Journal of Systems Architecture 111 (2020) 101783

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Experiment and enabled flow for GPGPU-Sim simulators with fixed-point instructions

Chao-Lin Lee[a], Min-Yih Hsu[a], Bing-Sung Lu[a], Ming-Yu Hung[b], Jenq-Kuen Lee[a,*]

[a] Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
[b] MediaTek Inc, Hsinchu, Taiwan

ARTICLE INFO

Keywords:
Low-power numerical
GPGPU
Simulator

ABSTRACT

Currently, GPGPU-Sim has become an important vehicle for academic architecture research. It is a cycle-accurate simulator that models the contemporary graphics processing unit. Machine learning has now been widely used in various applications such as self-driving car, mobile devices, and medication. With the popularity of mobile devices, mobile vendors are interested on porting machine learning or deep learning applications from computers to mobile devices. Google has developed TensorFlow Lite and Android NNAPI for mobile and embedded devices. Since machine learning and deep learning are very computationally intensive, the energy consumption has become a serious problem in mobile devices. Moreover, Moore's law cannot last forever. Hence, the performance of the mobile device and computers such as desktops or servers will have limited enhancements in the foreseeable future. Therefore, the performance and the energy consumption are two issues of great concern. In this paper, we proposed a new data type, fixed-point, which is a low-power numerical data type that can reduce energy consumption and enhance performance in machine learning applications. We implemented the fixed-point instructions in the GPGPU-Sim simulator and observed the energy consumption and performance. Our evaluation demonstrates that by using the fixed-point instructions, the proposed design exhibits improved energy savings. Our experiment indicate that the use of fixed-point data type saves at least 14% of total GPU energy consumption than floating-point data type.

1. Introduction

Artificial intelligence and machine learning is now considered as one of the biggest computer science revolution in modern technology impacting our lives. Hence, how to improve performance in machine learning become an important topic in current technology.

In this paper, we aim to develop a low-power numerical execution environment for machine learning. Therefore, we establish the fixed-point instruction architecture, which contains fixed-point conversion and fixed-point binary operations. With this design of instructions, we can run fixed-point instructions within CUDA [1] or OpenCL [2–6] kernels. In our experiment, we focus on the NNEF DNN inference model running on OpenCL framework. OpenCL kernels are compiled with SPIR-V compiler to enhance portability and then, compile to cuda backend.

NNEF(Neural network exchange format) is proposed by Khronos Group. The goal of NNEF is to create a unified network description format to transfer trained networks and facilitate deployment of networks from frameworks to inference engines. SPIR-V(Standard Portable Intermediate Representation - V) is the open standard intermediate representation which supports OpenCL and Vulkan. SPIR-V can be an intermediate representation in compiler flow across multiple backends from vendors. With SPIR-V, developers can use common frontend compiler, and therefore, improving kernel portability and reliability in heterogenous computing.

Our design of fixed-point instructions is based on integer instructions, which can be considered as the fixed-point data type with the binary point position at zero. We also designed the fixed-point operation with two factors, bit width and binary point positions. There are 8/16/32 bit widths in our design. The programmers or vendors may have concerns with the bit width, which will affect the data precision and the power consumption. The binary point position will mainly affect the data precision. Programmers may need more bits in the fractional bit in regard to certain triangular math functions.

In our proposed flow, we first compile the OpenCL kernel by clang, which is the front-end compiler of LLVM. Then, the bit code will be compiled to SPIR-V intermediate representation by LLVM-SPIRV. Next, we compile from bit code to the PTX assembly with the llc compiler. Finally, we can run the OpenCL kernel code with fixed-point instructions on GPGPU-Sim.

The experiment results demonstrate that our design of fixed-point instructions is capable of saving energy and improving performance with

* Corresponding author.
E-mail address: jklee@cs.nthu.edu.tw (J.-K. Lee).

https://doi.org/10.1016/j.sysarc.2020.101783
Received 28 February 2019; Received in revised form 8 March 2020; Accepted 13 April 2020
Available online 29 April 2020

limited precision loss. By using our fixed-point instructions, a kernel may achieve 14% energy consumption reduction.

This paper makes the following contributions:

- We provide possible OpenCL extension with fixed-point feature and linguistics.
- This work presents the design of fixed-point instructions in GPGPU-Sim.
- This work proposes the enabled flow for fixed-point simulations.
- This work demonstrates the use case of fixed-point instructions in a DNN with NNEF, which has data precision loss tolerance.
- This work demonstrates that with the proposed design of fixed-point instructions, the total energy consumption can be reduced by approximately 14%.

The remainder of this article is organized as follows. Section 2 introduces the motivation and background of the fixed-point data type and GPGPU-Sim simulator. Section 3 describes our design of fixed-point representation and the support of the fixed-point instruction set on the GPGPU-Sim simulator. In Section 3.4, we provide our power evaluation methodology. Section 4 shows the experimental result of the DNN framework with an OpenCL fixed-point type comparing to a floating-point type. Finally, Section 6 concludes our results.

