On the Challenges in Programming Mixed-Precision Deep Neural Networks

Ruizhe Zhao
ruizhe.zhao15@imperial.ac.uk
Imperial College London
UK

Wayne Luk
w.luk@imperial.ac.uk
Imperial College London
UK

Chao Xiong
chao.xiong@corerain.com
Corerain Technologies
China

Xinyu Niu
xinyu.niu@corerain.com
Corerain Technologies
China

Kuen Hung Tsoi
kuenhung.tsoi@corerain.com
Corerain Technologies
China

Abstract

Deep Neural Networks (DNNs) are resilient to reduced data precision, which motivates exploiting low-precision data formats for more efficient computation, especially on custom hardware accelerators. Multiple low-precision types can be mixed to fit the dynamic range of different DNN layers. However, these formats are not often supported on popular microprocessors and Deep Learning (DL) frameworks, hence we have to manually implement and optimize such novel data types and integrate them with multiple DL framework components, which is tedious and error-prone.

This paper first reviews three major challenges in programming mixed-precision DNNs, including generating high-performance arithmetic and typecast functions, reducing the recompilation time and bloated binary size caused by excessive template specialization, and optimizing mixed-precision DNN computational graphs. We present our approach, Lowgen, a framework that addresses these challenges. For each challenge, we present our solution implemented and tested on our in-house, TensorFlow-like DL framework. Empirical evaluation shows that Lowgen can automatically generate efficient data type implementations that enable significant speed-up, which greatly lowers the development effort and enhances research productivity on mixed-precision DNN.

CCS Concepts: • **Software and its engineering** → **Development frameworks and environments**; **Source code generation**; **Just-in-time compilers**.

MAPL '20, June 15, 2020, London, UK
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7996-0/20/06.
https://doi.org/10.1145/3394450.3397468

ACM Reference Format:
Ruizhe Zhao, Wayne Luk, Chao Xiong, Xinyu Niu, and Kuen Hung Tsoi. 2020. On the Challenges in Programming Mixed-Precision Deep Neural Networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '20), June 15, 2020, London, UK*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3394450.3397468

1 Introduction

Using mixed-precision DNNs is beneficial. We can run some layers in a DNN with a much lower data precision than floating-point without losing much accuracy [7, 15], and this reduction in precision results in less computation budget. Using different precisions for layers in a mixed manner can further improve the trade-off between computation budget and model accuracy [16]. It is therefore plausible to try out and to mix new data types when building DNNs, especially for custom accelerators, e.g., GPU [22], TPU [14], FPGA [11], etc., on which low-precision data types are natively supported, and the benefits from mixing types are significant.

Integrating a new data type into an accelerator is costly both in time and money. Before we decide which data type should be placed on an accelerator, it is sensible to simulate its performance first on widely available platforms, e.g., x86 CPUs. To accomplish this simulation task, we should implement the target data type based on existing types, e.g., `float`, and examine it on real-world DNN models. Nevertheless, DL frameworks that are commonly used to experiment with DNN models poorly support such data type development scenarios. Specifically, utilities for designing and integrating new data types and optimization dedicated to mixed-precision graphs are not accessible in existing DL frameworks. For instance, to implement a new type, we should first handcraft its definition in the low-level runtime library, which requires much expertise in system architecture and much effort to tune performance; then, we need to update the communication protocols among modules to recognize this new type; and finally, it is necessary to develop specific optimization passes. As a result, researchers need to take much effort to experiment with any new data type, resulting in a high barrier to adoption of mixed-precision DNNs..

This paper presents Lowgen, a supplementary framework that makes programming mixed-precision DNN much simpler. Here, supplementary means Lowgen is not standalone, instead, it contains utilities that can be integrated into an existing DL framework. It provides a source-to-source transformer that generates high-performance data type definition from a short, un-optimized piece of code. It also embeds a JIT compilation mechanism to dynamically integrate the generated data definition into a compiled DL runtime. Finally, an optimization algorithm on mixed-precision DNN graphs is proposed in Lowgen, which reduces redundant typecast nodes and places them at the right place. We have implemented Lowgen upon our in-house, TensorFlow-like DL framework and empirically studied its effectiveness on `bfloat16` [15], a recently proposed low-precision type.

The major contributions of this paper:

- summarizing the challenges in implementing mixed-precision DNNs on state-of-the-art DL frameworks;
- proposing Lowgen, a supplementary framework that can address these challenges, which contains a high-performance code generator, a JIT compilation utility, and a mixed-precision graph optimizer;
- empirically evaluating Lowgen on a specific custom data type and various DNN models.

2 Background and Problem Setting

2.1 DL Framework

A DL framework, such as TensorFlow [1] and PyTorch [24], provides us with the essential toolkit to build a DNN with much less effort as doing so from scratch. A typical DL framework has three major components: a high-performance runtime library for DNN execution, a graph library that constructs and manipulates the computation graph of DNN, and a set of easy-to-use API exposed to end-users. They work collaboratively to enhance the DNN development experience. Without loss of generality, this paper mainly looks at TensorFlow [1, 2], one of the most popular DL frameworks. Figure 1 illustrates its major modules. We have the following additional remarks on them.

