Enable the Flow for GPGPU-Sim Simulators with Fixed-Point Instructions

Chao-Lin Lee
National Tsing Hua University
Hsinchu, Taiwan
clli@pllab.cs.nthu.edu.tw

Min-Yih Hsu
National Tsing Hua University
Hsinchu, Taiwan
myhsu@pllab.cs.nthu.edu.tw

Bing-Sung Lu
National Tsing Hua University
Hsinchu, Taiwan
bslu@pllab.cs.nthu.edu.tw

Jenq-Kuen Lee
National Tsing Hua University
Hsinchu, Taiwan
jklee@cs.nthu.edu.tw

ABSTRACT

GPGPU-Sim nowadays has become an important vehicle for academic architecture research. In the aspect of machine learning, it has now been widely used in various applications, such as auto-drive, mobile device, and medication, etc. As these machine learning applications are power-consuming, which has become a critical issue in the machine learning area. This paper proposes the implementation of fixed-point instructions and enabled flow on GPGPU-Sim to replace floating-point instructions in machine learning applications which is with scalable precision. Preliminary experimental results with our revised GPGPU-Sim models show that this design saves GPU energy consumptions by 11% on average when using 16-bit fixed-point as the data type.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**;

KEYWORDS

Deep Learning, Low-power numerical, GPGPU, Simulator

ACM Reference Format:
Chao-Lin Lee, Min-Yih Hsu, Bing-Sung Lu, and Jenq-Kuen Lee. 2018. Enable the Flow for GPGPU-Sim Simulators with Fixed-Point Instructions. In *ICPP '18 Comp: 47th International Conference on Parallel Processing Companion, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3229710.3229722

1 INTRODUCTION

In recent years, machine learning has played an important role of the computer science revolution impacting our lives. The Deep Neural Network is a branch of machine learning algorithms, which contains many complex components. Moreover, DNN is computationally demanding, and needs large memory footprint. However, DNN has an attribute that it is very tolerant to approximation. It is an opportunity to improve performance and save energy consumption via approximate computing.

Floating-point data type is often used in many applications, due to it is a precision-oriented data types. However, floating-point data type needs large memory footprint and expensive computing power. Especially when it comes to DNN, we need to do much training and testing. If we use floating-point as data type, the memory access will cost many powers, though we preserve better precision.

GPGPU-Sim[1] is a GPU simulator, which contains functional model, timing model and power model. Functional model simulates PTX[4] (Parallel Thread eXecution) instruction set which is used by NVIDIA GPU. PTX is a scalar low-level, data-parallel virtual ISA defined by NVIDIA. Timing model models micro-architecture timing related to GPU compute. Power model is modeled by GPUWattch[5] which estimate power consumed by the GPU according to the timing behavior.

In this paper, we implement fixed-point data type in GPGPU-Sim, which provides a detailed simulation model of modern GPUs running CUDA[8] or OpenCL[2, 3, 6, 9, 10] workloads. Fixed-point is a data type that uses fixed bit width and fixed binary point position. The bit width can be 8, 16, 32 bits, which depends on hardware vendors. The binary point position can be fixed position or arbitrary position, which also depends on hardware vendors. In this paper, we show that the energy consumption is decreased by using fixed-point as data type. The experiments describe the energy consumptions for both fixed-point benchmark and floating-point benchmark on our revised GPGPU-Sim simulator. Our implementation reduced 11% of overall energy consumption of the GPU with fixed-point computation, simulated by our revised GPGPU-Sim model.

The remainder of this article is organized as follows. Section 2 introduces our background information of fixed-point data type and GPGPU-Sim simulator. Section 3 describes our design of fixed-point representation and the support of fixed-point instruction set on GPGPU-Sim simulator. In Section 4, we revised GPUWattch with fixed-point power estimation. Section 5 shows the experiment result of the fixed-point type benchmark comparing to floating-point type. Finally, Section 6 concludes the paper.