2. Background and motivation

This section provides background information about the GPGPU-Sim architecture and the motivation for this study as well as the fixed-point enabled flow, which demonstrates source compilation from OpenCL host to kernel.

2.1. GPGPU-Sim simulator

GPGPU-Sim [7] is a GPU simulator and contains a functional model, timing model and power model. The functional model simulates the Parallel Thread eXecution (PTX) [8] instruction set, which is used by NVIDIA GPU. PTX is a scalar low-level, data-parallel virtual ISA defined by NVIDIA. The timing model models microarchitecture timing related to GPU computation. The power model is modeled by GPUWattch [9], which estimates the power consumed by the GPU according to the timing behavior.

GPGPU-Sim is a cycle-accurate GPGPU simulator produced by Admodt's team at UBC University. Many academic papers have been published with GPGPU-Sim as a hardware simulator [10]. GPGPU-Sim is an NVIDIA PTX/SASS simulator that can be used to simulate a general-purpose compute kernel and pass the cycle-by-cycle arguments to GPUWattch, which is the power model of GPGPU-Sim. GPUWattch is an energy model based on MCPAT, which is an integrated power, area, and timing modeling framework for multithread and multicore architectures. Fig. 1 shows the relation between functional model, timing model and power model. PTX assembly is read by OpenCL/CUDA API interface. Hardware model passes the instruction to functional model to emulate the instruction set. Each executed instruction will passes instruction latency and active count to timing model. Timing model calculates the instruction cycle and cycle-by-cycle performance counters and passes them to power model. GPGPU-Sim calculates all the components' performance counters and passes them to GPUWATTCH. Power model then computes the dynamic power and static power to the runtime power statistics. The power statistics then passed to the GPGPU-Sim to complete feedback-driven optimizations.

2.2. NNEF

Deep learning frameworks have been burgeoning in recent years, such as Caffe, Torch, TensorFlow, and PyTorch. Each framework has its model format to trained networks. The chip vendors need to develop compatible reference engines to different frameworks or vendors need to develop specific reference engines to reference the trained data from the specific deep learning framework.

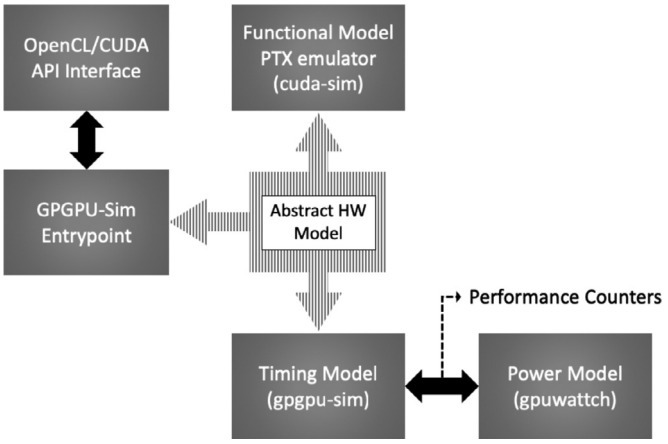

Fig. 1. GPGPU-sim Overview.

Neural network exchange format (NNEF) transforms different network frameworks to unified network structures with clear semantics. The advantages of NNEF is designed to import and export across NN frameworks and the inference engine. Moreover, NNEF is a human readable file format; in this way, users can have the overview of this NN model or further optimized once users read NNEF format. While the shortcomings of the NNEF is not much framework use this file format, as it did not come up earlier than other file format, such as ONNX.

2.3. Motivation

The deep neural network is a branch of machine learning algorithms and contains many complex components. Moreover, DNN is computationally demanding and needs a large memory footprint. However, DNN has an attribute that it is very tolerant to approximation, which creates an opportunity to improve performance and save energy consumption via approximate computing.

The floating-point data type is often used in many applications because it is a precision-oriented data type. However, the floating-point data type needs a large memory footprint and expensive computing power, especially in regard to DNN, for which we need to perform much training and testing. If we use floating-point as the data type, the memory access will have high energy consumption even though we preserve better precision.

The above considerations indicate that the energy consumption can be greatly reduced in DNN runtime if we use fixed-point data type rather than floating-point data type.

In this paper, we designed the fixed-point data type and its instruction implementations on GPGPU-Sim.

3. Fixed-Point design and implementation

This section describes the design and implementation of fixed-point instructions on the GPGPU-Sim simulator. Our implementation on GPGPU-Sim is available as an open source package.[1] We implemented the fixed-point instructions in the functional model and modeled the fixed-point energy consumption in GPUWattch, which will be described in Section 3.4.

