processing and image classification, where the relationship between input data and desired output can be exceedingly complex.

Deep learning solutions are often created in the form of a neural network, as depicted in the following figure.

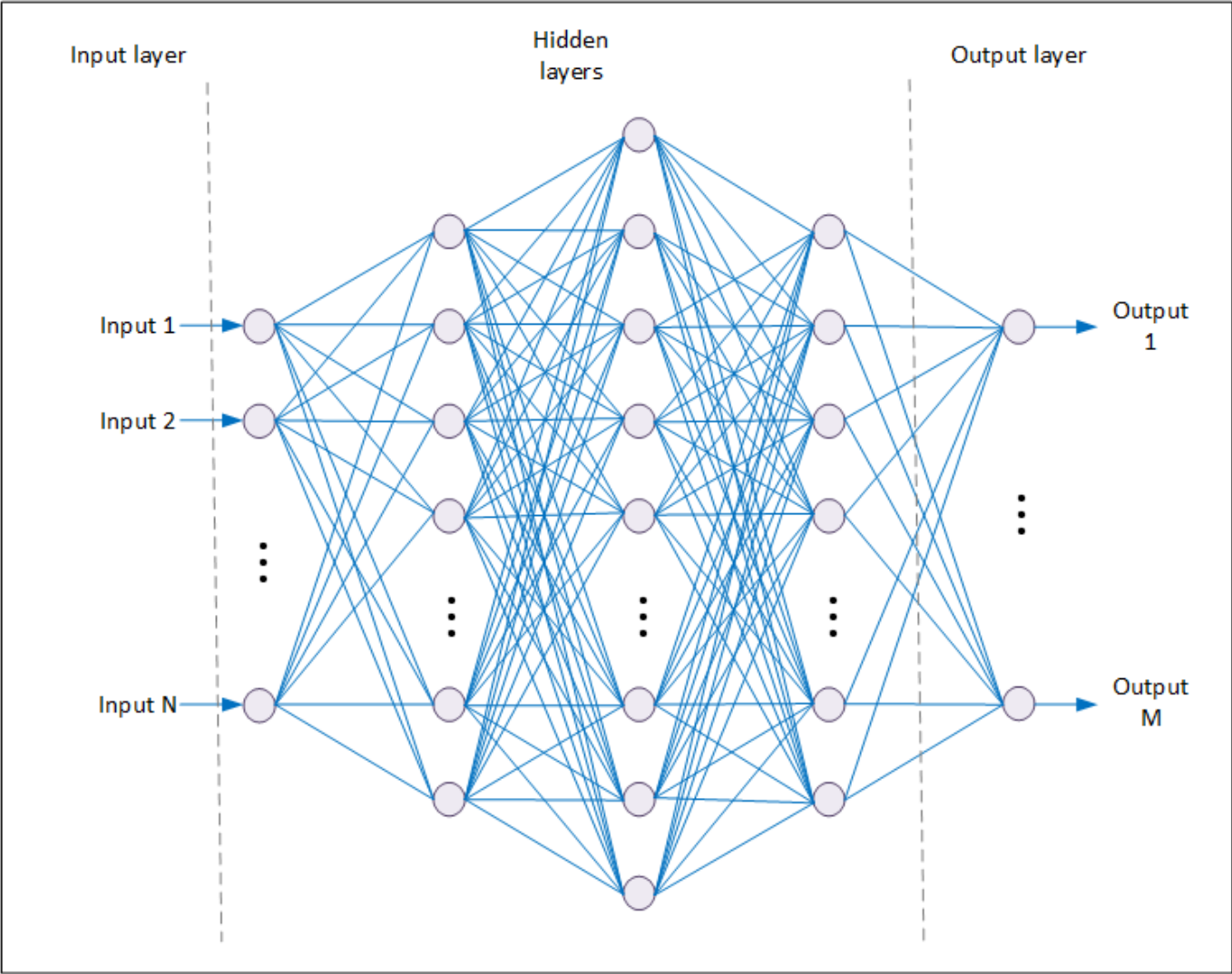


Figure 1 - Artificial Neural Network

An artificial neural network is comprised of layers of neurons intended to mimic behavior of the human brain. Each layer consists of nodes, which are connected to adjacent layers. The number of nodes in a layer and number of hidden layers can vary depending on implementation. Nodes in the graph represent individual neurons, which might be depicted in more detail as shown in the following figure.