Runtime library. A runtime library mainly implements *basic linear algebra subprograms* (BLAS) on tensors that are fundamental to DNN operations. TensorFlow normally delegates this task to existing platform-specific BLAS libraries, e.g., Eigen [10], cuDNN [6], etc. These BLAS routines will be wrapped into DNN operators, which are further exposed to upper-level framework components. Typically, a DNN operator normally has a C++ template specification, which is parameterized by the data type and target platform, and for each valid pair of type and platform, there will be an explicit template specialization instance. Figure 2 shows the pattern of how DNN operator templates are implemented.

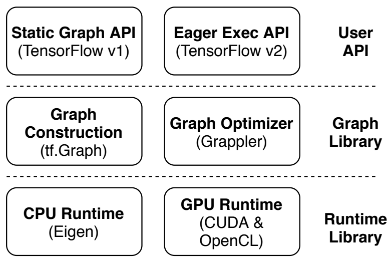


Figure 1. The general architecture of TensorFlow. Each rectangle indicates a specific module, in which module names are bolded and corresponding instances are wrapped in brackets. Dotted lines separate modules into three major components (names are labelled to the right).

```
template<typename T, typename Device>
class Conv2DOp: public OpBase<T, Device> {
  // class contents ...
};
template<> // specialization
class Conv2DOp<float, CPUDevice> {
  // specific implementation for floating-point on CPU.
};
// Use a macro to register the mapping between operator
// label and template specialization
REGISTER_OP("Conv2D", Conv2DOp<float, CPUDevice>)
```

Figure 2. Template-based operator definition for `Conv2D`. There are a template definition, a specialization, and a registration macro to bind the operator name to an instance.

Graph library. DL frameworks treat a DNN as a computation graph, in which each node represents a specific DNN operator, and each directed edge indicates the data produced by the edge source and consumed by the destination. The graph library is responsible for constructing, optimizing, and executing the computation graph. TensorFlow provides an internal Python API, `tf.Graph`, to construct and manipulate graphs. Grappler [31] is another internal library that helps to optimize DNN graphs, with techniques like constant folding and arithmetic simplification. When executing the graph, the graph library will look up the label attached to each node in the runtime library for the corresponding operator.

User API. The interface provided to end-users is mostly Python, which delegates the workload to the underlying, and rather fixed, graph and runtime libraries. This API should take care of DNN model construction, dataset IO, training schedule, etc. There are two styles of API in TensorFlow: static, which constructs the graph before execution; and dynamic, which can execute while building the graph.

```
Tnew operator+(const Tnew &a, const Tnew &b) {
  return static_cast<Tnew>(static_cast<Tsim>(a) +
                           static_cast<Tsim>(b)); }
```

Figure 3. A C++ analog of the simulated addition arithmetic. Since $\mathtt{T}_{\mathtt{new}}$ is practically stored as $\mathtt{T}_{\mathtt{sto}}$, builtin function `static_cast<Tsim>` acts as $\mathtt{f}_{\mathtt{sto}\to\mathtt{sim}}$, and its counterpart `static_cast<Tnew>` is $\mathtt{f}_{\mathtt{sim}\to\mathtt{sto}}$. Using $\mathtt{T}_{\mathtt{sto}}$ directly in C++ for typecasting may be ambiguous.

2.2 Implementing a New Data Type

Theoretically, inventing a novel data type ($\mathtt{T}_{\mathtt{new}}$) is about defining the mapping from a bit representation to a value in the real domain and specifying how its arithmetic works. When implementing these two steps in software, we need to find an existing storage type ($\mathtt{T}_{\mathtt{sto}}$), e.g., unsigned integer, to hold the bit representation; and a simulation type ($\mathtt{T}_{\mathtt{sim}}$), e.g., floating-point, to carry out the arithmetic operations of $\mathtt{T}_{\mathtt{new}}$, which are not available on existing processors. Besides, we should define the typecast functions that convert between $\mathtt{T}_{\mathtt{sim}}$ and $\mathtt{T}_{\mathtt{sto}}$, $\mathtt{f}_{\mathtt{sim}\to\mathtt{sto}}$ and $\mathtt{f}_{\mathtt{sto}\to\mathtt{sim}}$, such that we can bring the values stored as $\mathtt{T}_{\mathtt{sto}}$ to $\mathtt{T}_{\mathtt{sim}}$ to do simulated arithmetic, and cast the result back to $\mathtt{T}_{\mathtt{sto}}$. Figure 3 shows an example of how the simulated data type works.