3.1. Fixed-Point enabled flow

Fig. 2 shows the OpenCL compilation flow of our design from source to binary. We generated PTX code with the LLVM integrated with SPIR-

[1] https://github.com/jaspercllee/GPGPU-Sim-Fxp

```
4 ConvertFToFxp 60 61 54
4 ConvertFToFxp 60 62 59
4 ConvertFxpToFxp 35 63 61
4 ConvertFxpToFxp 35 64 62
5 FxpMul 35 65 63 64
4 ConvertFxpToFxp 35 66 47
4 ConvertFxpToFxp 35 67 65
```

Listing 1. SPIR-V Extension for Fixed-Point Instruction.

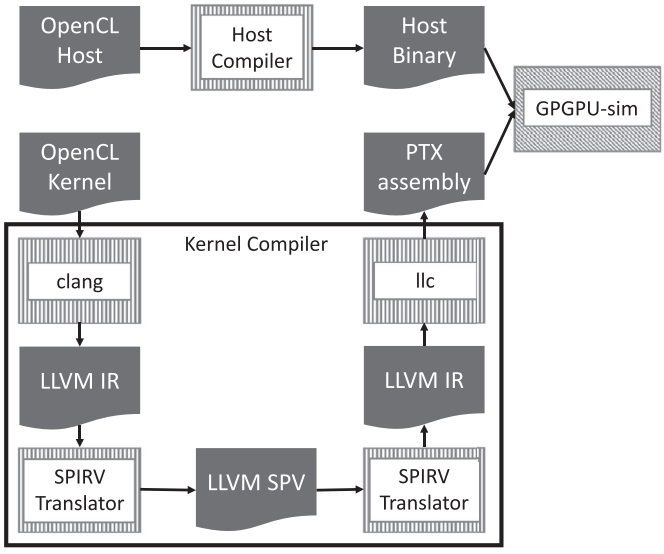

Fig. 2. Fixed-Point Enabled Flow with SPIR-V.

```
float atmp = a[gid];
float btmp = b[gid];
float ctmp = 0.0;
fixedpoint<16,-8> a_fxp;
fixedpoint<16,-5> b_fxp;
fixedpoint<16,-8> c_fxp;
a_fxp = convert_cast<fixedpoint<16,-8>>(atmp);
b_fxp = convert_cast<fixedpoint<16,-5>>(btmp);
c_fxp = convert_cast<fixedpoint<16,-8>>(ctmp);
c_fxp = a_fxp - b_fxp;
```

Listing 2. Sample Code for Fixed-point.

V extension, which is the kernel compiler for the OpenCL program. First, the LLVM IR was generated by clang. SPIR-V is an intermediate representation for compute kernels. We used LLVM SPIR-V compiler to transform from the LLVM IR to LLVM SPV. Listing 1 shows part of the SPIR-V IR code. We can see that we have designed the conversions and operations for fixed-point instructions in SPIR-V extension. Then, we transform the SPIR-V IR back to LLVM IR. The LLVM back-end llc will generate the PTX assembly from IR.

In our revised flow, LLVM back-end will generate the PTX assembly with fixed-point instructions. Fixed-point instructions in PTX assembly contains mathematical computations and type conversions between fixed-point and floating-point. We can now execute the OpenCL host binary with OpenCL kernel in which the data type is fixed-point on the modified GPGPU-Sim simulator.

3.2. Fixed-Point representation and example

The fixed-point type is a format for representing numbers in computer science. It is a data type that uses integers and integer arithmetic to approximate real numbers. This data type needs a smaller memory footprint than the floating-point data type and can perform computations without floating-point support in hardware.

We now describe fixed-point representations. Our fixed-point representation is based on a C++ fixed-point proposal [11] with some specification restrictions. A fixed-point data type is a fixed number of bit width with a specific binary point position. We can express the equivalent value of a fixed-point variable in Eq. 1.

$$N * pow(2, Exp) \tag{1}$$

where the bit width of N and value of Exp are equal to the *Width* and *Exponent* template arguments in the fixed-point type declaration, respectively. Thus, for example, Eq. (2) shows variables for a fixed-point type.

$$fixedpoint < 16, -4 > \tag{2}$$

Its value is effectively equal to the Eq. (3)

$$S16 * pow(2, -4) \tag{3}$$

Fixed-point data types can be either signed or unsigned. Signed binary fixed-point numbers are typically represented in one of three ways: sign/magnitude, one's complement, and two's complement. In our design, we used the two's complement as our representation of signed fixed-point numbers.

We now describe the fixed-point advantages. Fixed-point numbers are closely related to an integer representation. The two numbers only differ in the position of the binary point. In other words, integer representation can be considered as a fixed-point number where the binary point is at the zero position. Therefore, the chip, which can support integer numbers, can easily implement the fixed-point data type as well. Moreover, fixed-point is less complicated than floating-point in terms of the hardware design of the logic circuits. Thus, the fixed-point chip size is smaller with less power consumption when compared with the floating-point hardware. Fixed-point calculations also require less memory and less processor time than floating-point computing, greatly enhancing the performance of the program. Currently, the mobile computing has become more and more important because it is closely related to our lives, and deep learning applications have been increasingly ported to mobile devices. There are, however, power concerns since the battery of a mobile devices is limited. Therefore, we can use the fixed-point type as the data type to save power consumption.