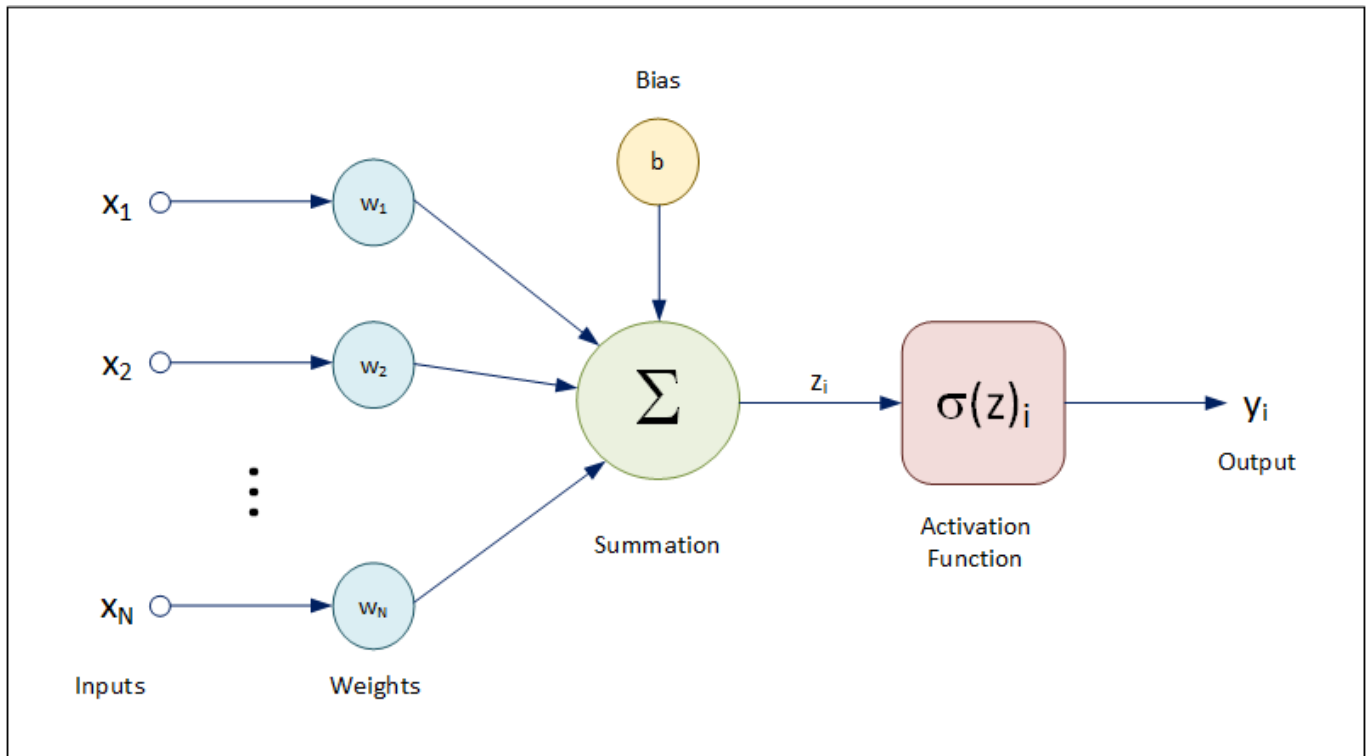


Figure 2 - Artificial Neuron

Input data to each layer is multiplied by a weight before being summed together. These weights are determined by training the neural network using empirical data. The activation function transforms the weighted sum of inputs into an output signal. Activation functions add non-linearity to neural networks, enabling them to effectively approximate any complex function of the input data. Different types of activation functions are available to use within the various layers of the neural network. An activation function often applied to the output layer is known as Softmax.

## Softmax Function Definition

The softmax function is defined for a vector of real values  $\mathbf{z} = (z_1, z_2, \dots, z_M)$  by the equation

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^M e^{z_j}}$$

where  $z_i$  are the individual outputs of the layer. Softmax differs from other popular activation functions in that it takes into account the entire layer and scales outputs so they sum to a value of 1. Each individual output can then be interpreted as a probability. So in classification problems, softmax output may be interpreted as probability that the input data belongs to a specified class.

When computing the softmax function, there is a risk of overflow occurring during evaluation of the individual exponential functions that comprise the formula. For bfloat16 floating-point numbers, the exponential function overflows when input values exceed 88.5. To avoid overflow, the softmax function is often evaluated using the equivalent formula

$$\sigma(\mathbf{z})_i = \frac{e^{z_i - \alpha}}{\sum_{j=1}^M e^{z_j - \alpha}}$$

where  $\alpha$  is a real-valued constant. In particular,  $\alpha$  is often chosen to be the maximum of all  $z_i$  values comprising the input vector. By subtracting the maximum value from all others, inputs to the

exponential functions are constrained to the range  $(-\infty, 0]$ , which in turn limits the exponential function values to the range  $[0, 1]$ .

Another alternative to evaluating the softmax function is to use the equivalent formula

$$\text{Large} \left( \sum \left( \mathbf{z} \right)_{\text{small } i} \right) = \exp \left( z_i - \log \sum_{j=1}^M e^{z_j} \right)$$

which is attractive because no division is required. However, it has been shown that in practice this formula tends to produce larger computational errors [1].

## Computing the Exponential Function

### IEEE 754 Format Trick

In addition to basic arithmetic operations, softmax computation depends on efficient evaluation of the exponential function. While there are several ways to accomplish this, an attractive alternative is to estimate the exponential function using a trick based on IEEE 754 floating-point format [2]. A double-precision, floating-point number represented by IEEE 754 format is shown in the following figure.

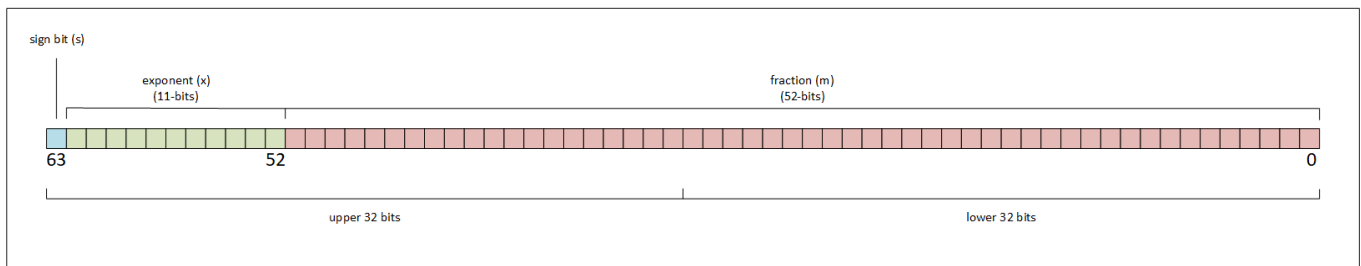


Figure 3 - IEEE 754 Format for Double-Precision Numbers

This format is used to represent a number  $(-1)^s 2^{x-x_0} m$ , where  $s$  is a sign bit,  $m$  is the 52-bit fractional part of a normalized mantissa, and  $x$  is an 11-bit exponent with bias  $x_0 = 1023$ .