ICPP '18 Comp, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6523-9/18/08…$15.00
https://doi.org/10.1145/3229710.3229722

2 BACKGROUND ARCHITECTURES

2.1 GPGPU-Sim Simulator

GPGPU-Sim is a cycle-accurate GPGPU simulator produced by Admodt's team at UBC University. Many academic papers has been published with GPGPU-Sim as a hardware simulator[11]. GPGPU-Sim is a NVDIA PTX/SASS simulator, it can be used to simulate General Purpose compute kernel and pass the cycle by cycle arguments to GPUWattch, which is the power model of GPGPU-Sim. GPUWattch is a energy model based on MCPAT which is an integrated power, area, and timing modeling framework for multi-thread and multi-core architectures. Figure 1 shows the relation between GPGPU-Sim and GPUWattch. GPGPU-Sim calculates all the components' performance counters and passes them to GPUWATTCH. GPUWattch then computes the dynamic power and static power to runtime power statistics. Power statistic is then passed to the GPGPU-Sim to complete feedback-driven optimizations.

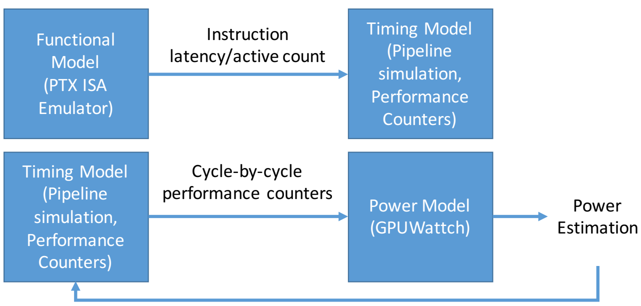

Figure 1: GPGPU-sim Architecture

2.2 Fixed-Point Enabled Flow

Figure 2 shows the flow of our experiment. We generated PTX code with the LLVM integrated with SPIR-V which is the kernel compiler for OpenCL program. First of all, the LLVM IR was generated by clang. LLVM SPIR-V is an intermediate language for graphical shader and compute kernels. We used LLVM SPIR-V to transform from LLVM IR to LLVM SPV. Then we transform the SPIR-V IR back to LLVM IR. LLVM back-end llc will generate PTX assembly from IR. In our revised flow, LLVM back-end will generate PTX assembly with fixed-point instructions. We can now run OpenCL host binary with OpenCL kernel annotated with fixed-point linguistics to PTX assembly on modified GPGPU-Sim simulator.

3 FIXED-POINT DESIGN AND IMPLEMENTATION

This section describes the design and implementation of fixed-point instructions on GPGPU-Sim simulator. We implement the fixed-point instructions in functional model and we also modeled fixed-point energy in GPUWattch, which will be described in Section 4.

3.1 Fixed-Point Representation and Example

Fixed-Point is a format for representing numbers in computer science area. It is a data type that uses integers and integer arithmetic

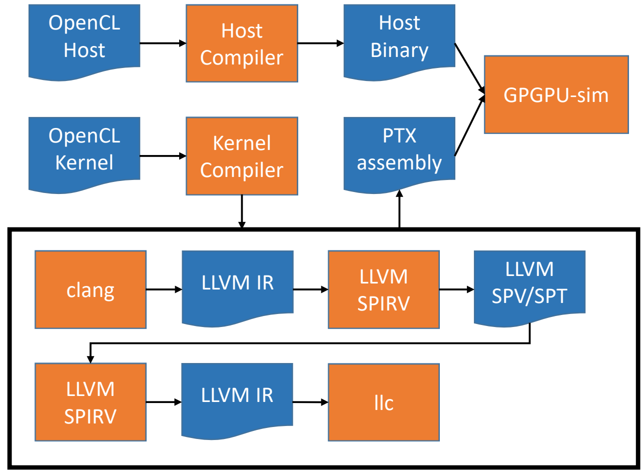

Figure 2: Fixed-Point Enabled Flow

to approximate real numbers. This data type can use less memory footprint than floating-point data type, and perform computations without floating-point support in hardware.

We now describe fixed-point representations. Our fixed-point representation is based on C++ fixed-point proposal[7] with some specification restrictions. A fixed-point data type is a fixed number of bit width with specific binary point position. We can express the equivalent value of a fixed-point variable in equation 1.

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

Where bit width of N and value of Exp equal to $Width$ and $Exponent$ template arguments in fixed-point type declaration, respectively. Thus, for example, equation 2 shows variables for a fixed-point type.