One of the disadvantage of fixed-point representations is the loss of precision when compared to floating-point representations. For example, in a fixed-point < 16,-1 > data type, our fractional part is only precise to 0.5. We cannot represent a number such as 0.75. Rather, we can represent 0.75 with fixed-point < 16,-2 > , but then, we lose precision on the integer part. Another disadvantage of fixed-point representations is not compliant with IEEE-754 standard; hence, we need to convert to same representation before different representation is computed, such as the multiplication of floating-point and fixed-point representation.

We now demonstrate the example of a fixed-point type representation. In the demo code below, we convert floating-point to a 16-bit fixed-point and return the fixed-point subtraction in the end Figure Listing 2.

3.3. Functional model design

In our work, we revise the GPGPU-Sim functional model, which is based on the Nvidia PTX instruction set. We add fixed-point type conversion and fixed-point math instruction set in the functional model.

The supported fixed-point instructions in our work are as follows. Instructions can effectively classified to two parts, conversion and math.

We design cvt_fxp2fp, cvt_fp2fxp conversion instruction to convert between fixed-point and floating-point representation. Moreover, we consider that we will confront the computation with two different fixed-point type(different width and binary point), so we design cvt_fxp2fxp for this case. We also design math instructions, such as fxp_add, fxp_mul, fxp_sub, to satisfy basic computational demands.

Algorithm 1 is the instruction implementation of the type conversion between floating-point and fixed-point data type. Once the PTX parser meets the convert instruction in PTX assembly, it will invoke the conversion function in the GPGPU-Sim functional model. The input of this conversion function depends on the source value and *ConvertInfo* of destination type.

ALGORITHM 1: InvokeFxpConversion(*SV, ConvertInfo*).

Input: SV are the source value ready for conversion.
Input: $ConvertInfo$ are destination type information.
Result: DV are destination value convert from SV

```
1  begin
2    Extract source and destination type from ConvertInfo
3    Extract Significand, exponent and mantissa bit from SV
4    switch Source Type do
5      case FloatingPoint
6        if dest type is FixedPoint then
7          exponent ← dest. exponent - source exponent
8          width ← destination width
9          sign ← is_sign
10         DV ← RoundingMode(width, exponent, sign)
11         if Significand = 1 then
12           two's complement of DV
13         end
14         Write DV to thread
15       end
16     end
17     case FixedPoint
18       if dest type is FloatingPoint then
19         DV ← SV * pow(2, Exponent)
20         Write DV to thread
21       end
22     end
23   endsw
24 end
```

If the source type is fixed-point, it will invoke ConvertFxp2Float or ConvertFxp2Fxp functions. If the source type of the conversion instruction is floating-point and the destination type is fixed-point, it will invoke the ConvertFloat2Fxp instruction. In our implementation, we first extract source and destination data type from *ConvertInfo*. If the source data type is floating-point, significand bit, exponent and mantissa bit will be extract from source value. The converter will convert with rounding according to the *ConvertInfo*, significand bit, exponent, width. Our implementation support rounding modes which is listed in Table 1. Users can rounds the fixed-point numbers with these four strategies when doing type conversions. In addition, if the floating-point value is negative, we do the two's complement of the fixed-point value. Then, we write the destination value to threads.

Table 1
Our supported rounding modes .

Rounding Mode	Description
RTE	round to nearest even
RTZ	round to zero
RTP	round to positive infinity
RTN	round to negative infinity

If the source data type is fixed-point and the destination data type is floating-point, it will invoke the ConvertFXP2Float instruction. In this instruction, we first extract the information such as value, exponent, and data type from the source value. The function will do a signed check; then, the source value will multiply a power of 2 with the source exponent and return the value to the destination value. Finally, we write the destination value to threads.

To successfully run a program in GPGPU-Sim, we also design binary math instructions in the functional models. We design addition, subtraction, and multiplication instructions. Algorithm 2 shows the implementation of the fixed-point math functions. Once a PTX parser meets the binary math instructions, it will invoke the fixed-point binary function; it extracts LHS and RHS arguments, such as value, width, and exponent information, and fetches the operand from arguments.

ALGORITHM 2: InvokeFxpMath(*LV, RV*).

Input: LV is the LHS value in Fixed-Point type.
Input: RV is the RHS value in Fixed-Point type.
Result: DV are result value after math computation.

```
1  begin
2    Extract LHS and RHS width and exponent information
3    switch Operator do
4      case Addition
5        exponent ← Min(LHSexponent, RHSexponent)
6        width ← Max(LHSwidth, RHSwidth)
7        DV ← LHSvalue + RHSvalue
8      end
9      case Substraction
10       exponent ← Min(LHSexponent, RHSexponent)
11       width ← Max(LHSwidth, RHSwidth)
12       DV ← LHSvalue − RHSvalue
13     end
14     case Multiplication
15       exponent ← LHSexponent + RHSexponent
16       width ← LHSwidth + RHSwidth
17       DV ← LHSvalue * RHSvalue
18     end
19   endsw
20 end
```

