B.Comp. Dissertation

A Variable Precision GPU Architecture for Deep Learning

By

Au Liang Jun

Department of Computer Science

School of Computing

National University of Singapore

2020/2021

B.Comp. Dissertation

A Variable Precision GPU Architecture for Deep Learning

By

Au Liang Jun

Department of Computer Science

School of Computing

National University of Singapore

2020/2021

Project No: H074940
Advisor: Assoc Prof Wong Weng Fai

Deliverables:
Report: 1 Volume
Program: 1 Collection of GitHub Repositories

Abstract

In recent years, deep learning models have grown in complexity, demanding substantial computational resources to train and deploy. The trend has drawn focus to mixed-precision deep learning — the use of narrow number formats in deep learning to trade precision for faster computation and reduced memory and bandwidth consumption. Past research in this domain has largely been constrained to using the formats that are already supported in hardware, or naive strategies that only employ a single low-precision format. In our work, we modify a popular open-source General-Purpose GPU simulator to support arbitrary, low-precision floating-point formats and use it to conduct mixed-precision experiments on deep learning models. We provide a methodology for executing deep learning applications on the variable precision simulator and demonstrate an empirical approach to deriving an optimal mixed-precision strategy for a given model.

Subject Descriptors:

Computing methodologies–Modeling and simulation–Simulation support systems–Simulation tools
Computer systems organization–Architectures–Parallel architectures–Single instruction, multiple data
Computing methodologies–Machine learning

Keywords:

GPGPU simulation, floating-point, mixed-precision deep learning, trans-precision inference

Implementation Software and Hardware:

Ubuntu 16.04, GPGPU-Sim 4.0, CUDA 8.0, MFFR 3.1, GCC 5.4, Python 3.7

Acknowledgement

I extend my heartfelt gratitude to my supervisor Associate Professor Wong Weng Fai, who has not only been a source of invaluable guidance in the face of every challenge, but whose patience and kind words of encouragement have enabled me to keep making progress.

I also thank Associate Professor Ooi Wei Tsang for being generous with his time in evaluating this project and giving feedback.

List of Figures

List of Tables

Table of Contents

Chapter 1

Introduction

1.1 Description

The growth of deep learning in the last decade has been driven by the proliferation of GPUs and the advancements in General-Purpose GPU (GPGPU) computing, which is the use of GPUs beyond the purpose of graphics. GPUs are capable of performing a large number of floating-point operations efficiently, making them highly suitable for the training and execution of neural networks (NNs).

As the size of NNs continues to grow, there has been active research in lowering the precision of floating-point numbers to reduce hardware overhead, leading to faster or lower power inference and training, while retaining model accuracy. This field of study is broadly known as mixed-precision deep learning. The research in this domain is fueled by the introduction of the Tensor Core in GPU manufacturer NVIDIA's Volta microarchitecture. The Tensor Core is a functional unit capable of multiplying half-precision floating-point matrices and accumulating the result into single-precision. Today, automatic mixed-precision is already a feature in popular deep learning frameworks such as PyTorch and Tensorflow, where programmers can simply enable mixed-precision training and inference with the help of Tensor Core bearing GPUs.

1.2 Objectives

Existing research into mixed-precision deep learning has largely centered around utilizing the IEEE 754 half-precision format (FP16), and its variants of similar bit length, in conjunction with the standard IEEE 754 single-precision format (FP32). This is in part because these formats are the only low precision formats supported by commercial hardware like Tensor Cores.

Thus, in this project, we aim to remove the limitations presented by GPU hardware in experimenting with different floating-point formats, as well as explore mixed-precision strategies beyond uniformly enabling half-precision computation. The approach will be to modify an open-source GPGPU simulator to support IEEE 754-style floating-point formats of arbitrary precision, which serves to both:

- model a hypothetical GPU architecture capable of representing and operating on floating-point numbers with arbitrary precision, and
- be used to simulate and experiment on GPU architectures with fixed low precision formats not supported by current commercial hardware.

We then aim to utilize the modified simulator to empirically discover mixed-precision strategies for selected NNs, measuring accuracy and commenting on any efficiency gains.

Finally, we aim to examine the feasibility of using a variable precision GPGPU simulator to perform exploratory research in this domain, highlighting any drawbacks and challenges.

1.3 Outline

Chapter 2 provides a literature review of the existing work in mixed-precision deep learning. We then introduce the GPU simulator that we have chosen and highlight the key aspects relevant to our project in Chapter 3. Chapter 4 walks through the enhancements that we have performed on the simulator to meet our objectives. The deep learning models chosen as benchmarks, along with the modifications made to them, are then described in Chapter 5. Chapter 6 details the experiments and our analysis of the results. We summarize our research in Chapter 7 and present some avenues for future work.

Chapter 2

Literature Review

2.1 Floating-Point in Deep Learning

In recent years, deep learning architectures of increasing complexity have been proposed to boost inference accuracy, at the expense of computations and model storage resources. The cost of training and running these models have therefore posed some challenges in their real-world application. For example, when executing real-time inference on mobile devices or embedded hardware with a limited power budget (Lin, Talathi, & Annapureddy, 2015).

Reducing the computational complexity and memory footprint of NNs while maintaining accuracy is thus a goal sought after by researchers, to yield faster or lower power training and inference (Johnson, 2018). One such strategy is quantization, which is the process of approximating floating-point numbers using low bit-width integers (Wu, Judd, Zhang, Isaev, & Micikevicius, 2020) (Jacob et al., 2017) or bits (Cai, He, Sun, & Vasconcelos, 2017) (Courbariaux, Bengio, & David, 2015). Apart from reducing memory and storage requirements (Courbariaux et al., 2015), this enables bandwidth savings in memory-limited layers, as well as operations to be carried in integer-only arithmetic, leveraging the higher throughput math pipelines provided by many processors. These benefits contribute to faster inference and training (Wu et al., 2020).

An alternative to quantization is the use of low precision floating-point formats to reduce word size. This approach is seen to be easier to work with than integer quantization, as it avoids the need to convert between floating-point and integer, keeping data and operations closer to the original implementation (Johnson, 2018).

The majority of research into using floating-point variants in NNs is limited to the variants provided in CPU/GPU Instruction Set Architectures (ISAs) (Johnson, 2018). Two of the most common formats applied to deep learning are the IEEE 754 FP16 (Micikevicius et al., 2017), and the more recently defined brain float type (Kalamkar et al., 2019).

Format	Sign	Exponent	Significand (Explicit)
FP32	1	8	23
FP16	1	5	10
bfloat16	1	8	7
TensorFloat-32	1	8	10
“FP8”	1	5	2

Table 2.1: Common floating-point formats in deep learning and their constituents.

FP16

As seen in Table 2.1, the FP16 type occupies half the width of the standard IEEE 754 FP32 type. The size of the significand is reduced from 23 to 10, while the exponent width is reduced from 8 to 5. FP16 was the first floating-point data type smaller than single-precision width to be supported in NVIDIA GPUs, beginning from the Pascal microarchitecture (Ho & Wong, 2017). Due to the way FP16 arithmetic was initially implemented, the full throughput of FP16 operations was only realized in later generations of microarchitectures.

bfloat16

The Brain Floating Point format (abbreviated to bfloat16) is identical in size to the FP16 type. However, it retains the full exponent width of the FP32 type and instead further reduces the width of the significand to fit in a 16-bit size. Conceived for deep learning, bfloat16 bears two advantages over FP16 (Kalamkar et al., 2019). Firstly, as the exponent size is identical to FP32, it also shares the same range of values. This helps save on the tuning of an additional hyperparameter for loss scaling, a step required to achieve convergence when performing FP16-based training (see Section 2.2). Secondly, as the exponent is arranged before the significand in the bit representation, conversion from FP32 to bfloat16 simply involves truncating the last 16 bits (corresponding to a round-toward-zero) and is fast compared to FP32 to FP16 conversion. bfloat16 is supported by NVIDIA's CUDA and Tensor Cores from the Ampere microarchitecture onward (NVIDIA Corporation, 2020), as well as Google's second and third-generation Tensor Processing Units (TPUs) (Wang & Kanwar, 2019).

TensorFloat-32

TensorFloat-32 (TF32) is a format recently introduced by NVIDIA alongside the Ampere microarchitecture. TF32 uses the same 10-bit significand as FP16 and, like bfloat16, keeps the full 8-bit exponent width of the FP32 type. TF32 thus reaps the aforementioned advantages bfloat16 has over FP16. However, TF32 operations are designed to operate directly on FP32 inputs and produce results in FP32 (Kharya, 2020) - this means that there are no gains in terms of storage size and data transfer speeds over FP32. The benefit of using TF32 over FP32 comes from faster arithmetic on the reduced precision data.

Others

Apart from the formats supported by hardware, some other types have been considered in NN research. An unofficial 8-bit "FP8" type explored by Wang et al. (2018) shares the same exponent width as FP16 but cuts the significand down to 2 bits. Aside from removing bits from the exponent and significand, other techniques to reduce representation size include dynamically setting exponent width (Dettmers, 2016), or representing data as a collection of significands with a shared exponent (block floating-point) (Das et al., 2018) (Fowers et al., 2018) (Köster et al., 2017).