So far, implementing a new data type requires defining $\mathtt{T}_{\mathtt{sim}}$ and $\mathtt{T}_{\mathtt{sto}}$ and the typecast functions. Arithmetic operations normally match the previously shown pattern (Figure 3) and can be intuitively implemented. However, the programming paradigm imposed by DL framework adds two more constraints that make the development less straightforward:

- The runtime library should be extended to support the new data type, and therefore, a data type should be implemented in C++. Specifically, we should instantiate its corresponding numeric traits, and realize specific interface functions required by upstream libraries. To achieve acceptable performance, we also need to take effort to build vectorized arithmetic functions to utilize parallel hardware.
- The graph library and user API should recognize this new type. The data type field of each node should accept this new type, and in case there is any pair of nodes with incompatible types, we should be able to insert explicit typecast nodes, which should include cases of converting between the new type and any other existing types. Meanwhile, since the data type changes the underlying runtime, we may need to recompile the affected libraries.

The flow chart in Figure 4 illustrates the steps we should take to build a new data type and integrate into a DL framework. The overall development cycle can become too long to support productive research. The next section will elaborate in details the specific challenges we need to deal with during the implementation.

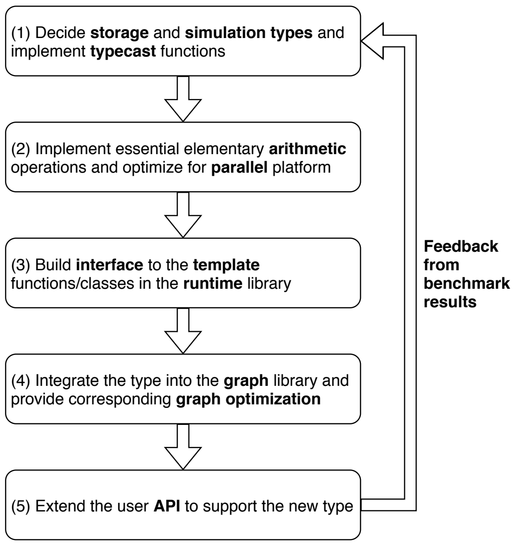

(1) Decide **storage** and **simulation types** and implement **typecast** functions
(2) Implement essential elementary **arithmetic** operations and optimize for **parallel** platform
(3) Build **interface** to the **template** functions/classes in the **runtime** library
(4) Integrate the type into the **graph** library and provide corresponding **graph optimization**
(5) Extend the user **API** to support the new type
Feedback from benchmark results

Figure 4. A flow chart of the process of implementing a new data type into a DL framework.

3 Challenges

Each step of implementation listed in Figure 4 imposes a different kind of challenge. Step 1 is mainly for a data type designer, who needs to propose the essential components of a data type based on heuristics. Step 2, 3, and 4 are what this paper focuses on, and we will discuss them in the following sections. Step 5 is beyond the scope of this paper.

We particularly examine the implementation of one data type, `bfloat16` (Brain Floating Point), which has 1 signed bit, 8 exponent bits, and 7 mantissa bits, which is a half truncated format from single-precision floating-point (`float`). Recently, `bfloat16` is proven to perform well in DNN [15] and already supported by TPU. Nevertheless, this paper looks specifically at the challenges in the development phase of `bfloat16`, when no native hardware support is available for it, while we need to evaluate its performance to decide whether it is appropriate for an accelerator.

3.1 Efficient Arithmetic Implementation

A mixed-precision DNN, even if its types can only be simulated, still needs high-performance execution. Otherwise, we cannot evaluate its accuracy efficiently on a baseline dataset that may have many thousands of images. To meet this challenge, at the runtime library level, we should ensure that the elementary arithmetic operations are efficiently implemented, which in practice depends on the following aspects:

(1) Scalar arithmetic operator, which is normally the starting point of implementation, should have low computation budget, e.g., calling only a small number of instructions. It is in the developers' common skill set.

```cpp
struct bfloat16 {
public:
  // float32 -> bfloat16
  explicit bfloat16(float v) {
    uint32_t u = *(
      reinterpret_cast<uint32_t*>(&v));
    uint32_t lsb = (u >> 16) & 1;
    uint32_t round_bias = 0x7fff + lsb;
    val = static_cast<uint16_t>(
      (u + round_bias) >> 16);
  }
  // bfloat16 -> float32
  explicit operator float() const {
    uint32_t v =
      static_cast<uint32_t>(this->val)
      << 16;
    return *(
      reinterpret_cast<float*>(&v));
  }
  // storage type
  uint16_t val;
};
```

(a) The minimal definition of `bfloat16` that a data type designer can come up with. It contains $\mathtt{T}_{\mathtt{sto}}$, $\mathtt{T}_{\mathtt{sim}}$, $\mathtt{f}_{\mathtt{sim}\to\mathtt{sto}}$, and $\mathtt{f}_{\mathtt{sto}\to\mathtt{sim}}$. In this snippet, $\mathtt{f}_{\mathtt{sim}\to\mathtt{sto}}$ implements a tricky rounding mechanism, and $\mathtt{f}_{\mathtt{sto}\to\mathtt{sim}}$ extends the storage value by zero to the least significant bits.

```cpp
inline bfloat16 operator+(
    bfloat16 a, bfloat16 b) {
  return bfloat16(static_cast<float>(a) +
                  static_cast<float>(b));
}
inline bfloat16 operator*(
    bfloat16 a, bfloat16 b) {
  return bfloat16(static_cast<float>(a) *
                  static_cast<float>(b));
}
inline bfloat16 operator/(
    bfloat16 a, bfloat16 b) {
  return bfloat16(static_cast<float>(a) /
                  static_cast<float>(b));
}

template <>
inline bfloat16 exp(const bfloat16& x) {
  return static_cast<bfloat16>(
    expf(static_cast<float>(x)));
}
template <>
inline bfloat16 abs(const bfloat16& x) {
  return static_cast<bfloat16>(
    fabsf(static_cast<float>(x)));
}
```