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

Its value is effectively equal to the equation 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 use two's complement as our representation of signed fixed-point numbers.

We now describe fixed-point advantages. Fixed-point numbers are a close relative to integer representation. The two only differs in the position of binary point. In other words, integer representation can be considered as a fixed point numbers, where the binary point is at position 0. Therefore, the chip, which can support integer numbers, can easily implement fixed-point type as well. Moreover, fixed-point is much less complicated than floating-point in terms of the hardware design of the logic circuits. This means that the fixed-point chip size is smaller with less power consumption when compared with floating-point hardware. Fixed-point calculations also require less memory and less processor time than floating-point computing. It will greatly enhance the performance of the program. Nowadays, mobile phone has much more functional than ever before, the deep learning application has increasingly porting to mobile device. It is power concerned since the mobile device's battery is limited. Therefore, we can use fixed-point as data type to save power consumptions.

The disadvantage of fixed-point, is the loss of precision when it is compared with floating-point representations. For example, in a fixed-point<16,-1> data type, our fractional part is only precise to 0.5. We can not represent number like 0.75. We can represent 0.75 with fixed-point<16,-2>, but then we lose precision on the integer part.

We now demonstrate the example of fixed-point type representation. In the below demo code, we convert cast floating-point to 16-bit fixed-point and return the fixed-point add in the end.

```
    float atmp = a[gid];
    float btmp = b[gid];
    float ctmp = 0.0;
    fixedpoint<16,-3> a_fxp;
    fixedpoint<16,-5> b_fxp;
    fixedpoint<16,-5> c_fxp;
    a_fxp = convert_cast<fixedpoint<16,-3>>(atmp);
    b_fxp = convert_cast<fixedpoint<16,-5>>(btmp);
    c_fxp = convert_cast<fixedpoint<16,-5>>(ctmp);
    c_fxp = a_fxp + b_fxp;
```

3.2 Functional Model Design

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

Algorithm 1 is the instruction implementation of the fixed-point type converted to other data types. Once the PTX parser meets the convert instruction in PTX assembly, it will invoke the conversion function in the GPGPU-Sim functional model. This conversion function depends on the source and destination type of *convertinfo*.

If the source type is fixed-point, it will invoke ConvertFxp2Float or ConvertFxp2Fxp functions. If the source type is integer, it will invoke ConvertInt2Fxp. If the source type is floating-point, it will invoke ConvertFloat2Fxp.

If the source type of conversion instruction is floating-point, and destination type is fixed-point, it will invoke the ConvertFloat2Fxp instruction. Algorithm 2 shows how we implement floating-point to fixed-point conversion. In this function, we first extract the argument information such as value, exponent, and type. We will convert from floating-point to fixed-point according to fixed-point bit width, and the floating-point value information which extract from the arguments. Besides, if the floating-point value is negative, we do two's complement of fixed-point value. Then we write the destination value to threads.

To achieve running a program in GPGPU-Sim, we also 3design binary math instructions in functional models. We design addition, subtraction, and multiplication instructions. Algorithm 3 shows the implementation of fixed-point math functions. Once a PTX parser meets the binary math instruction, 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 1: InvokeFxpConversion(pI, $thread$, $convertinfo$)

```
Input: pI is the PTX instruction,
       thread is the PTX thread information,
       convertinfo is Fixed-Point convert information
begin
    Extract Argument Info from threads
    Fetch source and destination type from convertinfo
    switch Source Type do
        case FixedPoint do
            if dest type is FloatingPoint then
                ConvertFxp2Float
            end
            else
                if dest type is FixedPoint then
                    ConvertFxp2Fxp
                end
            end
        end
        case Integer do
            if dest type is FixedPoint then
                ConvertInteger2Fxp
            end
        end
        case FloatingPoint do
            if dest type is FixedPoint then
                ConvertFloat2Fxp
            end
        end
        otherwise do
            Conversion Type not support
        end
    end
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 LHS width and RHS width. The return value is calculated by the subtraction of LHS value and RHS value. When the binary multiplication instruction is invoked, the return exponent is assigned by the addition of LHS exponent and RHS exponent. The return width is assigned by the addition of LHS width and RHS width. The return value is calculated by the multiplication of LHS value and RHS value.

4 FIXED-POINT POWER MODEL

Power consumption of fixed-point is less than floating point due to the fixed-point data size can be smaller than floating-point. Fixed-point data type, in our design, can be scaled to 8, 16 and 32 bits width, depending on what data precision programmer needs.

The Algorithm 4 shows how we design fixed-point power consumption in GPUWattch which is the power model of GPGPU-Sim.

ALGORITHM 2: ConvertFloat2Fxp(pI, $thread$, $srcTy$, $destTy$)