2.2 Mixed-Precision Training

Existing studies have shown that NNs can be trained using only 16-bit (Kalamkar et al., 2019) (Micikevicius et al., 2017) (Zhao, Vogel, & Ahmed, 2019), and even 8-bit (Dettmers, 2016) (Wang, Choi, Brand, Chen, & Gopalakrishnan, 2018) wide floating-point representations, with little or no loss in accuracy. When going from FP32 to a format with a smaller exponent width, the loss in dynamic range presents issues. Micikevicius et al. (2017) explore this problem with FP16 and FP32 mixed-precision training and introduces a couple of techniques to achieve state-of-the-art equivalent accuracy. We highlight some of the techniques and ideas in brief.

Loss Scaling

During backpropagation, Micikevicius et al. (2017) find that FP16's narrow dynamic range is often unable to sufficiently represent error gradients, which are typically dominated by small magnitudes. Gradient values thus end up below the minimum representable range and become zeros, leaving much of the FP16 range unused. To counteract this, a technique known as loss scaling can be employed to shift the gradient values into the FP16 representable range. The scaling factor can be regarded as another model hyperparameter and can be derived via a host of methods.

High Precision Master Weights

Micikevicius et al. (2017) apply reduced precision to NNs by storing activations and gradients in FP16. They propose keeping an FP32 master copy of weights that is maintained with gradient updates. In each iteration of the forward and backward pass, an FP16 copy of the master weights is then used to leverage reduced precision arithmetic. The FP32 master copy mitigates the following issues:

1. the weight update, which is the product of weight gradients and learning rate, can be too small for the FP16 dynamic range,

2. the weight update can be too small compared to the weight value. During addition, the alignment of the binary point between the two operands could cause the weight update to be zeroed out (this is known as swamping (Higham, 1993)).

By representing the weight updates and values in FP32, both the range-related problems are overcome, at the expense of increasing the memory consumption for weights over single-precision training, due to the additional copy. However, the use of FP16 in the forward pass and backpropagation significantly outweighs this, approximately halving the overall memory usage of NNs.

Accumulating to Higher Precision

During forward and backward passes, multiple vector dot products are performed. Just like in weight updates, floating-point additions in these dot products can suffer from swamping if one of the numbers is sufficiently smaller than the other (Wang et al., 2018). This can be addressed by accumulating the products to a higher precision like FP32. Both Micikevicius et al. (2017) and Wang et al. (2018) noted that this measure was critical to matching the accuracy of baseline single-precision models.

The importance of accumulating lower precision operands to a higher precision result has already made its way to commercial GPUs. Tensor Cores are capable of multiplying two FP16 matrices and adding another FP16 or FP32 matrix to give an FP32 result matrix. This operation is known as general matrix multiply, or GEMM (Appleyard & Yokim, 2017). Micikevicius et al. (2017) uses the aforementioned operation for accumulations in convolutions, fully connected layers, and matrix multiplies in recurrent layers.

2.3 Mixed-Precision Inference

Much of current research around mixed-precision inference techniques have been targeted at integer quantization (Lin et al., 2015) (Jacob et al., 2017) (Courbariaux et al., 2015). This is because quantization techniques are receptive to acceleration by high-throughput integer math pipelines, which perform faster than reduced precision floating-point (Jacob et al., 2017).

In the studies exploring mixed-precision with floating-point data types (Kalamkar et al., 2019) (Micikevicius et al., 2017) (Wang et al., 2018), inference is typically performed with the same setup as training. This means the uniform use of low precision types in the weights and activations for the forward pass. While the techniques of loss scaling and using high precision master weights are irrelevant once training is over, accumulating to high precision to avoid swamping is still important during inference.

Trans-Precision Inference

In addition to performing reduced precision training and inference in tandem, there has also been research into the low-precision deployment of NNs trained in regular precision. This is critical in use cases where cost and energy constraints can limit performance (Sun et al., 2019). This model of reducing precision only after training was coined Trans-Precision Inference in Sun et al. (2019) and has been experimented on with integer

(Krishnamoorthi, 2018) and floating-point formats (Sun et al., 2019).

Wu (2019) reports that training NNs in FP32 and performing inference in FP16 would lead to the same accuracy as FP32 most of the time, with a 1.5 to 4.59$\times$ inference speedup over FP32 observed on state-of-the-art models across various configurations. He also observes that batch normalization is helpful to mitigating overflow.

In Sun et al. (2019), the authors note the advantages of using floating-point formats over integer quantization in trans-precision inference, namely that the latter is ineffective for ultra-short bit-widths and when applied to compact models on large datasets. On the other hand, floating-point schemes have a wider dynamic range and do not need to find the right quantization range for each layer and channel, which serves trans-precision inference more naturally. Their study uses a variable 8-bit format (1-bit sign, 4-bit exponent and 3-bit significand for inference) and proposes some strategies, for example, keeping the first and last layer in high precision and retaining precision in convolutional layers with activations of smaller variances. They achieve trans-precision inference accuracies within $\sim$0.5% of the FP32 baseline on a host of state-of-the-art models.

Chapter 3

GPGPU-Sim

The central piece of software in this project is GPGPU-Sim (Khairy, Shen, Aamodt, & Rogers, 2020), an open-source, cycle-level GPU simulator capable of running workloads written in CUDA or OpenCL. The simulator supports two simulation modes: performance and functional. In performance simulation mode, the simulator collects performance statistics at the expense of simulation speed. In functional simulation mode, the statistics are not collected and the only output is that of the workload.

3.1 Toolchain

Modern versions of the simulator use the Parallel Thread eXecution (PTX) pseudo-assembly instruction set developed by NVIDIA. PTX is embedded in binary files compiled by NVIDIA's CUDA Compiler (`nvcc`).

When performing regular hardware execution, a just-in-time compiler included in the graphics driver converts the embedded PTX instructions into another instruction set native to the specific hardware generation. These instructions are then run on NVIDIA's GPU processors.

For the simulator to run the compiled CUDA binary, it first invokes a CUDA binary

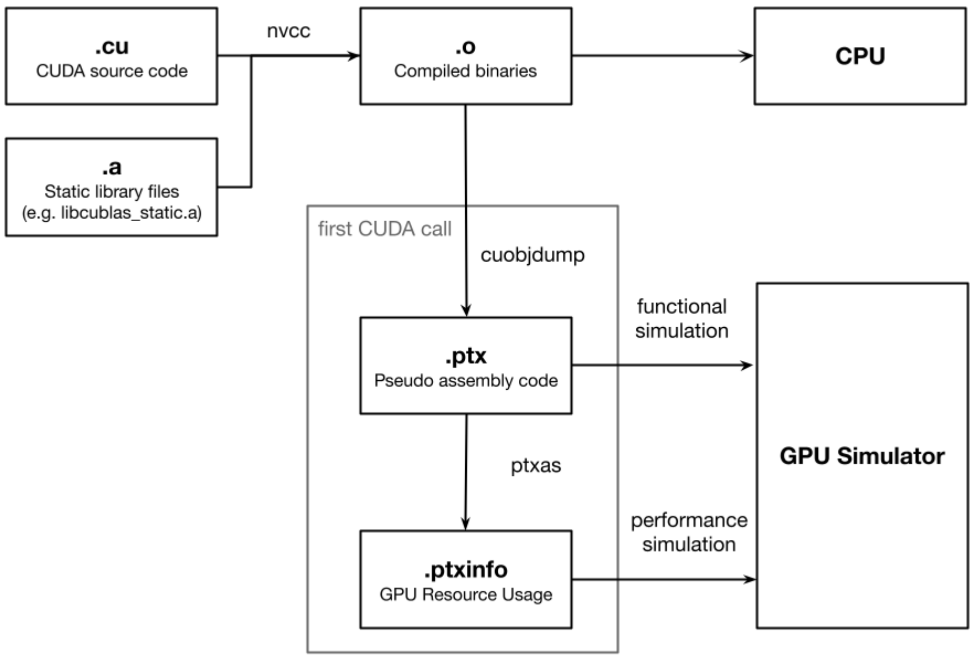


Figure 3.1: Simulator compilation toolchain.

utility, `cuobjdump` to disassemble the binary into PTX files. It does this on the program's first call to the CUDA API. The simulator then parses the PTX instructions to perform functional simulation. Additionally, the `ptxas` tool is used to generate resource usage information to be used in performance simulation. This entire process is illustrated in Figure 3.1.

Note that the simulator's use of PTX as input means that outside of source code manually compiled via `nvcc`, only libraries with embedded PTX can be run on the simulator. This poses restrictions on the CUDA programs that can be simulated. For example, the NVIDIA cuDNN deep learning library is used by many popular machine learning frameworks such as PyTorch and Tensorflow. However, NVIDIA has stopped embedding PTX in the cuDNN binaries from CUDA 9 onwards, rendering many of these deep learning frameworks incompatible.

3.2 PTX

Each kernel written in a CUDA program is translated into a corresponding kernel in PTX. The instruction set shares similar syntax with other assembly languages, defining statements using operation codes and operands. Each PTX statement is either a directive or an instruction.

```
.reg .f32 %f1;
```

Figure 3.2: Example of a PTX directive statement.

Directive statements begin with a dot and are used to declare constants, variables, or other details about the PTX file (such as the version of PTX used). In the example in Figure 3.2, a register, `f1`, is declared as FP32 type.

```
add.f32 %f3, %f1, %f2;
```

Figure 3.3: Example of a PTX instruction statement.