(b) Scalar-level arithmetic implemented. Some extend built-in operators, and others implement math functions.

```cpp
EIGEN_STRONG_INLINE __m64 to_bfloat16(__m128& x) {
  __m128i dx = _mm_castps_si128(x);
  // add the rounding base
  __m128i rb = _mm_set_epi32(
    0x7fff, 0x7fff, 0x7fff, 0x7fff);
  __m128i ma = _mm_set_epi32(
    0x10000, 0x10000, 0x10000, 0x10000);
  __m128i mx = _mm_and_si128(dx, ma);
  mx = _mm_add_epi32(rb,
    _mm_shufflelo_epi16(
      _mm_shufflehi_epi16(mx, 0xb1), 0xb1));

  __m128i dy = _mm_add_epi32(mx, dx);
  // shuffle back
  dy = _mm_shuffle_epi32(
    _mm_shufflelo_epi16(
      _mm_shufflehi_epi16(dy, 0x8d), 0x8d), 0xd8);
  return _mm_cvtsi64_m64(_mm_cvtsi128_si64(dy));
}
EIGEN_STRONG_INLINE __m128 to_float32(const __m64& x) {
  __m128i y = _mm_cvtsi64_si128(_mm_cvtm64_si64(x));
  y = _mm_shuffle_epi32(y, 0x72);
  y = _mm_shufflehi_epi16(y, 0xd8);
  y = _mm_shufflelo_epi16(y, 0xd8);
  return _mm_castsi128_ps(y);
}
template <>
EIGEN_STRONG_INLINE __m64 padd<__m64>(
  const __m64& a, const __m64& b) {
  return to_bfloat16(
    _mm_add_ps(to_float32(a), to_float32(b)));
}
```

(c) Vector-level arithmetics and typecast functions manually implemented in a way that Eigen can recognize.

Figure 5. Example of different aspects of an efficient implementation of data type `bfloat16`. By default we target Intel SSE instruction set for vectorization.

(2) To make use of the parallelism enabled by vector processing units, we should provide vectorized versions of these scalar-level functions, which is unfortunately not a skill generally accessible to most data type designers. Parallelizing code by SIMD instructions needs much knowledge of the target instruction set and requires exhaustive profiling.
(3) Since these implementations will be called by upper-level modules in the runtime library, e.g., Eigen, which extensively use templates to gain more performance but with the cost of losing readability and extendability, a developer cannot make an efficient implementation without understanding the codebase well.

Figure 5 illustrates different phases in the arithmetic implementation procedure. A data type designer is expected to propose Figure 5a in the beginning. Figure 5b shows the scalar arithmetic functions that follow almost the same pattern as Figure 3, and it is possible to generate these functions by a code template. Figure 5c is the most difficult: vectorized typecast functions have to originate from the human-designed typecast functions in Figure 5a and cannot be generated by code templates.

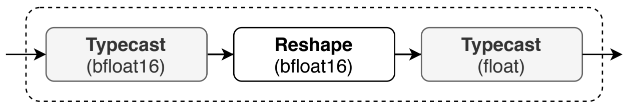
3.2 Recompilation and Code Bloating

We should explicitly specialize an operator template to support any new data type (Figure 2). It implies that whenever we implement or revise our data types, the whole DL framework codebase needs recompilation since almost all other components depend on data type definition. Considering the enormous scale of a modern DL framework codebase, the recompilation time is mostly intolerable. Intuitively, it is better to decouple parts that are affected by changes in data type from the rest of the codebase, such that we only need to recompile small code pieces for each revision.

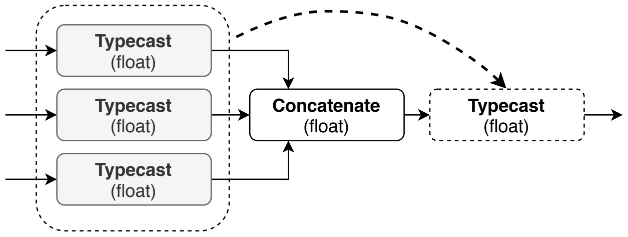
Another problem is code bloating. DNN operators need to be instantiated for each data type, which takes up space in the compiled binary file. Given the large set of operators supported by a DL framework, adding a new data type would greatly increase the binary size. It is possible to mitigate this problem: suppose we have a set of data types to be supported and a specific DNN only uses a subset of them, we can just compile those pairs of type and operator that are only required by the DNN we intend to run. However, we need Just-In-Time (JIT) compilation support since we know these pairs only when executing a DNN.