When the binary addition instruction is invoked, we compare the LHS and RHS exponents and choose the minimal to be assigned to the return exponent. The return width is assigned by the maximum of LHS width and RHS width. The return value is calculated by the addition of LHS value and RHS value. When the binary subtraction instruction is invoked, we compare the LHS and RHS exponent and choose the minimal to be assigned to the return exponent. The return width is assigned by the maximum of the LHS width and RHS width. The return value is calculated as the difference between the LHS and RHS values. When the binary multiplication instruction is invoked, the return exponent is assigned by the addition of the LHS and RHS exponents. The return width is assigned by the addition of LHS and RHS widths. The return value is then calculated by the multiplication of the LHS and RHS values.

3.4. Power model for low power numerical

The proposed power model is designed according to the instruction power. First, there are two types of instructions in our current design, fixed-point conversion and fixed-point binary operations. The power of fixed-point conversion is modeled as the power of shift instructions because the conversion instructions are designed to shift the binary point to an assigned position. The power of fixed-point binary operation is modeled as an integer operation because our fixed-point implementations are based on integer operations. These two types of instructions

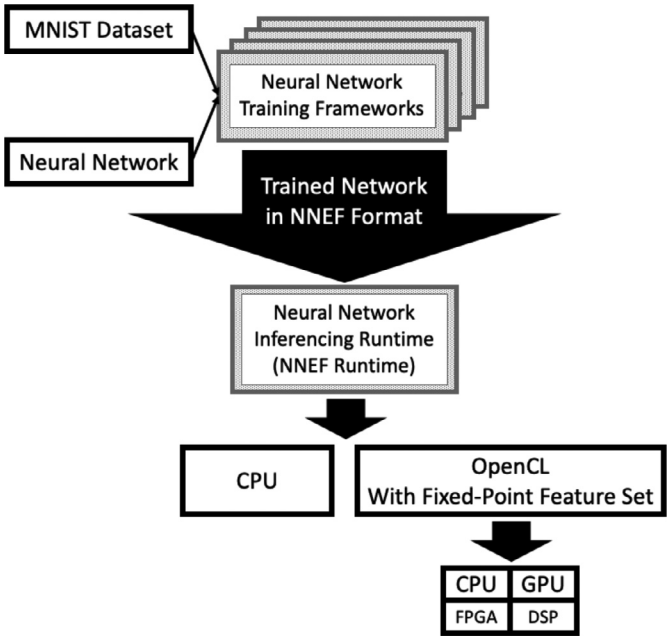

Fig. 3. NNEF Flow.

will eliminate the power consumption of the floating-point unit but will result in an increase in the power consumption of the integer ALU.

4. Experiments

In this paper, we set up a MNIST NNEF model with inference runtime as our benchmark. We first describe an overview of NNEF and then demonstrate our experimental environments. In the last part of this section, we present the experimental results.

4.1. Experimental methodology - NNEF

In our experiments, we devised the inference flow of the MNIST [12] to the NNEF framework.

Our detailed MNIST training settings are as follows. The training set contains 60,000 sample, and the test set contains 10,000 sample. We trained the MNIST with 1 epoch which means that the dataset is passed forward and backward through the neural network once. We didn't divide data sets into parts, so our batch size is 1. Thus, with 60,000 samples for training set, we need to perform 60,000 iterations during training phase. According to the above configurations, we achieved 90% inference accuracy with floating-point data types.

Fig. 3 shows our experimental flow in NNEF. We import neural network trained data to NNEF and then export NNEF to neural network inference runtime. Finally, we assign workloads to the CPU and OpenCL.

We come up with a NNEF file for the MNIST neural network. Listing 3 shows the neural network for MNIST in NNEF.

We assign the NNEF matrix multiplication node to run the OpenCL framework with fixed-point instructions. Listing 4 shows the matrix multiplication OpenCL kernel program. In this kernel program, we obtain the kernel arguments from the host program. First, we convert the kernel arguments from floating-point to fixed-point type. Second, we perform matrix multiplication with fixed-point MUL and ADD instructions. Lastly, we convert the result with the fixed-point type to the floating-point type to the return argument.

In our experiment, the power consumption is evaluated by the GPGPU-Sim simulator. Table 2 shows the simulation parameters of the GPGPU-Sim. The shader core frequency is 1.4 GHz, and we use 16 streaming multiprocessors (SMs). A warp is a set of 32 threads within a thread block that execute the same instruction. The SIMD width is 32,

```
graph mnist( x ) -> ( output )
{
    x = external(shape = [1, 784])
    weight1 = variable(shape=[784, 100],label='weight1')
    weight2 = variable(shape=[100, 10],label='weight2')
    bias1 = variable(shape=[1, 100],label='bias1')
    bias2 = variable(shape=[1, 10],label='bias2')
    matmul1 = matmul(x, weight1)
    add1 = add(matmul1, bias1)
    sigmoid1 = sigmoid(add1)
    matmul2 = matmul(sigmoid1, weight2)
    add2 = add(matmul2, bias2)
    output = sigmoid(add2)
}
```