Instruction statements are used to specify operations. They are formed by an instruction opcode followed by a comma-separated list of zero or more operands. The destination operand is first in the list, followed by the source operands. The example in Figure 3.3 describes an addition operation on FP32 operands `f1` and `f2`, where the result is stored in `f3`.

More details about the syntax, primitives, and operations in PTX can be found in the official documentation (NVIDIA Corporation, 2021).

3.3 Architecture

GPGPU-Sim is a large program ($\sim$ 277,000 lines of code), comprising multiple components that provide a suite of simulation and visualization features. The full architectural details of GPGPU-Sim are documented in the simulator manual (Aamodt, Fung, & Hetherington, 2017). At a high level, there are three major modules:

- `cuda-sim`: Functional simulator
- `gpgpu-sim`: Performance simulator
- `intersim`: Interconnection network simulator

The majority of our modifications, described in Chapter 4, are functional, and as such revolve around the `cuda-sim` module. At a high level, the module is initialized on the simulated program's first CUDA call, reading from generated PTX files and saving all function and symbol table information. Any global or constant variables initialized on the device are also captured. When a kernel is launched by the host, GPGPU-Sim takes the kernel information (grid and block dimensions, parameters, etc.) and combines them with the function information (parsed instructions). Each thread in the kernel is then simulated one instruction at a time using the simulator's implementation of PTX operations.

Outside of these modules, there are also utility classes that orchestrate the core components. Of note, `abstract_hardware_model.cc` provides an interface between the functional and timing simulator, containing the abstractions modeling instructions, kernels, and memory.

Chapter 4

Simulator Enhancements

In this chapter, we describe the modifications that we perform on GPGPU-Sim to meet our objectives.

4.1 Low Precision Formats

The first and largest modification in the simulator is the addition of alternative, lower precision data types. We observed that to functionally simulate lower precision data in the simulator, we did not need to modify the representation of data within the simulator, which would require extensive architectural changes to the simulator (e.g. registers, pipelines, memory and bandwidth). We could rely on the same abstractions used to represent FP32 values and only override the implementation of individual PTX operations involving the simulated type. This is done by casting the operands to a low precision format, performing the operation in low precision, then saving the result back to the FP32 abstraction used by the simulator.

To accurately represent the operands and perform the operations in low precision, we utilized GNU MPFR 3.1.4 (Fousse, Hanrot, Lefèvre, Pélissier, & Zimmermann, 2007). MPFR is an open-source C library capable of representing floating-point numbers in arbitrary precision and performing arithmetic operations on them. In this section, we illustrate how we emulate a data type with arbitrary exponent and significand width with MPFR. To aid in visualization, we use the example of adding support for the PTX add operation for the bfloat16 type. The same steps can be extended to support other operations and data types.

4.1.1 FP32 Add

```
    // instructions.cc: add_impl
    case F32_TYPE:
      data.f32 = src1_data.f32 + src2_data.f32;
      break;

```

Figure 4.1: Single-precision add implementation.

Figure 4.1 showcases a snippet of the original implementation of the PTX `add` operation. The existing, single-precision add is performed by reading from the variables representing the operand registers (`src1_data` and `src2_data`). The result is then written to the variable representing the result register (`data`).

4.1.2 bfloat16 Add

To support bfloat16, we append a case block to be called when the parser reads that operands are of the `BF16_TYPE`, as depicted in Figure 4.2.

```
 1      // instructions.cc: add_impl
 2      case BF16_TYPE:
 3      {
 4          mpfr_t first, second;
 5          mpfr_set_emin(-132);
 6          mpfr_set_emax(128);
 7          mpfr_inits2(8, first, second, NULL);
 8          mpfr_set_flt(first, src1_data.f32, MPFR_RNDZ);
 9          mpfr_set_flt(second, src2_data.f32, MPFR_RNDZ);
10          int i = mpfr_add(first, first, second, mpfr_rounding_mode);
11          mpfr_subnormalize(first, i, mpfr_rounding_mode);
12          data.f32 = mpfr_get_flt(first, MPFR_RNDZ);
13          mpfr_clears(first, second, NULL);
14          break;
15      }
16
```

Figure 4.2: bfloat16 add implementation.

Significand

The significand width of the two source operands, declared in line 4, is specified as a parameter during initialization on line 7. While the IEEE 754 standard considers significands between 1 and 2 by placing the leading bit before the radix point (and making it implicit), MPFR considers significands between 1/2 and 1 by placing the leading bit right after the radix point. This means that bfloat16's 7 explicit bits of precision will require a precision of 8 bits in MPFR to be fully represented.

We then cast the FP32 operands into MPFR variables set to bfloat16 precision (lines 8 and 9), using the rounding mode round-toward-zero to perform truncation, which simulates the values being natively stored in bfloat16. We then perform the add operation on the MPFR variables with the help of a library function (line 10), using the prevailing rounding mode of the simulator (assigned to `mpfr_rounding_mode`). The result is then cast back to FP32 with round-toward-zero once again, and stored in the simulator's FP32 abstraction (line 12). Support for other bfloat16 PTX operations, such as subtract and multiply, is added similarly.

Exponent

The exponent range of MPFR variables is set pre-initialization via global flags. To emulate an 8-bit exponent width in MPFR, we first consider the representable range in IEEE 754 encoding, in exponents of 2. Taking into account the two special values (all 0s and all 1s), the range is -126 to 127. As highlighted in the previous section, MPFR considers significands in between 1/2 and 1 - as such, exponent values in MPFR are increased by 1 to compensate. This gives us a maximum exponent of 128, which we set in line 6.

To get the minimum exponent, we need to further consider the IEEE 754 subnormal range of the exponent. Subnormal numbers do not have the implicit leading bit, and as such, each bit in the significand grants us the ability to represent a value that is a factor of 2 below the smallest representable exponent. As bfloat16 has 7 bits of significand, this means that the minimum exponent is -126 - 7 = -133. Once again, accounting for the fact that exponent values in MPFR are larger by 1 compared to IEEE 754, we set a minimum exponent of -132 in line 5.

To fully emulate subnormal numbers, we also need to account for the shorter significand width of subnormal numbers. On line 54, prior to storing the output of the operation, we call on an MPFR function designed to round the number (if subnormal) to the appropriate subnormal precision.

4.2 PTX Instruction Overriding

With a methodology for performing PTX operations on custom data types in place, we then need a way for the CUDA application to use these custom data types.

Analyzing the GPGPU-Sim compilation toolchain (Figure 3.1), we note that both `nvcc` and the `cuobjdump` tool are closed-source. While open-source, third-party alternatives exist, the most feasible approach would be to avoid modifying any of these tools.

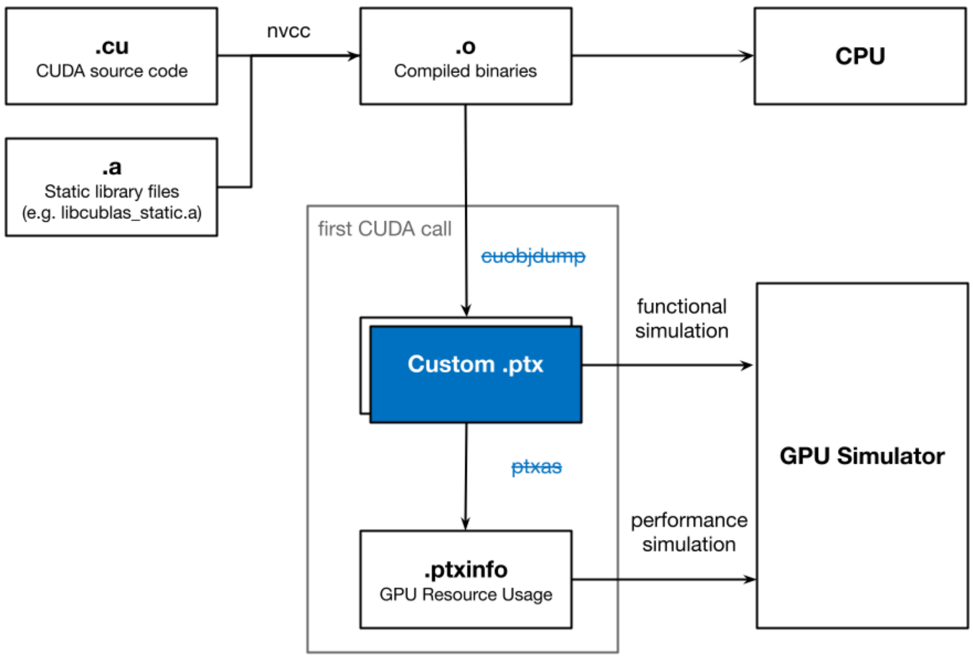


Figure 4.3: Modified compilation toolchain.

Instead, we introduce the custom data types at the PTX stage of the toolchain. First, the programmer obtains the original PTX and PTX information files by invoking the original program on the simulator. They will set a newly introduced configuration flag to block the `cuobjdump` and `ptxas` binaries from being invoked by the simulator. The programmer is then free to modify the previously generated PTX files and replace the data types of instructions with the custom data types introduced, which the PTX parser is modified to recognize. For example, to do an addition in bfloat16 precision, the replacement in Figure 4.4 would be performed.