3.3 Typecast Nodes in Graph

Once we have the new type well optimized and fully integrated with the runtime library, we need further support at the graph level. A common scenario is that a user labels specific kinds of operators, e.g., convolution and matrix multiplication, by low-precision data types with the expectation of reducing overall DNN workload. There are quite a lot of other operators that users cannot deliberately specify their data types. Some of these operators need to run at high-precision, e.g., softmax, while others are type-agnostic, e.g., transpose, which give an identical result with any data type. How to decide the types of the latter group of operators has an impact on performance through typecast nodes inserted.



(a) An example of case 1. *Reshape* is a type-agnostic operator. Names in brackets denote the types of node output. Here we can remove all typecast nodes and change the type of reshape directly to `float`.



(b) An example of case 2. *Concatenate* is an operator that joins multiple input tensors into one, which is also type-agnostic. Here we show that instead of doing three typecast in the input, moving the typecast to the output and change the type of concatenate to `bfloat16` is beneficial.

Figure 6. Two example showcase graph transformations we perform to eliminate non-essential typecast nodes.

A typecast node simply converts the type of the input data to its target type. It needs to iterate through every element in the input and call typecast functions, as well as allocating memory space for its output. It is almost always better to remove non-essential typecast nodes. Figure 6 shows two cases that typecast nodes can be removed, both are around type-agnostic operators. In the first case, the inner operator can produce identical results with different types, such that there is no need to typecast. Nodes in the second case have typecast nodes on either the side of the input or the output, and we can move them back and forth in case the typecast workload is different on each side. Neither of these cases can be optimized by existing graph optimization passes.

4 The Lowgen Framework

This paper proposes Lowgen to address these challenges. Lowgen aims to reduce the overall workload of implementing and experimenting with new data types in a mixed-precision DNN scenario. It is not a standalone DL framework; instead, it is designed to be integrated into any DL framework that has a similar architecture to TensorFlow. With Lowgen, users who intend to develop a new data type only need to specify the essential parts, i.e., $\mathtt{T}_{\mathtt{sim}}$, $\mathtt{T}_{\mathtt{sto}}$, $\mathtt{f}_{\mathtt{sim}\to\mathtt{sto}}$, and $\mathtt{f}_{\mathtt{sto}\to\mathtt{sim}}$. Lowgen can generate efficient arithmetic implementation, provide a JIT compilation mechanism to dynamically specialize operator templates, and optimize mixed-precision DNN graphs. We look at each part in the following sections, still using `bfloat16` as the target data type.

4.1 Generate High-Performance Data Type

Based on our previous example (Figure 5), Lowgen only expects users to provide Figure 5a and it is responsible to generate the rest with high quality. Specifically, the user input to Lowgen is a piece of C++ code that contains one class that defines the new data type. That class should have three essential parts: one data field of the storage type $\mathtt{T}_{\mathtt{sto}}$, an explicit constructor that takes a single argument of the simulation type $\mathtt{T}_{\mathtt{sim}}$, which is given by $\mathtt{f}_{\mathtt{sim}\to\mathtt{sto}}$, and an explicit typecast function $\mathtt{f}_{\mathtt{sto}\to\mathtt{sim}}$.

Lowgen generates high-performance data type by source-to-source transforming this input file. The transformation should generate elementary arithmetic functions, both in scalar and vector forms, and vectorized typecast functions, $\mathtt{f}_{\mathtt{sim}\to\mathtt{sto}}$ and $\mathtt{f}_{\mathtt{sto}\to\mathtt{sim}}$. Generating arithmetic functions is trivial to implement, while generating vectorized typecast functions is challenging. Lowgen addresses this challenge by a novel method which is simple and effective: we directly map each instruction in the scalar typecast functions to its vectorized counterparts. The rest of this section presents the intuition behind this approach and its implementation.

Intuition. A typecast function normally consists of a simple dataflow, which has a single source, i.e., the scalar data input to be typecast, and a single sink, i.e., the scalar output of the target type (ignoring all constants). This property suggests that executing a typecast function on multiple independent input scalars in parallel will not incur race condition. Therefore, replacing scalar instructions by their vectorized counterparts, which practically executes multiple continuously addressed input sources in parallel, is a viable solution to vectorize the whole typecast function. We only need to perform a sanity check on the aforementioned dataflow condition before running the replacement.

Implementation. Given the AST parsed by Clang [18], Lowgen iterates every AST node in the typecast function declaration and generate code accordingly. If the node being visited is a variable declaration, then in the vectorized function body, we create a corresponding vector variable. If we come across a unary or binary operator, and their vectorized counterparts are available in the target instruction set, we then directly carry out the replacement. If there is an if-else construct, we first generate vector instructions for all branches and then create a vector mask calculated from the branch condition to aggregate branch results. There are cases that multiple AST nodes should be mapped to a single vector instruction, or a single node needs to be implemented

Table 1. Comparison of the overall forward execution time (ms) per input sample between `float`, `bfloat16` without any optimization in Lowgen, and `bfloat16` implementation generated by Lowgen. We also list the speedup of `bfloat16` models compared with the baseline.