Listing 3. MNIST Neural Network NNEF.

```
__kernel void matrix_mult(
      const int Ndim,
      const int Mdim,
      const int Pdim,
      __global const float* A,
      __global const float* B,
      __global float* C)
{
   int i = get_global_id(0);
   int j = get_global_id(1);
   C[i*Mdim + j] = 0;

   int k, x;
   float tmp = 0.0;
   float tmpA = 0.0;
   float tmpB = 0.0;

   fixed_point<8,-2> A_fxp;
   fixed_point<8,-3> B_fxp;
   fixed_point<8,-6> tmp_fxp =
       convert_cast<fixed_point<8,-6>>(tmp);

   if ((i < Ndim) && (j < Mdim)) {
      for (k = 0; k < Pdim; k++)
      {
         tmpA = A[i*Pdim + k];
         tmpB = B[k*Mdim + j];
         A_fxp =
             convert_cast<fixed_point<8,-2>>(tmpA);
         B_fxp =
             convert_cast<fixed_point<8,-3>>(tmpB);
         tmp_fxp = tmp_fxp + A_fxp * B_fxp;
      }
      C[i*Mdim + j] = convert_cast<float>(tmp_fxp);
   }
}
```

Listing 4. NNEF node - OpenCL Matmul with Fixed-Point Instructions.

and the register file has 32,768 bytes. The integer and floating-point power consumption per instruction [13] are also listed in Table 2.

4.2. Experimental results

In this paper, we simulated a fixed-point instruction power as an integer instruction power. The baseline benchmark is the floating-point type of the matrix multiplication in the MNIST program. Fig. 4 shows the

Table 2
Simulation parameters .

Parameter	Value
GPU Architecture	Fermi
Execution Model	In-Order
Number of SM Cores	16
Number of SP per SM Core	32
Max Number of Warps per SM	48
Int8/Int16/Int32 addition	0.03/0.05/0.1 pJ
Int8/Int32 multiplication	0.2/3.1 pJ
Float16/Float32 multiplication	1.1/3.7 pJ

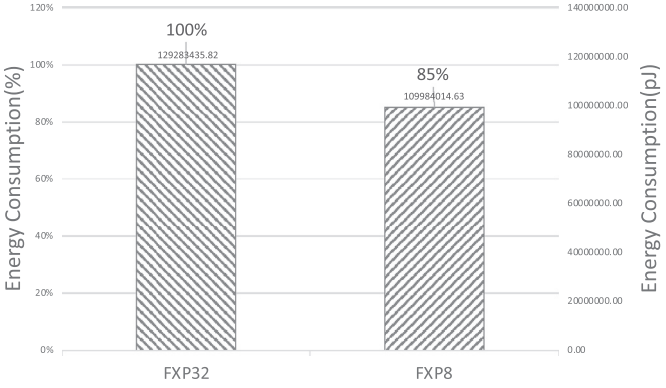

Fig. 4. Normalized Energy Consumption of DRAM Component.

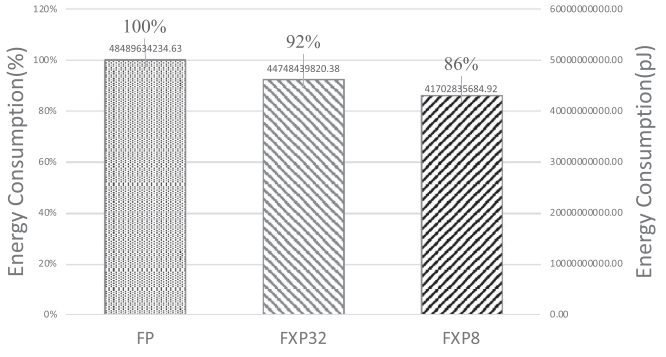

Fig. 5. Normalized Energy Consumption of Total GPU.

power consumption of the DRAM component reported by GPUWattch, while Fig. 5 shows the total energy consumption of the GPU.

Since the data width is reduced, the fixed-point data type can use a smaller memory footprint than the floating-point data type. Fig. 4 shows the energy consumption of DRAM access. The program with 8-bit fixed-point instructions has 15% less energy consumption than that with 32-bit fixed-point instructions.

We model the fixed-point instruction power according to related research [13]. Fig. 5 shows that the 32-bit and 8-bit fixed-point instructions have energy consumption savings that are 8% and 14% of that relative to the floating-point instructions.

Fig. 6 shows that the 32-bit and 8-bit fixed-point instructions compare to floating-point instructions. In our experiments, we run the MNIST with 50 different image recognition. Results show that the 32-bit fixed-point with the binary point is set to 16 has 82% accuracy and the 8-bit fixed-point with the binary point is set to 4 has 74% accuracy. Users need to consider the precision-power trade-off.