`add.f32 %f3, %f1, %f2;` → `add.`**`bf16`**` %f3, %f1, %f2;`

Figure 4.4: Example of PTX replacement.

The modified toolchain is illustrated in Figure 4.3. When running functional simulation, we are now able to control, at the instruction level, which floating-point operations will be performed in low precision.

Note that a consequence of this approach is that the programmer needs to have a clear understanding of the purpose of each PTX kernel to perform meaningful overriding. We elaborate on this later in Section 5.1.

4.3 Varying Precision Dynamically

After initial exploration with the bfloat16 data type on a simple MLP model (Section 5.1), we realized that because PTX kernels were functions, overriding a kernel to use a specific precision would also determine the precision across all calls to the function throughout the lifetime of the program. This is extremely restrictive and would limit the techniques that the programmer could employ with the simulator.

One solution to address this issue was to "unroll" the kernel calls, making a separate PTX copy of the kernel for each call so as to use different data types in each of these copies. As we had opted not to modify the compiler or `cuobjdump`, this would require significant manual effort from the programmer to copy each function and call them separately. Another simpler approach was to modify the simulator to have a special data type — arbitrarily named `VF32` (variable float 32). Instead of using a statically defined precision, operations for this data type would set their precision based on kernel attributes, which are in turn set during kernel invocation by reading from environment variables. This gives us a way to synchronize the setting of precision with PTX kernels at runtime. We chose to adopt the latter approach and the abstract hardware model of the simulator was modified accordingly.

As each kernel is loaded in the simulator, three environment variables are read: `VF32_SIGNIFICAND`, `VF32_EXPONENT_MIN` and `VF32_EXPONENT_MAX`. These correspond to the significand precision, minimum exponent, and maximum exponent fields described in Section 4.1.2. To efficiently compute the appropriate parameters, we use the utility program in Figure 4.5. The program generalizes the steps in section 4.1.2 to work on any valid exponent and significand widths.

```python
def mpfr_exponent_range(exponent_width, significand_width):
    assert(exponent_width >= 1)
    assert(significand_width >= 1)

    # exponent range
    exponent_range = 2**exponent_width
    # account for IEEE special numbers
    exponent_range -= 2
    # max and min exponent with IEEE style offset
    maximum = exponent_range / 2
    minimum = - exponent_range / 2 + 1
    # account for MPFR significand representation
    mpfr_offset = 1
    maximum += mpfr_offset
    minimum += mpfr_offset
    # account for subnormal exponent range
    minimum -= significand_width

    print("VF_SIGNIFICAND  = {}".format(significand_width + mpfr_offset
    ))
    print("VF_EXPONENT_MIN = {}".format(int(minimum)))
    print("VF_EXPONENT_MAX = {}".format(int(maximum)))
    return minimum, maximum

```

Figure 4.5: Utility function to compute MPFR exponent range (Python 3).

4.4 New PTX Operations

In mixed-precision deep learning, not all operations are performed at one precision. In particular, accumulation and reduction require one operand (partial sum or product) to be kept in higher precision, as described in Section 2.2. Thus, mixed-precision versions of the PTX fused-multiply-add (`fma`) and addition operations (`add`) were introduced.

fmam

The `fmam` (fused-multiply-add mixed-precision) operation serves to perform accumulation to higher precision. It is modeled after the mixed-precision operation in Tensor Cores (Figure 4.6), where two operands are first multiplied in lower precision and added with an FP32 accumulator. The result is then stored as a FP32 value for further accumulation. Unlike the NVIDIA Tensor Core which only operates on FP16, our `fmam` instruction operates on formats of arbitrary precision thanks to the `VF32` type from the previous section.

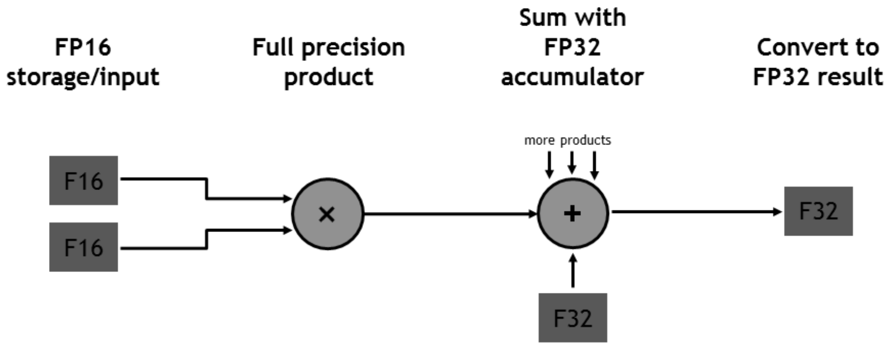


Figure 4.6: NVIDIA Tensor Core accumulation (Appleyard & Yokim, 2017).

addm

As some accumulation operations do not take the form of a fused-multiply-add operation, we add a variant of the PTX add operation to perform simple reduction to higher precision. Similar to `fmam`, `addm` takes an input operand of lower precision and sums it with an FP32 accumulator.

4.5 Variable Precision Performance

To measure the theoretical performance gained from running the simulator in low precision, we make a minor modification to the performance simulator. While the simulator produces a host of metrics, we target the total simulation time in cycles as a benchmark for execution time.

The simulator enables the configuration of latency and throughput (i.e. initiation interval) values for the integer, single-precision, and double-precision functional units present in the GPU's cores. There are many different ways to architecture a processing unit capable of low precision operations, however, we opt for the simplest approach that is apparent to us. We model a hypothetical processing unit where the single-precision floating-point pipeline has access to a separate set of low/variable precision functional units in addition to single-precision functional units. This means that we only have to read a separate set of latency and throughput values that the simulator would apply to `VF32` operations, and avoid complex changes to the performance simulator.

Chapter 5

Methodology

5.1 Deep Learning Models

Our search for appropriate deep learning models to run on the simulator was plagued with two challenges.

First, GPGPU-Sim had limited compatibility with libraries, in turn limiting compatibility with frameworks. Both high-level frameworks written in Python (e.g. TensorFlow) and lower-level alternatives (e.g. flashlight) typically required versions of CUDA/cuDNN that the simulator could not run. As explained in Section 3.1, this primarily stemmed from the fact that libraries could only be used on the simulator when PTX was embedded.

In addition, as the use of low precision data types takes place at the PTX level, the mapping from CUDA code to PTX kernels needs to be clear, so that the programmer is aware of what kernels to modify. For many libraries, each API call does not map to a single kernel, instead calling a series of kernels. In cases where the CUDA source code is closed-source, this mapping between library function and PTX kernel is entirely opaque.

As a result, we determine that in order to conduct meaningful experimentation, the deep learning programs that we choose should have minimal layers of abstraction in their implementation (i.e. be written in plain CUDA, as far as possible). This meant scrapping the first two deep learning projects we found and modified to be compatible with the simulator, as they both utilized the cuBLAS library (which provides an implementation of linear algebra subroutines). We instead found two models that had no dependencies on external libraries — MLP2 and CNN2, which we detail in this section. Both models take different approaches to performing identification on the MNIST handwritten digits dataset (LeCun, Cortes, & Burges, 2010) comprising 60,000 training examples and 10,000 test examples.

5.1.1 Multilayer Perceptron (MLP2)

The MLP2 project consists of a Multilayer Perceptron and is adapted from an open-source repository on GitHub (Powierza, 2018). The code is modified to fix bugs in data communication, loss calculation, and backpropagation, and subsequently made compatible with a simulator runner script. This includes splitting training and inference into independent stages by writing trained weights into an intermediate file.

Layers	Fully Connected Layer (128 nodes) ReLU Activation
	Fully Connected Layer (64 nodes) ReLU Activation
	Fully Connected Layer (10 nodes) Softmax Activation
Learning Rate	10^{-6}
Batch Size	500
Epochs	10000

Figure 5.1: MLP2 model configuration.

A few different network configurations were assessed to find an optimal setup for experimentation. The final model configuration is shown in Figure 5.1, yielding a baseline inference accuracy of 92.10% when run on hardware.

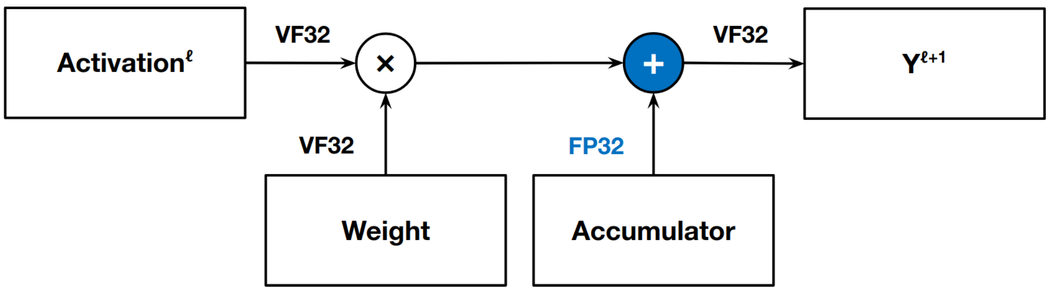


Figure 5.2: Accumulation to higher precision in activation step.