Model	Task	`float`	`bfloat16` (Lowgen)	`bfloat16` (baseline)	Speedup
ResNet-v1-50 [12]	Classification	624.61	2312.38	17517.80	7.576
ResNet-v1-101 [12]	Classification	1143.13	4923.54	37500.10	7.616
ResNet-v2-50 [13]	Classification	990.51	3425.27	32049.40	9.357
Inception-v4 [30]	Classification	1267.38	5913.06	44277.00	7.488
YOLO-v3-Tiny [26]	Detection	525.66	1780.26	14557.60	8.177
YOLO-v3 [26]	Detection	3813.83	14834.20	164261.00	11.073
SSD [20]	Detection	301.66	928.24	7671.72	8.265
SSD-FPN [19]	Detection	5976.73	29243.90	301022.00	10.293
Faster-RCNN [27]	Detection	52013.30	134392.00	1244230.00	9.258
DeepLab-v3 [4]	Segmentation	10329.90	41503.90	468629.00	11.291
U-Net [28]	Segmentation	64139.40	59886.10	65217.4	1.089

by a sequence of vector instructions. Lowgen provides ad-hoc pattern matchers (inherited from `ASTMatcher` in Clang) and code templates to handle such cases.

4.2 Just-In-Time Operator Template Specialization

Once we have the data type generated, we then need to specialize DNN operator templates with it. Our objective is to perform the specialization right after the DNN model information is given, such that we can just specialize those DNN operators that actually need this new type. Lowgen provides a novel JIT compilation mechanism for this purpose.

1. We first fetch all pairs of operator and type in the given DNN model and identify which operators need $\mathsf{T}_{\mathsf{new}}$.
2. Next, we re-assemble the operator implementation, typically a template class, into a temporary file, and we specialize this template by $\mathsf{T}_{\mathsf{new}}$.
3. Finally, we compile the generated code into a shared library. Given the user API is in Python, we can load the generated shared library into the current Python process and use it to execute the mixed-precision DNN.

This technique has limitations though, especially when we cannot access the operator template source file that includes the declaration, or when the frontend is not Python and cannot load shared libraries during execution. We will look for alternatives in future work.

4.3 Mixed-Precision Graph Optimization

We now have the data type implemented and imported, and the next step is to experiment with it. A common experimentation scenario is that, given a reference DNN model, we intend to mark some of its nodes to use $\mathsf{T}_{\mathsf{new}}$ and see its performance. Intuitively, we need to insert typecast nodes around these marked nodes since they have incompatible types with their peripherals. But as we mentioned earlier, there might be redundancy in these auto-inserted typecast nodes and we can transform the mixed-precision graph to get rid of them. Lowgen implements the two types of transformations mentioned in Figure 6.

1. *Type 1.* We examine all pairs of typecast nodes in the graph and see whether all the paths that connect them are type-agnostic. If so, we remove both typecast nodes and change the types of nodes on these paths.
2. *Type 2.* We iterate every type-agnostic node that has adjacent typecast nodes and evaluate whether the expected typecast workload may reduce after moving typecast nodes to the other end. The expected workload is calculated simply by counting the total number of elements to be typecast.

5 Empirical Evaluation

We evaluate Lowgen on one specific scenario: implementing `bfloat16` and experimenting its performance on various DNN models. Instead of integrating Lowgen directly into TensorFlow, we use our own in-house DL framework that has similar architecture to TensorFlow but is much easier to extend. Our framework has its own implementation of the runtime library based on Eigen, the graph library that can construct and optimize TensorFlow-like DNN graphs, and a Python-based user API. Our framework also supports converting DNN model representation to and from TensorFlow graphs.

Our experiments are carried out on a wide range of DNN models, which have state-of-the-art accuracy on tasks including image classification, object detection and semantic segmentation. We first showcase the overall performance gain we can get from using Lowgen, which is significant. Next, besides the reduction on overall execution time, we describe the following benefits of Lowgen: we compare the performance measured in runtime and lines of code of the

Lowgen generated `bfloat16` definition and a manually written one; we examine the effectiveness of dynamic template instantiation for DNN operators, mainly about the compilation time and binary size; and we study whether applying the mixed-precision graph generation can practically reduce the overall execution time.

5.1 Experimental Setup

The experimental platform is Intel Xeon Silver 4110 CPU that has 8 cores and runs at 2.10 GHz base frequency. We target the Intel Streaming SIMD Extensions (SSE) instruction set when generating vectorized functions. The in-house DL framework itself is compiled by GCC v5.4.0 with the `-O3` flag turned on. The Eigen module integrated into our DL framework is v3.3.7 and compiled with OpenMP to support multi-threading. By default, all the experiments in this paper use 2 inter-operator threads to run DNN operators in parallel, and 8 intra-operator threads that Eigen will employ to parallelize its BLAS routines.