```
Input: pI is the PTX instruction,
       thread is the PTX thread information,
       srcTy is the type of source,
       destTy is the type of destination
begin
    Extract argument info from threads
    switch fixed-point width do
        case 8 do
            PerformFxp2Fxp<32, 8>
        end
        case 16 do
            PerformFxp2Fxp<32, 16>
        end
        case 32 do
            PerformFxp2Fxp<32, 32>
        end
    end
    if source value < 0 then
        two's comliment of dest value
    end
    Write destination value to thread
end
```

ALGORITHM 3: InvokeFxpMath(pI, $thread$, $binaryinfo$)

```
Input: pI is the PTX instruction,
       thread is the PTX thread information,
       binaryinfo is Fixed-Point binary math information
begin
    Fetch source operand from threads
    Extract LHS and RHS argument information
    Fetch LHS and RHS Operand
    switch Operator do
        case Addition do
            exponent ←
              Min(LHSexponent, RHSexponent)
            width ← Max(LHSwidth, RHSwidth)
            value ← LHSvalue + RHSvalue
        end
        case Substraction do
            exponent ←
              Min(LHSexponent, RHSexponent)
            width ← Max(LHSwidth, RHSwidth)
            value ← LHSvalue − RHSvalue
        end
        case Multiplication do
            exponent ← LHSexponent + RHSexponent
            width ← LHSwidth + RHSwidth
            value ← LHSvalue * RHSvalue
        end
        otherwise do
            binary math not support
        end
    end
end
```

ALGORITHM 4: PowerModel($PCOUNT$)