To set up the model for mixed-precision execution, the PTX generated (∼2000 lines) was studied and the relevant floating-point operations were overridden with the `VF32` format. This includes arithmetic operations, but not data-moving operations. For the experiments performing inference with accumulation to higher precision, we identified the accumulation operation in the activation step of the fully connected layer and applied the `fmam` operation. Depicted in Figure 5.2, this entails multiplying the previous layer's activation and the weight in `VF32` precision, then summing with an FP32 accumulator to give the result of the linear equation (which is fed into an activation function and becomes the activation value for the layer).

5.1.2 Convolutional Neural Network (CNN2)

For our second deep learning model, we sought to find a model more complex than a simple MLP. The CNN2 project, also sourced from an open-source GitHub repository (dwha, 2020), provides a generalized framework for writing convolutional neural networks as well as pre-configured models for the MNIST and CIFAR-10 dataset. We stick to using the MNIST model due to execution time concerns, as the CIFAR-10 dataset and model are orders of magnitude larger.

Layers	Convolutional Layer (3x3 kernel, 4 channels, stride 1, padding 1) ReLU Activation Max Pool
	Convolutional Layer (3x3 kernel, 8 channels, stride 1, padding 1) ReLU Activation Max Pool
	Convolutional Layer (3x3 kernel, 16 channels, stride 1, padding 1) ReLU Activation
	Fully Connected Layer (1000 nodes) ReLU Activation
	Fully Connected Layer (10 nodes) Softmax Activation
Learning Rate	10^{-3} (with decay)
Batch Size	50
Epochs	100

Figure 5.3: CNN2 model configuration.

From the initial codebase, we also modify the project to be compatible with our execution script, separating training and inference. The network configuration as provided by the author, shown in Figure 5.3, was suitable and used directly. In this configuration, the model produces an inference accuracy of 99.15%, significantly higher than MLP2. This is to be expected, given that a CNN has added complexity designed to handle image recognition tasks.

We generate and replace PTX in the same manner as MLP2. To perform inference with accumulation to higher precision, the modification as shown in Figure 5.2 is applied to the fully connected layers. However, accumulation to higher precision should also take place in the dot product between the kernel weights and receptive field in convolutional layers. The way `fmam` is utilized to achieve this is identical to the activation step in fully connected layers.

5.2 Operating Environment

In this section, we describe the operating environment in which we perform our experiments, as well as an auxiliary repository that we create to assist in running deep learning applications on the simulator.

5.2.1 Simulator Configuration

GPGPU-Sim provides different configurations modeling different NVIDIA GPUs. The latest CUDA microarchitecture modeled into the simulator as of writing is the Volta architecture (compute capability 7.0), in the form of the TITAN V GPU. The TITAN V configuration was rigorously tested alongside updates to the simulator's memory model (Khairy, Jain, Aamodt, & Rogers, 2018) and found to model performance much more precisely than previous configurations. We thus utilize this configuration in our experimentation.

5.2.2 Simulator Runner

In each run of the simulator, the latest application binary needs to be compiled. In addition, the simulator requires a setup script to be sourced and a configuration file to be provided. During execution, the simulator generates a series of temporary files that can be cleaned up after completion.

```
sim-dl-runner
├─ config/
│  ├─ config_volta_islip.icnt
│  └─ gpgpusim.config
├─ programs/
│  ├─ MLP2/
│  └─ CNN2/
├─ README.md
├─ runner.py
└─ utils.py
```

Figure 5.4: `sim-dl-runner` directory structure.

We set up a project `sim-dl-runner` (GPGPU-Sim Deep Learning Runner) to automate the above steps with a script and to manage the different application files and execution parameters on the simulator. The project's directory structure is shown in Figure 5.4. A summary is as follows:

- `config/`: Central location containing the simulator configuration files to be applied to all programs. We deliberately avoid keeping separate configuration files for different programs for greater ease. For our experiments, this directory contains the configuration files for the TITAN V card. Note that GPGPU-Sim expects that configuration files are located in the application's working directory, and so we make a simple modification to the simulator to check an environment variable `SIM_CONFIG_PATH` to determine if the files are located in a different directory.

- `programs/`: Contains the deep learning projects as git submodules. Making the projects submodules allows the repositories to be updated independently, an important feature if the project is still undergoing development.

- `README.md`: Simple document with instructions on how to use the runner script, as well as the make commands and execution parameters that the deep learning programs need to implement to be compatible with the runner script. This serves as a standard framework for adding any deep learning program to the `programs/` directory.

- `runner.py`: Python 3 script automating the simulator set up and clean up. In addition, the script sets the environment variables for `VF32` exponent and significand precision. Note that this refers to the initial precision — if a model intends to vary precision midway during execution, the environment variables need to be set dynamically in the application code. Users can also set the target deep learning application, toggle between executing on hardware and the simulator, as well as indicate whether training and/or inference should be executed. Finally, we can also specify the epochs to train for, as well as the file to save trained weights to. These features serve to improve the ease of running deep learning experiments on the simulator.

- `utils.py`: Python 3 utility functions to assist with IEEE 754 format translation and analysis. This includes the function in Figure 4.5.

5.2.3 Hardware Resources

The experiments are performed on a desktop computer with an AMD Ryzen 3700X 8-Core Processor and 64 GB of RAM, running Ubuntu 16.04. All of GPGPU-Sim's dependencies are also pre-installed alongside MPFR 3.1 for our variable precision enhancements and Python 3.7 for the runner script.

Chapter 6

Results

As we found that training had a prohibitively long run time on the simulator, we scope our experiments to trans-precision inference. Top-1 inference accuracy serves as the primary dependent variable, as it is a generalizable metric that succinctly captures the performance of a NN.

6.1 Trans-Precision Inference

6.1.1 Standard Types

Format	Accuracy	
	MLP2	CNN2
FP32	92.09%	99.15%
FP16	55.56%	99.15%
bfloat16	92.07%	99.14%
TensorFloat-32	92.07%	99.15%
FP8	69.46%	97.71%

Table 6.1: Accuracy across standard formats.

The initial experiment compares the common data types used in the industry. FP32 serves as the baseline and corresponds to the result produced when running the simulator on physical hardware. From Table 6.1, it can be seen that between MLP2 and CNN2, MLP2 is more sensitive to lower precision than CNN2.

Expectedly, running the models in baseline FP32 yields the highest Top-1 accuracy. The next best performing data type is TensorFloat-32, which has the same exponent as FP32 but a narrower significand. Further truncation of the significand to bfloat16 brings down accuracy marginally. However, when the exponent width is reduced, MLP2 suffers greatly, with accuracy falling to slightly above 50%. This is not observed with CNN2 — the reduction in accuracy between FP32, TensorFloat-32, bfloat16, and FP16 for CNN2 is so minor that the change can be reasonably attributed to noise. Finally, for FP8, which comprises both a narrow exponent and narrow significand, trans-precision inference on both models suffers greatly.

Except for FP8, these formats are all fairly wide at 16 bits and larger. We explore narrower formats in the following experiments.

6.1.2 Accumulation to Higher Precision

Model	Format	Accumlation To FP32	
		No	Yes
MLP2	FP8	69.46%	88.69%
CNN2	FP8	97.71%	98.23%

Table 6.2: Accumulating to higher precision.

Accumulating to higher precision is a technique commonly used to address swamping errors, as explained in Section 2.2. We experiment with accumulation to higher precision for each model using the modifications discussed in the methodology section.

We use the FP8 type for this experiment, since the accuracy of types with higher precision was already close to the original model accuracies. The benefit of accumulating to FP32 is observed with both models, suggesting that swamping was present and being mitigated in the activation step of the model. Our results agree with the observations made in past work, and thus we employ accumulation to higher precision for all further experiments.

6.1.3 Reducing Exponent

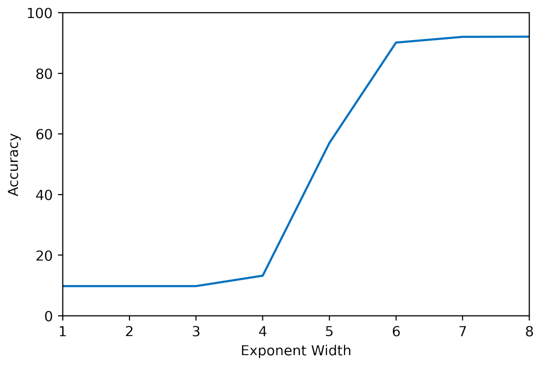
The next experiment reduces the width of the exponent while holding the significand constant. This translates to reducing the dynamic range of the floating-point values in the model. A constant significand width of 7 was chosen, along with a base exponent width of 8 (bfloat16). We chose to start from a narrow format like bfloat16 as we expect that reducing the range in shorter formats would produce a more discernible trend.

From Figure 6.1, we observe that both models degrade to an accuracy of approximately 10% as exponent width is reduced to 1. As the MNIST dataset has 10 category labels, this observation is congruent with the expected value of randomly assigning labels.