Approximation is based on the identity  $e^y = 2^{y/\log(2)}$ . So for any floating-point number  $y$ , the value  $e^y$  is approximated by setting the exponent  $x$  of the result to  $y/\log(2) + x_0$ . To perform the computation, it helps to divide a double precision number into two groups comprised of upper 32 bits and lower 32 bits. The lower 32 bits are set to 0 in this approximation, while the upper 32 bits are the same bits used to represent the signed 32-bit integer value

$$I_{\text{upper}} = \left\lfloor \frac{2^{20} y}{\log(2)} + 2^{20} x_0 - C \right\rfloor$$

A factor of  $2^{20}$  represents a binary shift necessary to align with the exponent field of the IEEE 754 format. Residual mantissa bits help provide a degree of interpolation between exponent values. The parameter  $C$  is a correction factor meant to mitigate estimation error. It was found that a value of  $C=60801$  minimizes RMS error [2]. This estimation method might be adapted for other variations of floating-point number representations, such as bfloat16 data type.

### Improving Accuracy

While this trick is computationally very efficient, it can result in an estimation error as large as ~4% of the true value. To reclaim lost accuracy, a method was proposed in [3] where all 64 bits are used in computation and a

correction function  $F$  is defined. To derive the solution, begin by expressing the exponential function in the form

$$e^y = 2^{y/\log(2)} = 2^{\{y\}_i} \cdot 2^{\{y\}_f} \approx 2^{\{y\}_i} \cdot \left(1 + m - F\right),$$

where  $\{y\}_i$  and  $\{y\}_f$  are the integer and fractional parts of  $y/\log(2)$ , respectively. The correction function is defined by solving for  $F = 1 + m - 2^{\{y\}_f}$ . Since  $m \equiv \{y\}_f$ , an equivalent expression is  $F(\{y\}_f) = 1 + \{y\}_f - 2^{\{y\}_f}$ . The correction function  $F(\{y\}_f)$  might be modeled as a polynomial, where  $\{y\}_f$  is defined over the range  $[0, 1)$ .

The exponential function estimate is obtained by computing

$$I = \left\lfloor \frac{2^{52}}{\log(2)} \left( y - \log(2) F(\{y\}_f) \right) + 2^{52} x_0 \right\rfloor$$

as a 64-bit signed integer then reinterpreting the result as a double-precision floating-point value. Since all 64 bits are used, a factor  $2^{52}$  is necessary to align to the exponential field of the IEEE 754 format.

### Adapting for bfloat16 Floating-Point

AMD Versal™ Edge Adaptive SoCs primarily contain a variant of AI Engine processor, which has bfloat16 floating-point as a native data type. The structure of a bfloat16 number is shown in the following figure.

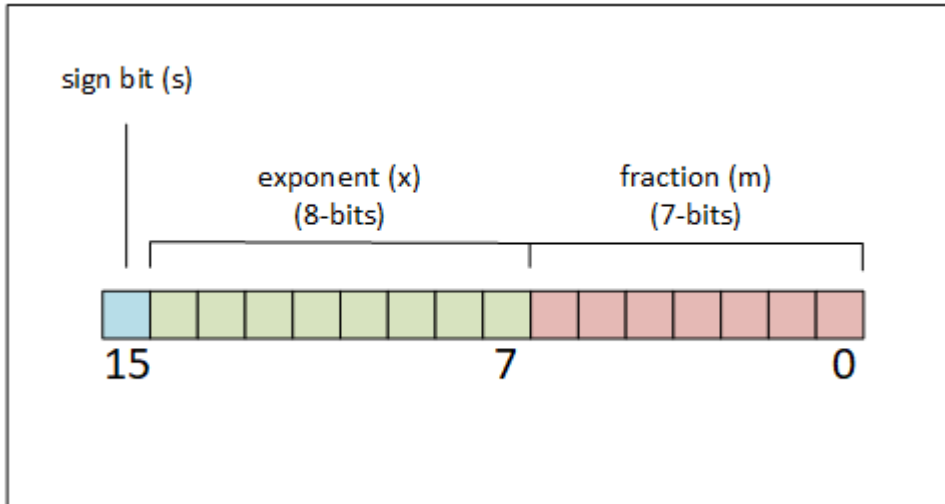


Figure 4 - bfloat16 Floating-Point Format

This format is structurally similar to double-precision format, but with reduced dynamic range and precision due to fewer bits being used to represent the exponent and mantissa. The number of exponent bits is the same as IEEE 754 single-precision format, giving bfloat16 essentially the same dynamic range. This is attractive in deep learning and other applications where dynamic range is more important than precision. To adapt exponential function approximation to bfloat16 data types, the equation becomes

$$I = \left\lfloor \frac{2^7}{\log(2)} \left( y - \log(2) F(\{y\}_f) \right) + 2^7 x_0 \right\rfloor$$

where  $x_0 = 127$  and  $I$  is computed as a signed 16-bit integer which is then reinterpreted as bfloat16.