```
Input: PCOUNT is Cycle-by-cycle performance counters passed by
       timing model
Global Data: PC ← Cycle-by-cycle performance counters
begin
    if PC ∈ fixed-point instructions then
        switch fxp width do
            case 8 do
                set memory read/write coefficient *= 0.25
            end
            case 16 do
                set memory read/write coefficient *= 0.5
            end
            case 32 do
                set memory read/write coefficient *= 1
            end
        end
    end
    return Cycle-by-cycle Energy consumptions
end
```

In our design, we revise the power model with the width of fixed-point, which is the most significant factor of Power consumption. As mentioned in Section 3, fixed-point calculations require less memory footprint due to less bit of width. The input is the cycle-by-cycle performance counters passed by timing model. If the instructions are fixed-point with 8/16 bit width operation, we reduce the dram access(read/write) hardware coefficient as a quarter/half of floating-point. We will keep read/write coefficient when we use 32-bit fixed-point type because 32-bit fixed-point is the same size as floating-point type. We will then return cycle-by-cycle energy consumptions to GPUWattch. A more accurate model will require to model ALU computation in addition to memory model.

5 EXPERIMENTS

We wrote an OpenCL vector addition program as our experiment benchmark. The program uses a 16-bit fixed-point type as data type. The baseline benchmark is the floating-point type of the vector addition program. Figure 3 shows the energy consumption of DRAM component reported by GPUWattch, whereas Figure 4 shows the total energy consumption of the GPU. As mentioned earlier in Section 3, the fixed-point type has less memory access than floating-point type. As illustrated in Figure 3, the fixed-point type DRAM component saves 43% of floating-point energy consumption.

With less DRAM access, the total GPU energy consumption reduced to 11% of floating-point type. This findings show that by using fixed-point data type, the energy consumption can be reduced comparing to floating-point data type.

6 CONCLUSIONS

In this article, we presented a fixed-point design and the implementation of the fixed-point data type on GPGPU-Sim simulator. We also revised GPUWattch, the power model of GPGPU-Sim. With the revision of GPUWattch , we can model energy consumption with

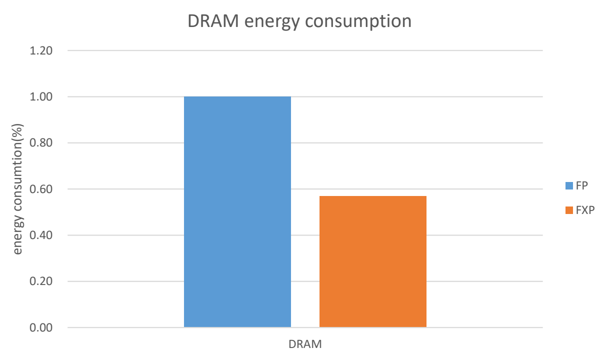

Figure 3: Normalized Energy Consumption of DRAM Component

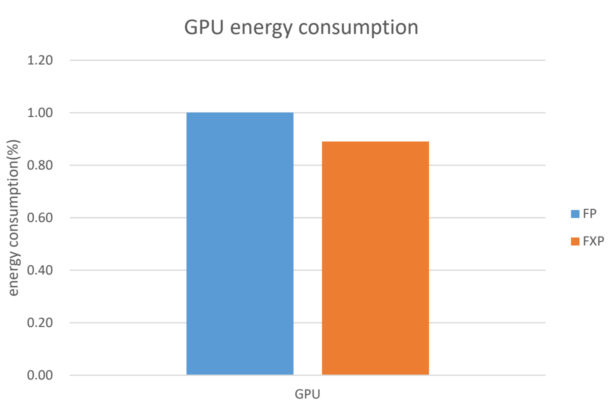

Figure 4: Normalized Energy Consumption of GPU

a fixed-point program. Our implementation reduced 11% of overall energy consumption of the GPU with fixed-point computation, simulated by our revised GPGPU-Sim model.

REFERENCES

[1] Tor M Aamodt, Wilson WL Fung, I Singh, A El-Shafiey, J Kwa, T Hetherington, A Gubran, A Boktor, T Rogers, A Bakhoda, et al. 2012. GPGPU-Sim 3. x manual.

[2] Yuan-Ming Chang, Shao-Chung Wang, Chun-Chieh Yang, Yuan-Shin Hwang, and Jenq-Kuen Lee. 2017. Enabling PoCL-based runtime frameworks on the HSA for OpenCL 2.0 support. *Journal of Systems Architecture* 81 (2017), 71–82.

[3] Kai-Mao Cheng, Cheng-Yen Lin, Yu-Chun Chen, Te-Feng Su, Shang-Hong Lai, and Jenq-Kuen Lee. 2013. 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, 88–95.

[4] NVIDIA Compute. 2010. PTX: Parallel thread execution ISA version 2.3. *Dostopno na: http://developer. download. nvidia. com/compute/cuda* 3 (2010).

[5] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: enabling energy optimizations in GPGPUs. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 487–498.

[6] Jia-Jhe Li, Chi-Bang Kuan, Tung-Yu Wu, and Jenq Kuen Lee. 2012. Enabling an opencl compiler for embedded multicore dsp systems. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 545–552.

[7] John McFarlane and Michael Wong. 2016. Fixed-Point Real Numbers. http://johnmcfarlane.github.io/fixed_point/papers/p0037r3.html

[8] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. In *ACM SIGGRAPH 2008 classes*. ACM, 16.

[9] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66–73.

[10] Shao-Chung Wang, Li-Chen Kan, Chao-Lin Lee, Yuan-Shin Hwang, and Jenq-Kuen Lee. 2017. Architecture and Compiler Support for GPUs Using Energy-Efficient Affine Register Files. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 23, 2 (2017), 18.

[11] Chun-Chieh Yang, Shao-Chung Wang, Min-Yi Hsu, Yuan-Ming Chang, Yuan-Shin Hwang, and Jenq-Kuen Lee. 2017. OpenCL 2.0 Compiler Adaptation on LLVM for PTX Simulators. In *Parallel Processing Workshops (ICPPW), 2017 46th International Conference on*. IEEE, 53–58.