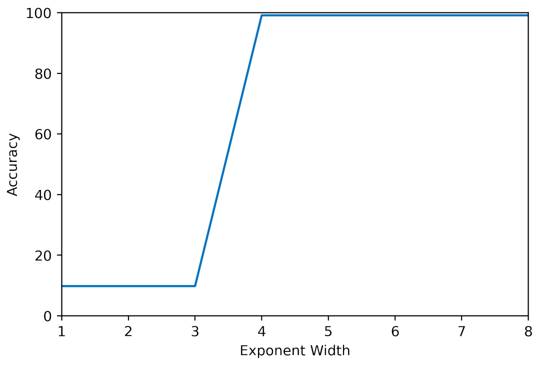
In line with our earlier experiment in Section 6.1.1, the CNN2 model is more resistant to reductions in dynamic range (Figure 6.1b). It retains its accuracy from 8 to 4 bits but sharply falls to the 10% expected value past 4 bits. We believe this indicates that the intermediate floating-point values in the model fall in the range provided by 4 bits.

On the other hand, for the MLP2 model, reductions in dynamic range instantly impact accuracy, although the effect is minor down to 6 bits (Figure 6.1a). From 6 to 4 bits, precision falls rapidly, plateauing after at 10%. Unlike CNN2, it appears that the range of intermediate floating-point values is an exponent bit wider, with the largest percentage falling inside the 4 and 6-bit range.

To gain further insight into the dynamic range in the model, we plotted the distribu-

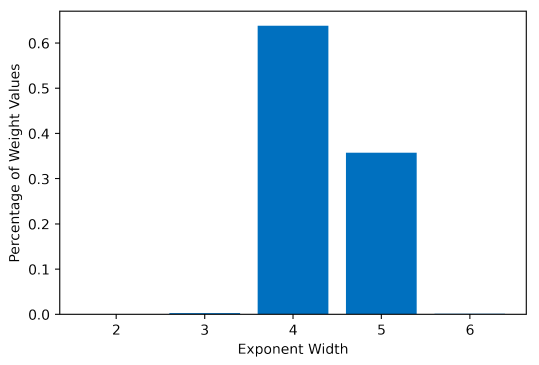


(a) MLP2.

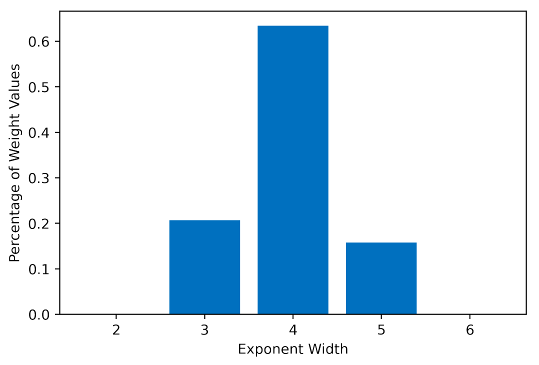


(b) CNN2.

Figure 6.1: Reducing exponent width (7-bit significand).



(a) MLP2.



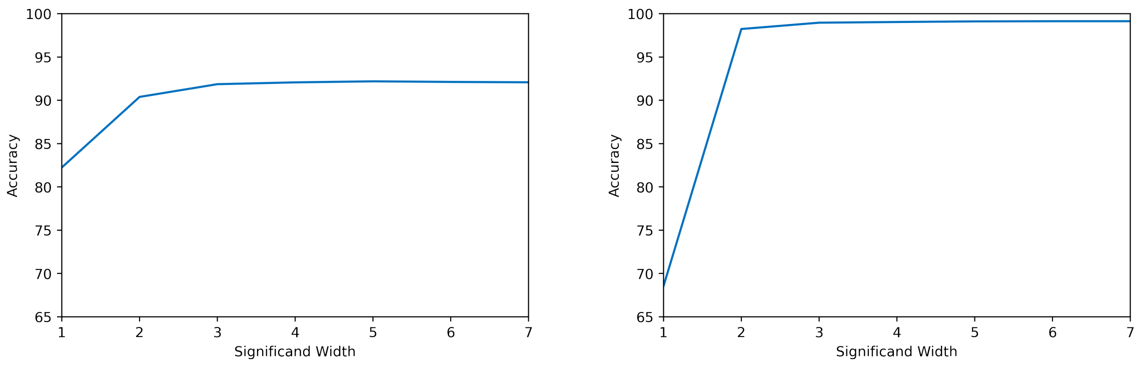
(b) CNN2.

Figure 6.2: Exponent width distribution of trained weights.

tion of the weights of both models (Figure 6.2) against their dynamic range (in exponent bits). Note that the initial weight values do not directly correlate with the intermediate values, but gives an indication of the magnitude of the operations in the NN. While both models have approximately 65% of weights in the range of a 6-bit exponent, it appears that CNN2 has approximately 0.2% of weights smaller than the 4-bit range while MLP2 does not. This could be a factor resulting in MLP2 working with a larger range of intermediate weights compared to CNN2, thus being more sensitive to the reduction in exponent width.

6.1.4 Reducing Significand

The experiment performed in the previous section was mirrored for the significand width. This is analogous to reducing the precision of the floating-point values while preserving dynamic range.

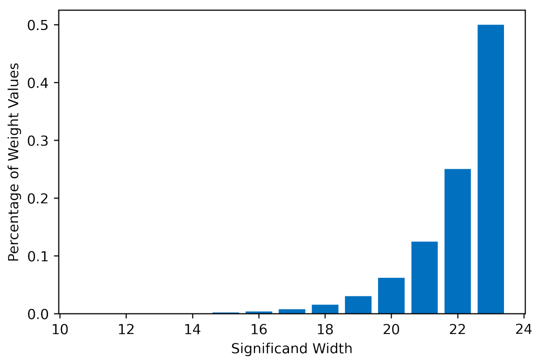


(a) MLP2.

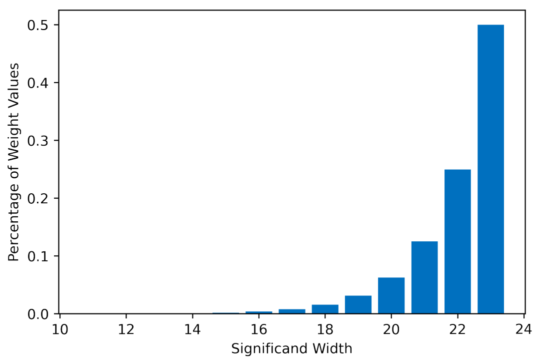
(b) CNN2.

Figure 6.3: Reducing significand width (8-bit exponent).

As in the previous experiment, both models exhibit gradual degradation as significand width is decreased (Figure 6.3). In this case, the accuracy never falls to the expected value of random guesses, with both models maintaining an accuracy above 65% even with a 1-bit significand. Furthermore, accuracy for both models only experience a noticeable



(a) MLP2.



(b) CNN2.

Figure 6.4: Significand width distribution of trained weights.

breakdown at 2 bits, and more drastically with a 1-bit significand. We believe that intermediate values rely more heavily on dynamic range to be expressive, and as such even with just a few bits of precision the models are capable of reasonable performance.

It also appears that the MLP2 model, despite starting from a lower baseline accuracy, experiences less of a decrease from 2 to 1 bits compared to CNN2. Taking into account the results from the previous section, it is likely that the MLP2 model has higher variance in the order of magnitude of intermediate values, and as such is less responsive to reductions in significand width as opposed to CNN2.

We do not expect the distribution of significand widths in the weights file to have any correlation with the results in Figure 6.3, as the precision of floating-point operands does not have any bearing on the precision of operation results. Analyzing the weights file proved this hypothesis (Figure 6.4). However, it is interesting to note that the significand width of the weights of both models followed an exponential trend.

6.1.5 Exponent-Significand Tradeoff

We then explore the trade-off between exponent and significand width, given a fixed bit-width. A "budget" of 12 bits, including the 1-bit sign, was chosen as we felt it offered a good balance between having a short width to observe the effects of limited precision, and having enough bits to vary. Furthermore, at representations greater than 12 bits, our experiments in Sections 6.1.3 and 6.1.4 suggest that it would be reasonable to adopt the naive strategy of truncating the significand.

Exponent	Significand	Accuracy
8	3	91.86%
7	4	**92.07%**
6	5	91.90%
5	6	63.18%
4	7	13.24%

(a) MLP2.

Exponent	Significand	Accuracy
8	3	98.96 %
7	4	99.04 %
6	5	99.11 %
5	6	**99.13%**
4	7	**99.13%**

(b) CNN2.

Figure 6.5: Exponent-significand tradeoff (12-bit representation).

From Figure 6.5, we can see that the highest accuracies of each model (in bold), coincide with our findings from the previous experiments. For CNN2, the ideal configuration would be trading off the exponent and preserving the significand. Even for MLP2, which is sensitive to reductions in exponent width, trading off 1 bit is beneficial under the 12-bit budget. A key insight that we can deduce is that given a fixed width, different models have different optimal allocations to exponent and signficand for trans-precision inference, or inference in general.

6.1.6 Varying Precision across Layers

Thus far, all experiments performed take a naive approach towards running trans-precision inference — a single low precision format is chosen and used across the entire execution. In this section, we attempt to vary the precision across the different layers of a model. We focus on strategies that only vary significand width, as we believe that varying exponent width within a model might realistically incur costly conversion between types.

MLP2

In the configuration we have chosen for MLP2, there exist 3 dense layers, inclusive of the output layer.

Significand Width			Accuracy
Fully Connected Layers			
7	3	1	90.77%
1	3	7	88.36%
3	1	1	89.57%
1	1	3	82.22%

Table 6.3: Decreasing and increasing precision across layers, MLP2 (8-bit exponent).