The correction function  $F(\{y\}_f)$  may be approximated with a polynomial. However, quantization introduced by bfloat16 arithmetic used to evaluate the polynomial, and remaining softmax function computation, counteracts the benefit of using a correction function. In addition, for bfloat16 numbers greater than 128.0, there is no fractional part to be used as an argument for the correction polynomial. When considering

exponential function evaluation with bfloat16 data types, unnecessary computation can be avoided by using the simpler estimation

$$I = \left\lfloor \frac{2^7}{\log(2)} y + 2^7 x_0 \right\rfloor$$

which becomes

$$I = \left\lfloor 185 y + 16256 x_0 \right\rfloor$$

after taking quantization of coefficients into account. For an example of using a correction function in exponential function estimation, consult the [Softmax Function Tutorial](#) based on single-precision floating-point computation.

## AI Engine Implementation

One of the key parameters impacting the amount of computation required for evaluating the softmax function is the number of classes. For the example presented here, 2048 classes are used to represent the output nodes of a neural network. Since data is in bfloat16 floating-point format, the floating-point vector unit of the AI Engine, shown in the following figure, is required. The floating-point multiply unit is designed to process vectors with 16 lanes, so softmax computation is designed to accommodate a SIMD factor of 16.

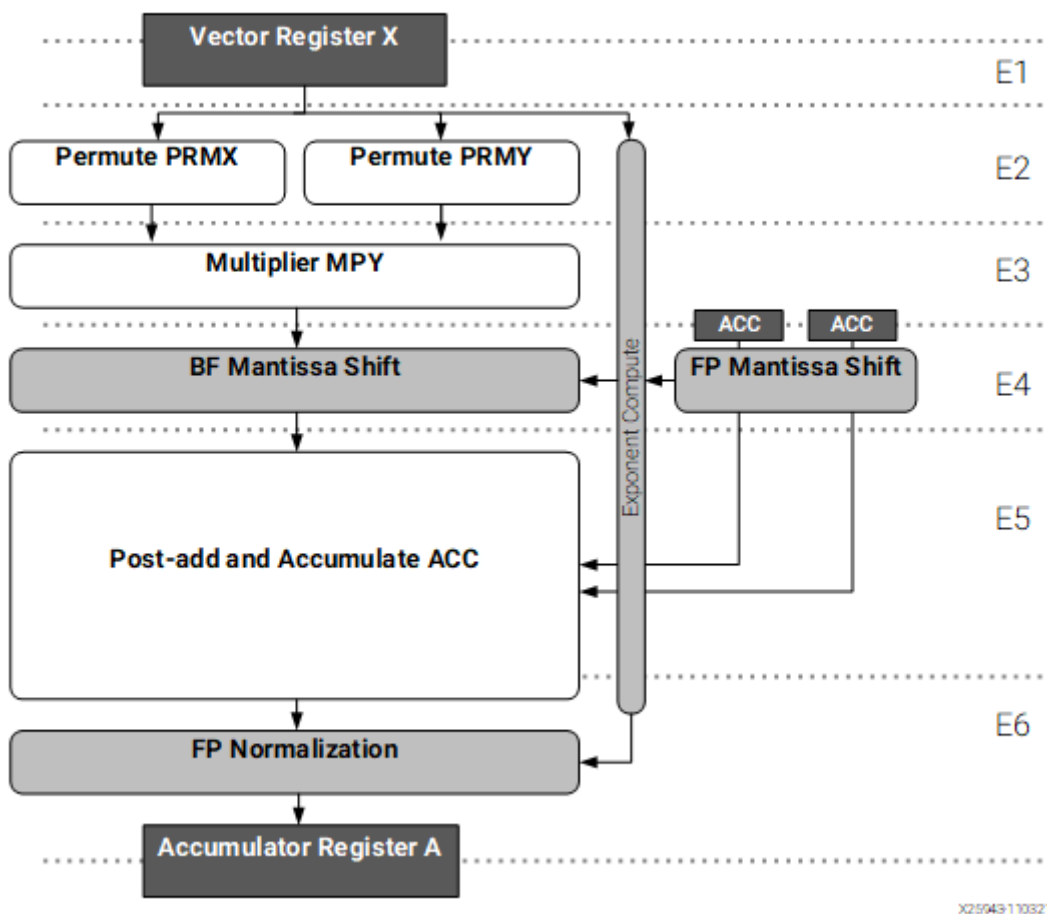


Figure 5 - AI Engine Floating-Point Vector Unit

From the preceding figure, you can observe that the floating-point vector processor has a pipeline depth of six. To improve compute efficiency, kernel processing should be designed to keep the pipeline full. This is not

the case when a computation needs to wait for intermediate results to proceed. To take full advantage of software pipelining, computation is broken up into components, where intermediate results are stored in data memory. Each loop in the kernel processes a specific computation for the entire number of classes in the softmax function, 16 elements at a time. Each invocation of the kernel computes a single softmax vector comprising the values for all outputs according to the following processing order:

1. Read and store all input values while searching for the maximum value, then subtract the maximum value from all inputs. (two computational loops)
2. Compute exponential function of all values. (two computational loops)
3. Sum all exponentials and invert sum to obtain scaling factor. (single computational loop plus scalar processor inverse operation)
4. Multiply all exponentials by scaling factor and send result to output. (single computational loop)

## AI Engine Kernel Processing

### Kernel Data Interface

While this kernel is designed for `bfloat16` processing, the function signature indicates that data type at the interface is `int16` for both input and output.

```
void softmax_kernel::softmax(input_stream<int16>* in, output_stream<int16>* out)
```

Each of these `int16` values represents the 16-bits of a `bfloat16` value. When used by the kernel, values are reinterpreted as `bfloat16` for processing. The reason for this is that when performing AI Engine simulation, text files are used for input and output of data. Use of `int16` preserves all bits of the floating-point number when read from or written to a text file and allows for test vector matching at the bit level.