The estimation of GPU energy consumption contains numerous functional units. In our experiments, the most significant changes are in DRAM, Integer ALU, FPU (floating-point unit). The width of fixed-point

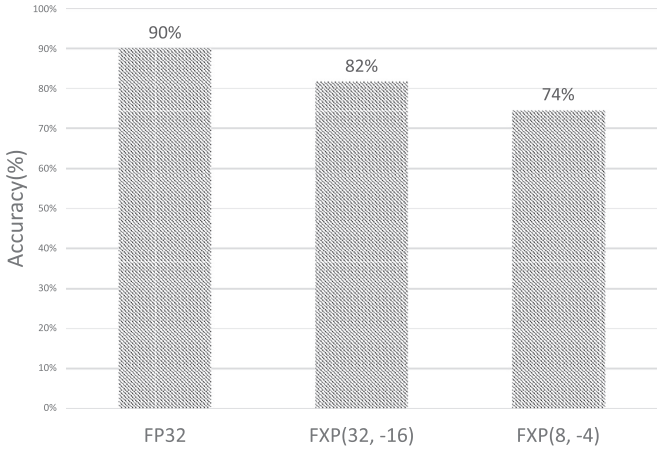

Fig. 6. Accuracy Comparison in MNIST.

representation is shorter than floating-point representation, so the energy consumption of DRAM access will be reduced. The energy consumption of FPU is also reduced because we convert floating-point to fixed-point representation. However, the energy consumption of Integer ALU is increased because our fixed-point is simulated by integer type. Therefore, when fixed-point data type usage increased, Integer ALU will also increase.

5. Related work and discussions

In this section, we will discuss some researches related to our works that also focus on tolerating lower-precision arithmetic without degrading accuracy.

Intel has published their initial white paper on BF16/ BFloat16 [14], a new floating point format to be supported by Intel processors. The BFloat16 format has one sign bit, eight exponent bits, and seven mantissa bits. The BFloat16 tensors are taken as input to the core compute kernels represented as General Matrix Multiply operations. It is then forwarded to the FP32 tensors as output. The advantage of BFloat16 is that one can effectively convert floating-point to BFloat16 by extract first 16 bit floating-point. However, BFloat16 has less accuracy than float16. Thus, if users need to have higher accuracy, our work for fixed-point representation is a better choice.

One of the typical data compression methods is quantization, which trade-off precision accuracy and power consumption. There are two modes of quantization: linear mode and logarithm mode. Linear mode divided into symmetric approaches and asymmetric approaches. Asymmetric quantization [15] maps to the actual floating-point value range to a smaller integer value range with scale factor, and uses zero point as bias mapping to actual zero. Symmetric quantization acquires the maximum absolute value between minimum or maximum of actual floating point to map a smaller integer value range, so zero points are not used as bias corresponding to zero. The advantages of the asymmetric method will be able to fully utilize the quantized range. The advantage of the symmetric method is that the hardware implementation is simpler. The logarithm mode [16,17] is mapping the original values to the closest power-of-2 value. Fixed-point representation is a bit level implementation, so it has advantages that the hardware is easier to design. If one wants to have a similar design to fixed-point, logarithm quantization will be a solution.

Compared to quantized neural networks, fixed-point is similar to floating-point with lower overhead, and the accuracy of fixed-point can be customized by user, and can even exceed the predetermined width. In our work, OpenCL with Fixed-point representation is compile-time heuristic, which can acquire the fixed-point information at compile time, and do further optimize. While our work is converting floating-point

data type to fixed-point data type with extracting original floating-point value and shifting the user-set binary point position. Detailed design and implementation are discussed in Section 3.

6. Conclusions

In this paper, we proposed a fixed-point design and the implementation of the fixed-point data type on the GPGPU-Sim simulator. With the evaluation of the energy methodology, we can model energy consumption with a fixed-point data type program. Our implementation reduced the total GPU energy consumption by 14% by using the fixed-point computation simulated by our revised GPGPU-Sim model.

Declaration of Competing Interests

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

Acknowledgements

This work was supported in part by MOST of Taiwan and MediaTek.