We first attempt to identify if decreasing or increasing precision across the layers produces better outcomes (Table 6.3). The first row represents an execution run where we decrease precision, starting with a 7-bit significand in the first layer, then using a 3-bit significand in the second and a 1-bit significand in the last. Comparing the first and second rows, it appears that keeping the earlier layers in higher precision serves this particular configuration well. We hypothesize that precision is more crucial at the beginning when values are closer in magnitude. Any errors lost to precision are also carried forward and amplified in later operations, which is mitigated with a higher precision at the start. In the later layers, the differences in magnitude between the values are likely large enough such that we can forego the significand precision.

The third and fourth rows validate the observations using smaller significand widths to amplify the trend.

Another strategy utilized in previous work is to keep the first and last precision in

Significand Width			Accuracy
Fully Connected Layers			
1	1	1	82.24%
3	1	3	90.50%
3	3	1	90.62%
1	3	3	88.10%

Table 6.4: High precision first and last layer, MLP2 (8-bit exponent).

high precision (Sun et al., 2019). We seek to test this hypothesis in Table 6.4. However, our results indicate that while a high precision first and last layer improves accuracy over low precision (first and second rows), it is beneficial to simply retain the precision at the earlier stage of the model (third row).

CNN2

The CNN2 model comprises 3 convolutional layers and 2 fully connected layers.

Significand Width			Accuracy
Convolutional Layers			
7	7	7	99.13%
3	3	3	98.96%
1	1	1	80.33%
7	3	1	98.24%
1	3	7	98.11%

Table 6.5: Varying precision across convolutional layers, CNN2 (7-bit significand in fully connected layers).

We run an experiment to investigate the effects of precision in the convolutional layer

Significand Width		Accuracy
Fully Connected Layers		
7	7	99.13%
3	3	99.04%
1	1	98.59%

Table 6.6: Varying precision across fully connected layers, CNN2 (7-bit significand in convolutional layers).

on a model's inference accuracy. In the first three rows of Table 6.5, we show that accuracy declines as the significand is reduced from 7 bits to 1. In particular, using a 1-bit significand gives close to a 20% reduction in accuracy. In the last two rows, we then attempt to determine if an increasing or decreasing pattern would be more optimal. Decreasing precision seems to perform marginally better, although the difference appears to be marginal and can be chalked up to random error.

Holding the precision in the convolutional layers constant, we vary the precision of the fully connected layers in Table 6.6. Comparing the results with Table 6.5, we observe that reducing precision in the fully connected layers has less impact on accuracy. This difference is especially apparent with a 1-bit significand, which loses less than 1%. As such, reducing precision in the fully connected layers seems like a viable strategy for this model, whilst keeping precision in convolutional layers as high as possible.

6.1.7 Optimal Precision

From our experiment findings, we aim to identify a trans-precision strategy that will result in minimal loss in accuracy compared to the baseline ($\sim$1%), while exploiting the narrowest floating-point formats to fully realize resource savings. As shown in our experiments thus far, different models have different characteristics. As a result, optimal strategies are specific to each model. In the following subsections, we informally demonstrate how one can run experiments similar to the ones we have conducted to discover traits about a given model, in order to develop and validate suitable mixed-precision strategies for it.

MLP2

Strategy	Fully Connected Layers						Accuracy
	Layer 1		Layer 2		Layer 3		
	E	S	E	S	E	S	
Baseline	8	23	8	23	8	23	92.10%
Uniform	6	3	6	3	6	3	**91.54%**
Uniform	6	2	6	2	6	2	90.91%
Decreasing	6	3	6	2	6	1	90.48%
Decreasing	6	3	6	3	6	2	**91.33%**
Decreasing	6	4	6	3	6	2	**91.55%**

E – exponent width, S – significand width.

Table 6.7: Search for optimal configuration, MLP2.

For the MLP2 model, inference exhibits significantly better results when accumulating to FP32, and as such we apply this strategy. In the experiments investigating exponent and significand width, it was found that the model is very sensitive to reductions in exponent range, hinting at the presence of a greater range of intermediate values. We hence seek to preserve as much of the exponent width as possible, trading off a maximum of 2 bits. Accordingly, the wider range of values also means that the model is resistant to reductions in significand precision, and we believe we can comfortably reduce the significand down to 3 bits or less. While decreasing precision across layers would not change the widest floating-point format a model has to work with, it may be effective in speeding up computational time by leveraging even lower precision functional units in later layers. We thus also attempt it.

We tabulate the results of our runs in Table 6.7. The first row is the baseline FP32 model accuracy, while the subsequent two rows denote a uniform low-precision strategy and the final three rows decreasing ones. It would appear that among the uniform strategy runs, reducing the significand to 2 bits drops the precision below our desirable threshold, and as such we should stick to a 3-bit significand. Starting with a 3-bit significand, decreasing precision across layers appears to degrade accuracy beyond our threshold. It is only when we start the first layer with 4 bits (last row) that we get a result comparable to uniformly setting a 3-bit significand. As a result, we can conclude that the narrowest format that yields a trans-precision inference accuracy within 1% of the baseline accuracy is a 10-bit floating-point representation with a 6-bit exponent and 3-bit significand. More generally, all configurations with accuraries in bold meet our threshold and are all viable mixed-precision strategies.

CNN2

As above, we first note that accumulation to higher precision has a desirable effect, albeit minor. In terms of exponent width, this model is resistant to reductions, only exhibiting a clear drop-off in performance at 3 bits and retaining high accuracy for all values above that — consequently, we set an exponent width of 4. A distinct drop-off is also present when experimenting with significand width at 1 bit, thus, 2 to 3 bits seem desirable. We also try reducing precision in fully connected layers, which our experiments earlier show might be a sound strategy.

The results of our hypotheses are shown in Table 6.8. It appears that a uniform strategy of using a 3-bit significand produces an outlier result, with accuracy significantly lower compared to using a 2-bit significand and our results in previous experiments. We perform a run with a 4-bit significand (second row) and verify that it is indeed an anomaly.

Strategy	Convolutional Layers		Fully Connected Layers		Accuracy
	All Layers		All Layers		
	E	S	E	S	
Baseline	8	23	8	23	99.15%
Uniform	4	4	4	4	**99.05%**
Uniform	4	3	4	3	95.62%
Uniform	4	2	4	2	**98.31%**
Decreasing	4	3	4	2	**98.89%**
Decreasing	4	3	4	1	**98.51%**
Decreasing	4	2	4	1	97.41%

E – exponent width, S – significand width.

Table 6.8: Search for optimal configuration, CNN2.

When using a 3-bit significand in convolutional layers and a 1 or 2-bit significand in fully connected layers, we get results that fall within our expectations. A decreasing strategy yields a result below our threshold only when starting with a 2-bit significand in the convolution layers. Thus, it appears that the narrowest format for this model that produces near baseline accuracy is a 7-bit representation with a 4-bit exponent and 2-bit significand. This finding is congruent with past work that have observed good inference accuracy with 8-bit formats. As with the previous model, all configurations with accuracies that meet the threshold are once again in bold.

6.1.8 Performance

In this section, we present some insights on run time and space benefits from running inference in reduced precision, utilizing a mix of empirical and theoretical methods.

Run Time

To measure improvements to execution time, we use the total simulation cycles metric produced from the simulator's performance mode, leveraging the changes introduced in Section 4.5. We conduct the experiments on the MLP2 model, which has a shorter execution time on the simulator compared to CNN2. Despite so, we find the run time of performance simulation prohibitively high, a limitation we discuss in Section 6.2.

Latency (Cycles)					Total Simulation Cycles
ADD	MAX	MUL	MAD	DIV	
4	13	4	5	39	54878026
3	10	3	4	30	54860737
2	7	2	3	20	54860545

Table 6.9: Total simulation cycles against latency values, MLP2.

The header of Table 6.9 lists the various instruction categories of `VF32` operations of which we can independently configure the latency and initiation intervals. Inference was executed in performance simulation with the respective latency values (in cycles) in each row.

The first row is set to the same latency values as FP32 operations and indicates the baseline total execution cycles when inference is performed in single-precision. As we were not able to find reference values for real-world, existing/proposed functional units, we reduce the latencies arbitrarily to observe the extent to which performance can be improved. However, it appears that the gains are minimal – the last row, which is extremely optimistic in functional unit latency reduction, only offers approximately 0.03% improvement over the baseline total simulation cycles. This is likely a limitation of how we model low precision data types in the simulator. By leveraging the existing FP32 abstractions to store the low precision formats, we are unable to capture any performance benefits from the reduction of bandwidth bottlenecks and better usage of the cache.

Storage

When running inference, the weights file is "state" that needs to be carried over from training. If a narrow floating-point format is used in inference, the weights can be stored directly in the same narrow format, producing a more compact model that can be loaded onto low-powered or embedded devices. When calculating the storage benefits of saving weights in low precision, it is important to consider that data needs to be kept in word-aligned chunks. This is a factor behind the common data types being of width 16 and 8, both of which can be stored compactly into 32 and 64 bit words. Naturally, we expect that a model using a 16-bit format would consume only 50% of the space of the original FP32 weights file, and an 8-bit format 25%.