The streaming interfaces are used for input and output, which reduces latency and eliminates the need for ping pong buffers in data memory.

### Kernel Code

The first two processing loops of the kernel are shown below. The iterators are defined for work buffers specified in data memory to hold intermediate results. The first processing loop reads input values for softmax computation and stores them to memory while searching for the maximum input value. Once the maximum value is determined, a second loop reads input values from memory, subtracts the maximum value from each, and stores results back to memory. The iterators are reset after each loop completes to prepare them for the next loop.

```
// set rounding to match MATLAB generated test vectors
aie::set_rounding(aie::rounding_mode::symmetric_inf);
aie::set_saturation(aie::saturation_mode::saturate);

// work buffers in data memory
auto pWbufA = aie::begin_restrict_vector<16>(wbufA);
auto pWbufB = aie::begin_restrict_vector<16>(wbufB);
```

```

// read and store data while searching for maximum value
float max_val = -2 ^ 127;

for (unsigned i=0; i < BUFFSZ/16; i++)
    chess_prepare_for_pipelining
    chess_loop_count(BUFFSZ/16)
{
    aie::vector<bfloat16,16> vin = v16bfloat16(readincr_v<16>(in));
    float vmax = aie::reduce_max(vin);
    if (vmax > max_val) {
        max_val = vmax;
    }
    *pWbufA++ = v16int16(vin);
}

pWbufA -= (BUFFSZ/16);

chess_separator();

// subtract maximum value from all input values
aie::accum<accfloat,16> accM;
accM.from_vector(aie::broadcast<float,16>(max_val));

for (unsigned i=0; i < BUFFSZ/16; i++)
    chess_prepare_for_pipelining
    chess_loop_count(BUFFSZ/16)
{
    aie::vector<bfloat16,16> vecA = v16bfloat16(*pWbufA++);
    aie::accum<accfloat,16> accA;
    accA.from_vector(vecA);
    *pWbufB++ = v16int16(aie::to_vector<bfloat16>(aie::sub(accA, accM)));
}

pWbufA -= (BUFFSZ/16);
pWbufB -= (BUFFSZ/16);

chess_separator();

```

The next segment of kernel code, as shown below, is comprised of two computational loops used to evaluate the exponential function for all inputs. The first loop scales all inputs by  $\frac{1}{\log(2)}$  and adds the exponent offset. Also built into this computation is scaling by a factor of  $2^7$  to align the result with the exponent field of the **bfloat16** format. Constants used in this computation are defined in the kernel header file. The second loop takes these values and extracts the 16-bit integer part used to represent a **bfloat16** number. Since the maximum value is subtracted from each input, the exponential function values should all be in the range  $[0, 1]$ . Some instructions are added to check for values outside this range. If any such values are detected, they are assumed to be the result of overflow and set to zero.

```

/***** Start of computation of exponential functions of all input values *****/
// convert results to IEEE 754 format - use 2 loops
aie::accum<accfloat,16> b_acc;

```



```

b_acc.from_vector(aie::broadcast<float,16>(exp_B));

for (unsigned i=0; i < BUFFSZ/16; i++)
    chess_prepare_for_pipelining
    chess_loop_count(BUFFSZ/16)
{
    aie::vector<bfloat16,16> vecB = v16bfloat16(*pWbufB++);
    aie::accum<accfloat,16> aout = aie::mac(b_acc, vecB, exp_S);
    *pWbufA++ = v16int16(aie::to_vector<bfloat16>(aout));
}

pWbufA -= (BUFFSZ/16);
pWbufB -= (BUFFSZ/16);

chess_separator();

for (unsigned i=0; i < BUFFSZ/16; i++)
    chess_prepare_for_pipelining
    chess_loop_count(BUFFSZ/16)
{
    aie::vector<bfloat16,16> vecA = v16bfloat16(*pWbufA++);
    aie::vector<int16,16> exp_i = aie::to_fixed<int16>(vecA,0);

    // integer values should be in the range [0, 16,256], find outliers and set to
    zero
    aie::mask<16> msk_neg = aie::lt(exp_i,int16(0));
    aie::vector<int16,16> exp_bnd = aie::select(exp_i, aie::zeros<int16,16>(),
msk_neg);
    aie::mask<16> msk_pos = aie::gt(exp_bnd, int16(16256));
    exp_bnd = aie::select(exp_bnd, aie::zeros<int16,16>(), msk_pos);
    *pWbufB++ = exp_bnd;
}

pWbufA -= (BUFFSZ/16);
pWbufB -= (BUFFSZ/16);

/***** End of computation of exponential functions of all input values *****/

chess_separator();