References

[1] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with cuda, in: ACM SIGGRAPH 2008 classes, ACM, 2008, p. 16.
[2] J.E. Stone, D. Gohara, G. Shi, Opencl: a parallel programming standard for heterogeneous computing systems, Comput. Sci. Eng. 12 (3) (2010) 66–73.
[3] S.-C. Wang, L.-C. Kan, C.-L. Lee, Y.-S. Hwang, J.-K. Lee, Architecture and compiler support for gpus using energy-efficient affine register files, ACM Trans. Design Automat. Electron. Systems. (TODAES) 23 (2) (2017) 18.
[4] Y.-M. Chang, S.-C. Wang, C.-C. Yang, Y.-S. Hwang, J.-K. Lee, Enabling pocl-based runtime frameworks on the hsa for opencl 2.0 support, J. Syst. Archit. 81 (2017) 71–82.
[5] J.-J. Li, C.-B. Kuan, T.-Y. Wu, J.K. Lee, Enabling an opencl compiler for embedded multicore dsp systems, in: Parallel Processing Workshops (ICPPW), 2012 41st International Conference on, IEEE, 2012, pp. 545–552.
[6] K.-M. Cheng, C.-Y. Lin, Y.-C. Chen, T.-F. Su, S.-H. Lai, J.-K. Lee, Design of vehicle detection methods with opencl programming on multi-core systems, in: Embedded Systems for Real-time Multimedia (ESTIMedia), 2013 IEEE 11th Symposium on, IEEE, 2013, pp. 88–95.
[7] T.M. Aamodt, W.W. Fung, I. Singh, A. El-Shafiey, J. Kwa, T. Hetherington, A. Gubran, A. Boktor, T. Rogers, A. Bakhoda, et al., Gpgpu-sim 3. x manual, 2012, (????).
[8] N. Compute, Ptx: parallel thread execution isa version 2.3, Dostopno Na: Http://Developer. Download. Nvidia. Com/Compute/Cuda 3 (2010).
[9] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N.S. Kim, T.M. Aamodt, V.J. Reddi, Gpuwattch: enabling energy optimizations in gpgpus, in: ACM SIGARCH Computer Architecture News, 41, ACM, 2013, pp. 487–498.
[10] C.-C. Yang, S.-C. Wang, M.-Y. Hsu, Y.-M. Chang, Y.-S. Hwang, J.-K. Lee, Opencl 2.0 compiler adaptation on llvm for ptx simulators, in: Parallel Processing Workshops (ICPPW), 2017 46th International Conference on, IEEE, 2017, pp. 53–58.
[11] J. McFarlane, M. Wong, Fixed-point real numbers, 2016,.
[12] Y. LeCun, The mnist database of handwritten digits, http://yann. lecun. com/exdb/mnist/ (1998).
[13] M. Horowitz, 1.1 computing's energy problem (and what we can do about it), in: 2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC), IEEE, 2014, pp. 10–14.
[14] BFLOAT16 Hardware Numerics Definition, (https://software.intel.com/en-us/download/bfloat16-hardware-numerics-definition).
[15] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Quantized neural networks: training neural networks with low precision weights and activations, J. Mach. Learning. Res. 18 (1) (2017) 6869–6898.
[16] D. Miyashita, E.H. Lee, B. Murmann, Convolutional neural networks using logarithmic data representation, arXiv preprint arXiv:1603.01025 (2016).
[17] B. McDanel, S.Q. Zhang, H. Kung, X. Dong, Full-stack optimization for accelerating cnns using powers-of-two weights with fpga validation, in: Proceedings of the ACM International Conference on Supercomputing, ACM, 2019, pp. 449–460.

Chao-Lin Lee received the BS degree in management of information systems from National Chung Hsing University in 2011, and the MS degrees in information systems and applications form National Tsing Hua University in 2018. He is currently working toward the PhD degree in Department of Computer Science, National Tsing Hua University, Taiwan. His research interests include parallel programming, compiler optimization and artificial intelligence Framework.

Min-Yih Hsu received the BS degree in computer science from National Tsing Hua University in 2018. He is currently working toward the PhD degree in Department of Computer Science, The University of California, Irvine. His research interests include AI applications, compiler optimization and artificial intelligence Framework.

Bing-Sung Lu received the BS degree in economics from National Chengchi University in 2015, and the MS degrees in computer science form National Tsing Hua University in 2018. His research interests include AI compiler optimization and artificial intelligence Framework.

Ming-Yu Hung, Ph.D., received the BS, MS, Ph.D degrees in computer science from National Tsing Hua University, Taiwan in 2002, 2004 and 2012 respectively. He is now a manager in MediaTek Inc. His research interests include compiler optimization, parallel computing, program analysis and AI frameworks.

Jenq Kuen Lee, Ph.D., received the B.S. degree in computer science from National Taiwan University in 1984. He received the M.S. and Ph.D. degrees in 1991 and 1992, respectively, in computer science from Indiana University. He is now a professor in the Department of Computer Science at National Tsing-Hua University, Taiwan, where he joined the Department in 1992. He was a key member of the team who developed the first version of the C++ language and SIGMA system while at Indiana University. He was a recipient of the most original paper award in ICPP 7 with the paper entitled ata Distribution Analysis and Optimization for Pointer-Based Distributed ProgramsHis supervised Ph.D. student received the distinguished dissertation award as an honorable mention by IICM, 1999. He received an achievement award from MOE of Taiwan for University and Industrial joint research, 2001. He received Google research award 2009. In addition, he is a recipient of Taiwan MOEA economic contribution award (Deep Plow Award), 2010. From Oct. 2015 to Oct. 2018, he participated in the new version of OpenCL proposals with Khronos OpenCL Next DSP Feature Set. His current research interests include optimizing compilers, AI framework compilers, embedded multicore compilers and systems, and computer architectures.