5.2 Overall Execution Time

Table 1 summarizes the execution time measured on various DNN models. All these models are first implemented and evaluated by our in-house DL framework with `float`. Then we build a baseline `bfloat16` implementation, which includes the class definition and scalar arithmetic functions declaration (Figures 5a and 5b). No vectorization is supported in this case. Finally, we use Lowgen to generate a vectorized `bfloat16` definition and collect the performance. Typecast nodes in the mixed-precision DNN graph are optimized in both cases. These performance numbers exclude the trivial JIT overhead.

From Table 1 we first notice that the speed-up we can get from Lowgen is significant. It mainly comes from the extensive vectorization that Lowgen generates. Intel SSE supports 128bit width vectors, which can give about 4 times speedup, given that the arithmetic functions are simulated by 32bit `float`. The additional speedup comes from scheduling: both the internal scheduler of Eigen and our inter-operator scheduler can parallelize the execution efficiently. This speedup is significant: before optimized by Lowgen, we need normally several hours to finish one round of evaluation; with the help from Lowgen, we can usually obtain the result within an hour.

Meanwhile, we notice that `bfloat16` with Lowgen still runs around 4 times slower than `float`, which is understandable since `bfloat16` DNN models are simulated by `float`. `bfloat16` DNN workload consists of everything from its `float` counterpart, with additional overhead that cannot be avoided, e.g., typecasting. The major objective of Lowgen is not to run DNN faster in low-precision types than `float`; we are interested in getting the simulation more efficient than a naive implementation with reduced development cost. One outlier is U-Net, which is a memory-bound DNN model based on our implementation, and therefore, `float` is not much faster than others and can have worse performance.

5.3 Other Lowgen Benefits

Lines of code. Take the `bfloat16` implementation as an example, to get the fully optimized performance, we need roughly 600 lines of code, including the data type definition, scalar and vectorized arithmetic functions, and optimized typecast functions. With help from Lowgen, we only need around 30 lines of code that specify the core structure: $\mathtt{T}_{\mathtt{sto}}$, $\mathtt{f}_{\mathtt{sim}\to\mathtt{sto}}$, and $\mathtt{f}_{\mathtt{sto}\to\mathtt{sim}}$.

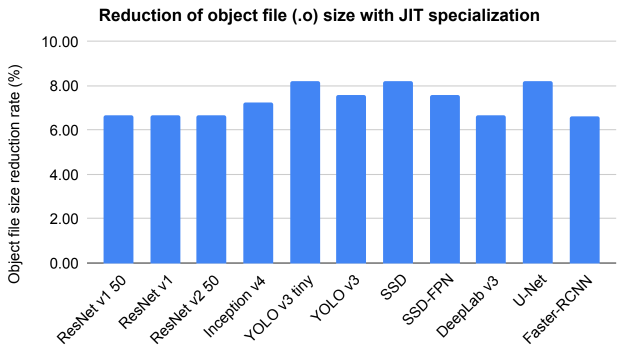


Figure 7. Measuring the reduction rate of the size of compiled object files (*.o) on different DNN models due to Lowgen.

Size of compiled operators. Figure 7 summarizes the reduction of compiled object file size when the JIT template specialization technique is applied. Different DNN models have different operators marked as `bfloat16`, such that they can benefit at different scales from Lowgen.

Table 2. Comparing the peak memory usage of running SSD and Inception-v4 with (opt) and without (base) typecast optimization. ↓ # Typecast denotes the number of typecast nodes eliminated.

Model	Mem. (base)	Mem. (opt)	↓ # Typecast
SSD	89.6 MB	88.9 MB	12
Inception-v4	578.4 MB	572.9 MB	93

Graph optimization. With redundant typecast nodes eliminated, we expect that mixed-precision DNN can run faster with a smaller memory footprint. There are two DNN models that can benefit much from this optimization, SSD and Inception-v4. They both have many type-agnostic nodes: SSD uses a lot of reshape nodes before its output, and there are extensive concatenate nodes in Inception-v4. Table 2 lists the comparison result.

6 Related Work

There are many aspects of the PL research that are related to our work. First, the high-performance data type generator in Lowgen is highly related to auto-vectorization, a technique that converts loops that execute one element at a time to those process multiple elements in parallel through vector instructions. It is an extensively studied method [8, 17, 21] and accessible in recent compiler releases. However, since there is no loop structure in the data type definition input to Lowgen, auto-vectorization is not an applicable method and we find a simpler solution to perform vectorization.

The concept of JIT compilation [3] is adapted in our approach as well. In DL community, people normally use JIT to enhance performance during runtime by compiling and optimizing segments of code and customize DNN implementation [9, 23, 25]. JIT in Lowgen has a similar motivation but with a different perspective: we intend to customize the DL workload relating to mixed-precision support for DNN operators based on runtime information, such as the types in the DNN models to be executed.

There is much recent development in DL compilers that aims to further improve the execution performance targeting various platforms [5, 29, 32]. Lowgen uses similar techniques as in those DL compilers, while it concentrates on improving mixed-precision DNN performance, which has not been effectively supported by them.