```

With the exponential function of all inputs computed, the softmax function is evaluated by the kernel code shown below. The first loop sums exponential values in individual vector lanes. Next, individual vector lanes are summed, and the scalar processor is invoked to compute a scale factor, which is the inverse of the sum. The final loop multiplies all the exponential values by the scale factor and sends the result to output.

```

// accumulate all vectors to determine scale factor
aie::accum<accfloat,16> accsum;
accsum.from_vector(aie::zeros<bfloat16,16>());

for (unsigned i=0; i < BUFFSZ/16; i++)
    chess_prepare_for_pipelining

```

```

    chess_loop_count(BUFFSZ/16)
{
    aie::vector<bfloat16,16> vecB = v16bfloat16(*pWbufB++);
    accsum = aie::add(accsum, vecB);
}

pWbufB -= (BUFFSZ/16);

chess_separator();

// compute inverse
bfloat16 scl_fctr = aie::inv(aie::reduce_add(aie::to_vector<bfloat16>(accsum)));

// scale values and write to output
for (unsigned i=0; i < BUFFSZ/16; i++)
    chess_prepare_for_pipelining
    chess_loop_count(BUFFSZ/16)
{
    aie::vector<bfloat16,16> vecB = v16bfloat16(*pWbufB++);
    aie::vector<int16,16> vout = v16int16(aie::to_vector<bfloat16>(aie::mul(vecB,
scl_fctr)));
    writeincr(out, vout);
}

```

## Running the Example

Running the example requires that both MATLAB and AMD Vitis™ tools are installed and configured correctly. After downloading the files, cd into the `.../05-Softmax-Function/aie-ml/` directory and use the make build process.

### Generating Test Vectors

Prior to running the AI Engine graph simulation, test vectors are required to provide input. Scripts are also provided to compare with AI Engine simulator output for verification. To generate the vectors, run the following command:

```
$ make gen_vectors
```

Test vectors are included with this tutorial, so this step is not strictly necessary prior to AI Engine simulation. If desired, one may run the `matlab/genvectors_softmax_aie_ml_bfloat16.m` function from the MATLAB environment to generate test vectors. This function accepts input arguments specifying the number of softmax classes and the number of softmax vectors to generate. This function also creates a file `aie-ml/src/config.h`, which configures the AIE kernel and simulation to properly process the generated test vectors.

### Running x86 Simulation

To perform a functional x86 simulation, enter the following sequence of commands:

```
$ make x86com  
$ make x86sim  
$ make check_x86sim
```

The first command compiles graph code for simulation on an x86 processor, the second command runs the simulation, and the final command invokes MATLAB to compare the simulator output with test vectors.

## Running AI Engine Simulation

To perform AI Engine emulation using the SystemC simulator, enter the following sequence of commands:

```
$ make aiecom  
$ make aiesim  
$ make check_aiesim
```

The first command compiles graph code for the SystemC simulator, the second command runs the simulation, and the final command invokes MATLAB to compare simulation output with test vectors. If it is desired to generate trace and profile data during simulation, use the sequence:

```
$ make aiecom  
$ make profile  
$ make check_aiesim
```

## Analyzing Results

### Vitis Analyzer

Vitis Analyzer is an essential tool for accessing information on compilation, simulation, and implementation of AI Engine graphs. It can be used to obtain a summary on profiling data and to graphically display trace events. The tool might be invoked with the `vitis_analyzer` command, or for this example, by entering:

```
$ make analyze
```

The Graph view displays connectivity of the AI Engine graph, which for this example, is displayed in the following figure. This simple example shows a softmax kernel with streaming data input and output. Also visible are two buffers in data memory used for holding intermediate computations.