In the previous section, we deduce that a 10-bit format for MLP2 and a 7-bit format for CNN2 is ideal. Theoretically, this would mean that the size of the weights file for the models can be reduced by 2/3 and 3/4 respectively, which are non-trivial reductions. As we do not model the reduced memory footprint of low precision types in the simulator, we are unfortunately unable to draw further conclusions about memory and bandwidth usage at this time.

6.2 Limitations

Over the course of the project, we discovered some limitations that challenge the feasibility of using GPGPU-Sim, and more broadly a GPU simulator to study mixed-precision in deep learning.

As discussed in Section 3.1 and Section 5, GPGPU-Sim has limited support for libraries due to its reliance on embedded PTX to perform simulation. This poses an inherent disjoint between programs that can run on the simulator and programs that are relevant in the industry. Researchers will have to work in the intersection of both spaces, which may be restrictive. Moreover, the simulation time on GPGPU-Sim can become infeasible for deep learning programs, which tend to be compute-intensive. For example, inference with the CNN2 model with an NVIDIA RTX 2060S card completes in 0.56 seconds but runs upwards of 10 hours on the simulator in our experiments. Given that the CNN2 model is extremely simple, running state-of-the-art NNs might be computationally infeasible. We also observe that GPGPU-Sim requires further development to handle big applications, as memory leaks were observed with the Valgrind debugging tool.

Our approach of overriding PTX to specify low precision operations also has some drawbacks. Firstly, the replacement process is highly manual, although this could potentially be automated to a degree. Next, for precise overriding, there needs to be a clear mapping between CUDA functions and PTX kernels. Library functions, where the CUDA source might not be available or one library call may invoke many kernels, obscure this mapping and make replacement challenging. When working with our initial projects that utilized the cuBLAS library, we found it impossible to identify the operation to override to perform accumulation to higher precision. This implicit requirement that the application should use minimal open-sourced libraries adds to the compatibility issues described earlier.

Lastly, varying precision dynamically requires calls to set the precision of the `VF32` type within the application code itself. Interleaving the code setting simulation parameters with the application program breaks abstraction, but is a necessary tradeoff for synchronizing precision changes with execution.

Chapter 7

Conclusion

7.1 Summary

In this project, we explore the use of a variable precision GPGPU simulator as a medium for empirical research into mixed-precision deep learning. To this end, we modify a popular open-source GPGPU simulator, GPGPU-Sim, to perform operations at low and mixed precisions with arbitrary IEEE 754-style floating-point types. Our modifications allow instruction-level control over which floating-point operations should be executed in low precision, with the ability to allow applications to dynamically set the specific precision during execution. Our modifications also take the first steps towards simulating the performance of a variable precision GPGPU, modelling functional unit latency and initiation intervals but short of memory and communication bandwidth. We also introduce an auxiliary project comprising a script and an accompanying workflow for running deep learning applications on the simulator.

Through our experiments, we demonstrate how the modified GPGPU-Sim can be used to examine and empirically derive desirable mixed-precision strategies for different deep learning models. Unlike previous work which has largely focused on enabling a single low precision format in a given model, our approach grants the ability to use multiple low precision types across different stages of execution. We also glean some insights about mixed-precision inference accuracy in a multilayer perceptron and a convolutional neural network. A deeper investigation is required to ascertain the generalizability of these findings; however, they can form the basis for further research.

7.2 Future Work

Despite the limitations of GPGPU simulation in exploring mixed-precision deep learning highlighted in Section 6.2, we believe that the approach is still viable and will be increasingly so given better computational resources and further developments to the simulator. In future studies, we propose the following directions for exploration:

- Modifying GPGPU-Sim to model low precision types in native abstractions, rather than utilize the existing FP32 data type. This would include adding a separate pipeline for low precision operations as well as modeling the reduced memory and bandwidth utilization. These enhancements would enable more accurate performance simulation of a hypothetical GPGPU capable of mixed-precision operations.
- More advanced mixed-precision strategies can be examined with the simulator. Instead of varying precision between layers, we encourage exploiting the ability to set precision at the PTX instruction level to experiment on more granular strategies.
- Should time and computational resources not be a constraint, mixed-precision training strategies can be studied with the simulator. This can be approached from two angles — improving the efficiency of training, and developing models specifically designed to thrive when deployed in lower precision.
- Lastly, it would also be meaningful to validate our approach and observations on more deep learning models of various types, including recurrent neural networks, and potentially generalize them.

References

Aamodt, T. M., Fung, W. W., & Hetherington, T. H. (2017). *GPGPU-Sim 3.1.1 manual*. GPGPU-Sim, 1.2 edition.

Appleyard, J., & Yokim, S. (2017). Programming tensor cores in cuda 9.

Cai, Z., He, X., Sun, J., & Vasconcelos, N. (2017). Deep learning with low precision by half-wave gaussian quantization. *CoRR*, *abs/1702.00953*, , 2017.

Courbariaux, M., Bengio, Y., & David, J. (2015). Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, *abs/1511.00363*, , 2015.

Das, D., Mellempudi, N., Mudigere, D., Kalamkar, D., Avancha, S., Banerjee, K., Sridharan, S., Vaidyanathan, K., Kaul, B., Georganas, E., Heinecke, A., Dubey, P., Corbal, J., Shustrov, N., Dubtsov, R., Fomenko, E., & Pirogov, V. (2018). Mixed precision training of convolutional neural networks using integer operations.

Dettmers, T. (2016). 8-bit approximations for parallelism in deep learning. *CoRR*, *abs/1511.04561*, , 2016.

dwha (2020). SimpleCudaNeuralnet. `https://github.com/dwha/SimpleCudaNeuralNet`.

Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., & Zimmermann, P. (2007). MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, *33*(2), , June, 2007, 13–es.

Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., Adams, L., Ghandi, M., Heil, S., Patel, P., Sapek, A., Weisz, G., Woods, L., Lanka, S., Reinhardt, S. K., Caulfield, A. M., Chung, E. S., & Burger, D. (2018). A configurable cloud-scale dnn processor for real-time ai. *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18 (p. 1–14), , 2018: IEEE Press.

Higham, N. (1993). The accuracy of floating point summation. *SIAM J. Sci. Comput.*, *14*, , 1993, 783–799.

Ho, N.-M., & Wong, W.-F. (2017). Exploiting half precision arithmetic in NVIDIA GPUs (pp. 1–7), , 09, 2017.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A. G., Adam, H., & Kalenichenko, D. (2017). Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, *abs/1712.05877*, , 2017.

Johnson, J. (2018). Rethinking floating point for deep learning.

Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., Yang, J., Park, J., Heinecke, A., Georganas, E., Srinivasan, S., Kundu, A., Smelyanskiy, M., Kaul, B., & Dubey, P. (2019). A study of bfloat16 for deep learning training.

Khairy, M., Jain, A., Aamodt, T. M., & Rogers, T. G. (2018). Exploring modern GPU memory system design challenges through accurate modeling. *CoRR*, *abs/1810.07269*, , 2018.

Khairy, M., Shen, Z., Aamodt, T. M., & Rogers, T. G. (2020). Accel-sim: An extensible simulation framework for validated gpu modeling. *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20 (p. 473–486), , 2020: IEEE Press.

Kharya, P. (2020). Tensorfloat-32 accelerates AI training HPC up to 20x.

Krishnamoorthi, R. (2018). Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, *abs/1806.08342*, , 2018.

Köster, U., Webb, T. J., Wang, X., Nassar, M., Bansal, A. K., Constable, W. H., Elibol, O. H., Gray, S., Hall, S., Hornof, L., Khosrowshahi, A., Kloss, C., Pai, R. J., & Rao, N. (2017). Flexpoint: An adaptive numerical format for efficient training of deep neural networks.

LeCun, Y., Cortes, C., & Burges, C. (2010). Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, *2*, , 2010.

Lin, D. D., Talathi, S. S., & Annapureddy, V. S. (2015). Fixed point quantization of deep convolutional networks. *CoRR*, *abs/1511.06393*, , 2015.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., & Wu, H. (2017). Mixed precision training.

NVIDIA Corporation (2020). *NVIDIA A100 tensor core GPU architecture*. NVIDIA, 1.0 edition.

NVIDIA Corporation (2021). *Parallel thread execution ISA version 7.2*. NVIDIA, 11.2.2 edition.

Powierza, J. (2018). CUDA-DNN-MNIST. `https://github.com/jpowie01/CUDA-DNN-MNIST`.

Sun, X., Choi, J., Chen, C.-Y., Wang, N., Venkataramani, S., Srinivasan, V., Cui, X., Zhang, W., & Gopalakrishnan, K. (2019). Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. *NeurIPS*, , 2019.

Wang, N., Choi, J., Brand, D., Chen, C.-Y., & Gopalakrishnan, K. (2018). Training deep neural networks with 8-bit floating point numbers.

Wang, S., & Kanwar, P. (2019). Bfloat16: The secret to high performance on cloud TPUs.

Wu, H. (2019). Low precision inference on GPU.

Wu, H., Judd, P., Zhang, X., Isaev, M., & Micikevicius, P. (2020). Integer quantization for deep learning inference: Principles and empirical evaluation.

Zhao, R., Vogel, B., & Ahmed, T. (2019). Adaptive loss scaling for mixed precision training.