7 Summary

This paper discusses the programming challenges in implementing and evaluating mixed-precision DNN models when using state-of-the-art DL frameworks. We show that even though the highly modularized framework architecture enhances the performance, it prevents users to efficiently design and implement new data types and evaluate them in mixed-precision DNNs. We list three specific challenges including the difficulties in implementing high-performance vectorized data type definition, the recompilation and code bloating issues caused by specializing templates for new types, and the redundant typecast nodes that can be eliminated. We summarize our preliminary solutions into a framework, Lowgen, which has a high-performance code generator for efficient data types, a JIT compilation mechanism that can specialize operator templates during runtime, and a graph optimizer that transforms mixed-precision DNN graphs to remove unnecessary or resource-consuming typecast nodes. Lowgen can support any TensorFlow-like DL framework to extend their ability for handling mixed-precision DNNs. For a specific data type `bfloat16`, we have empirically evaluated Lowgen on a wide range of DNN models, and the results look promising: we can get on average 8.317 times speed up over the baseline by using Lowgen.

Much future work can be carried out. First of all, Lowgen is evaluated on our in-house DL framework, and we look forward to testing it on the official TensorFlow codebase. Also, `bfloat16` is a rather simple custom data type: it has exactly 2 bytes and the typecast between it and `float` is not very complicated. Other data types may not have byte-aligned storage, and may use non-trivial typecast functions. Extending our approach to support additional applications and other SIMD platforms is also on our agenda.

Acknowledgements

The support of the U.K. EPSRC (grant number EP/L016796/1, EP/N031768/1, EP/P010040/1, EP/S030069/1 and EP/L00058X/1) is gratefully acknowledged.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Martín Abadi, Michael Isard, and Derek G. Murray. A computational model for TensorFlow an introduction. In *MAPL*, 2017. ISBN 9781450350716. doi: 10.1145/3088525.3088527.

[3] John Aycock. A Brief History of Just-In-Time. *ACM Computing Surveys*, 35(2):97–113, 2003. ISSN 0449749X.

[4] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2017.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: End-to-End Optimization Stack for Deep Learning. *arXiv preprint arXiv:1802.04799*, 2018.

[6] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[7] Zhen Dong, Zhewei Yao, Amir Gholami, Michael Mahoney, and Kurt Keutzer. HAWQ: Hessian AWare Quantization of Neural Networks with Mixed-Precision. In *ICCV*, pages 293–302, 2019. URL http://arxiv.org/abs/1905.03696.

[8] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *PLDI*, 2004. ISBN 1581138075. doi: 10.1145/996841.996853.

[9] Google. JAX: Autograd and XLA. https://github.com/google/jax, 2020.

[10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[11] Song Han, Huizi Mao, and William J. Dally. Deep Compression - Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *ICLR*, 2016.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity Mappings in Deep Residual Networks. In *ECCV*, 2016.

[14] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-Luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon Mackean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, 2017.

[15] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A Study of BFLOAT16 for Deep Learning Training. 2019. URL http://arxiv.org/abs/1905.12322.

[16] Hamed F. Langroudi, Zachariah Carmichael, David Pastuch, and Dhireesha Kudithipudi. Cheetah: Mixed Low-Precision Hardware & Software Co-Design Framework for DNNs on the Edge. pages 1–13, 2019. URL http://arxiv.org/abs/1908.02386.

[17] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, 2000. ISBN 1581131992. doi: 10.1145/358438.349320.

[18] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, volume 5, 2008.

[19] Tsung Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *CVPR*, 2017. ISBN 9781538604571. doi: 10.1109/CVPR.2017.106.

[20] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. SSD: Single Shot MultiBox Detector. In *ECCV*, 2016. URL https://arxiv.org/pdf/1512.02325.pdf.

[21] Saeed Maleki, Yaoqing Gao, María J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *PACT*, 2011. ISBN 9780769545660. doi: 10.1109/PACT.2011.68.

[22] Naveen Mellempudi, Sudarshan Srinivasan, Dipankar Das, and Bharat Kaul. Mixed Precision Training With 8-bit Floating Point. 2019. URL http://arxiv.org/abs/1905.12334.

[23] Numba. A just-in-time compiler for numerical functions in python. https://github.com/numba/numba, 2020.

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*, 2019.

[25] PyTorch. Torchscript. https://pytorch.org/docs/stable/jit.html, 2020.

[26] Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement. 2018. URL http://arxiv.org/abs/1804.02767.

[27] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *NeurIPS*, pages 91–99, 2015.

[28] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation Olaf. In *MICCAI*, 2015. doi: 10.1007/978-3-319-42999-1_15.

[29] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph Lowering Compiler Techniques for Neural Networks. 2018. URL http://arxiv.org/abs/1805.00907.

[30] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *AAAI*, 2017. URL https://arxiv.org/abs/1602.07261.

[31] TensorFlow. TensorFlow graph optimization with Grappler. https://www.tensorflow.org/guide/graph_optimization, 2020.

[32] Richard Wei, Vikram Adve, and Lane Schwartz. DLVM: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*, 2017.