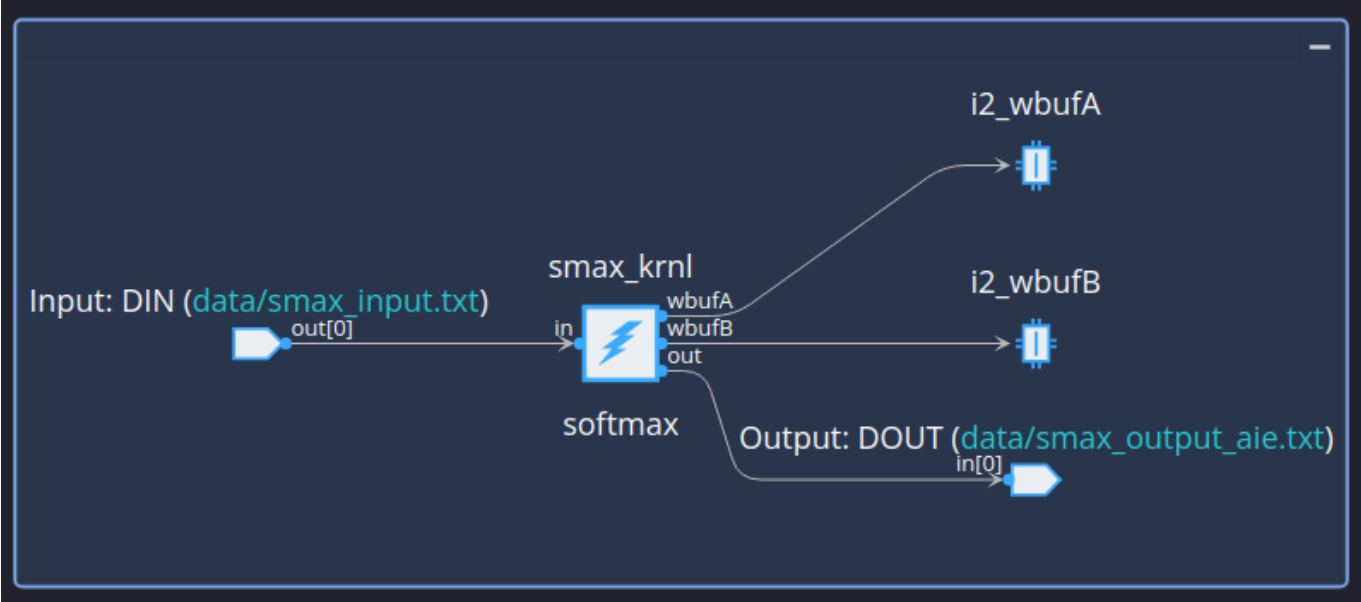


Figure 6 - Vitis Analyzer Graph View

The Array view displays how the AI Engine graph is mapped to the AI Engine array for the device specified. This example uses a VE2802 Versal AI Edge device which contains 304 AI Engine tiles. As shown in the following figure, this example utilizes a single AI Engine tile, which contains an AI Engine for kernel processing along with work buffers in data memory. The amount of memory required for these buffers depends on the number of classes in the softmax function. For this example with 2048 classes, only a small portion of the 64 kB associated with the tile is utilized.



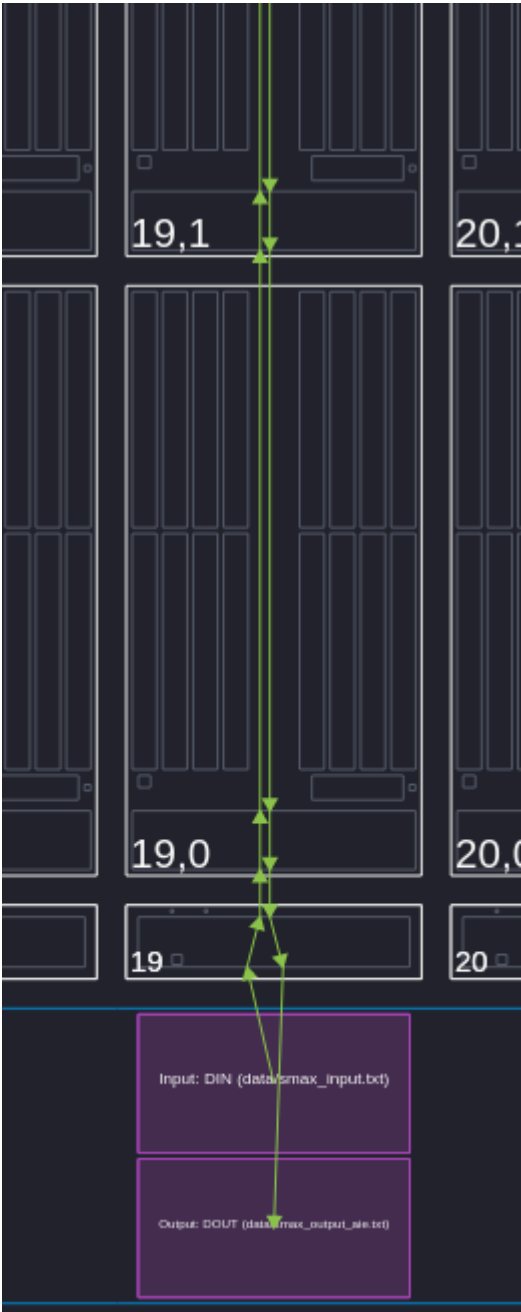


Figure 7 - Vitis Analyzer Array View

The following figure contains information from the Profile view. The highlighted fields show that the softmax kernel takes up to 8,083 cycles to process 2048 classes. For lowest speed grade Versal devices, this translates to a processing rate of ~123,716 softmax computations per second. Higher speed grade devices could have a peak rate of ~154,645 softmax computations per second.

MIN FUNCTION TIME (CYCLES)	AVG FUNCTION TIME (CYCLES)	MAX FUNCTION TIME (CYCLES)
8078	8081	8083
207	207	207
43	43	43
36	36	36
22	22	22
22	22	22

Figure 8 - Vitis Analyzer Profile View

The following figure shows part of the Vitis Analyzer trace view. The cursors show that the time between the end of one kernel invocation to the end of the next is 10.164  $\mu$ s. The additional overhead causes softmax computation rate to decrease to ~98,386 computations per second in higher speed grade devices. To improve processing rate, one might investigate using buffer kernel inputs instead of streams. This causes kernel data to be loaded from the wider memory interfaces. Generated microcode could also be analyzed to determine how computation could be further optimized.

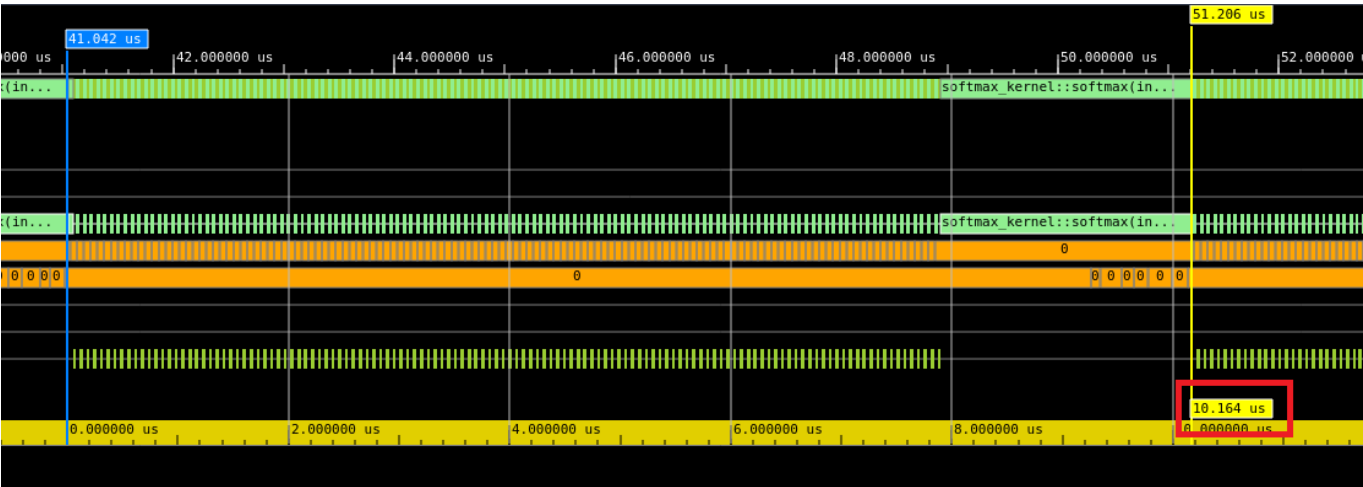


Figure 9 - Vitis Analyzer Trace View

Test Vector Comparison

When comparing simulation results against test vectors, a MATLAB script is invoked to perform the processing. An example of a successful comparison is shown in the following figure.

```

MATCH: Test vector 1
MATCH: Test vector 2
MATCH: Test vector 3
MATCH: Test vector 4
MATCH: Test vector 5
MATCH: Test vector 6
MATCH: Test vector 7
MATCH: Test vector 8
MATCH: Test vector 9
MATCH: Test vector 10

Comparing to gold reference (double precision)
Vec 1: softmax sum = 9.967266397e-01, max_diff = 1.488417584e-02, smax(bfloat16) = 6.328125000e-01, smax(gold) = 6.476966758e-01
Vec 2: softmax sum = 1.000090610e+00, max_diff = 4.436714249e-03, smax(bfloat16) = 5.834960938e-02, smax(gold) = 6.278632362e-02
Vec 3: softmax sum = 9.948491985e-01, max_diff = 5.809441965e-03, smax(bfloat16) = 9.609375000e-01, smax(gold) = 9.667469420e-01
Vec 4: softmax sum = 9.983031750e-01, max_diff = 8.904701439e-03, smax(bfloat16) = 8.085937500e-01, smax(gold) = 7.996890486e-01
Vec 5: softmax sum = 9.999716245e-01, max_diff = 1.076910455e-03, smax(bfloat16) = 9.687500000e-01, smax(gold) = 9.698269105e-01
Vec 6: softmax sum = 1.001401120e+00, max_diff = 4.619055247e-02, smax(bfloat16) = 4.960937500e-01, smax(gold) = 5.422843025e-01
Vec 7: softmax sum = 1.001481056e+00, max_diff = 1.515826420e-03, smax(bfloat16) = 1.000000000e+00, smax(gold) = 9.984841736e-01
Vec 8: softmax sum = 1.000998256e+00, max_diff = 4.494633422e-02, smax(bfloat16) = 3.535156250e-01, smax(gold) = 3.085692908e-01
Vec 9: softmax sum = 1.001348512e+00, max_diff = 1.018504023e-02, smax(bfloat16) = 8.593750000e-02, smax(gold) = 7.575245977e-02
Vec 10: softmax sum = 9.990525250e-01, max_diff = 6.746567843e-03, smax(bfloat16) = 8.300781250e-02, smax(gold) = 8.975438034e-02

```

Figure 10 - Simulation Verification

The output provides two different indications of simulation performance. The first is an indication of whether the simulation output matched corresponding test vectors. There is one comparison for each softmax function evaluated. The script compares `int16` values, which represent bfloat16 floating-point softmax function values. Since there may be variation when comparing floating point results, the comparison allows for mismatch in the two least significant mantissa bits of the floating-point number. The number of allowed LSB mismatches may be adjusted in the MATLAB script.

The second comparison indicates the maximum difference between AI Engine simulation results and double precision floating-point results generated by MATLAB processing. For each softmax computation, the maximum error is specified along with the two values compared. Also shown for each softmax computation is the summation of all terms, which should ideally be 1. The differences demonstrated suggest that bfloat16 computation is providing the expected 2 to 3 decimal digits of precision.

## References

- [1]: P. Blanchard, D. J. Higham, N. J. Higham, Accurately computing the log-sum-exp and softmax functions, IMA Journal of Numerical Analysis, Volume 41, Issue 4, October 2021, Pages 2311–2330.
- [2]: N. N. Schraudolph, "A fast, compact approximation of the exponential function," *Neural Comput.*, vol. 12, no. 9, pp. 2009-2012, 2000.
- [3]: A. C. I. Malossi, Y. Ineichen, C. Bekas and A. Curioni "Fast exponential computation on SIMD architectures," HiPEAC 2015 - 1st Workshop On Approximate Computing (WAPCO), Amsterdam, NL, 2015, doi: 10.13140/2.1.4362.3207.

## Support

GitHub issues will be used for tracking requests and bugs. For questions, go to [support.xilinx.com](https://support.xilinx.com).

## License

Components: xilinx-images

images in the documentation

Components: xilinx-files

## The MIT License (MIT)

Copyright (c) 2024 Advanced Micro Devices, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

© Copyright 2024 Advanced Micro Devices, Inc.