Wenyu Pan

Study Report for Gradient Compression Supercharged High-Performance Data-Parallel DNN Training and TernGrad Algorithm

I. Introduction

In a typical setting, each node iterates over its own data partition in parallel and exchanges a large volume of gradients with other nodes per iteration via a gradient synchronization strategy like Parameter Server of Ring-all reduce. However, in recent years, the fast-growing computing capability, driven by the booming of GPU architecture innovations and domain-specific compiler techniques, tends to result in more frequent and heavier gradient synchronization during data-parallel DNN training.

Gradient compression algorithms have a great potential to relieve or even eliminate the above tension since they can substantially reduce the data volume being synchronized with a negligible impact on training accuracy and convergence.

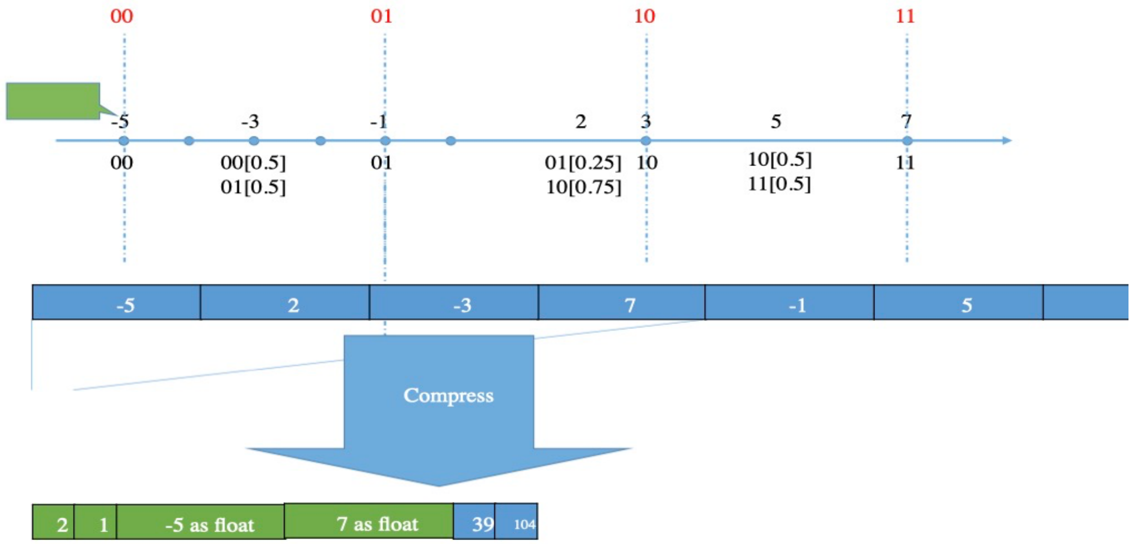

Figure 1. Overall Flow of TernGrad Algorithm

II. TernGrad Algorithm:

In the graph above, TernGrad maps and compresses the input array of float (-5, 2, -3, 7, -1, 5) into a heterogeneous array of float and unit_8, where 2 marks the bit width, 1 marks the tail of the array, -5 and 7 marks the min and max value of the array. During the quantification process, -5, -1, 3, and 7 are marked as 00, 01, 10, and 11. If the random is false, since 2 is much closer to 3(10), 2 is automatically quantified to 3 and so as the other intermediate values. If the random is false, there is a 25% of chance that 2 will be quantified to -1(01) and 75% that 2 will be quantified to 3(10). 39 is for 00100111, which is for the first four float values of the input. (-5,2,-3,7). 104 marks the last three float values of the list: -1, 5, and 3 with a tail we added manually to the list (00). Therefore, the output array is {2,1, -5,7,39,104}.

III. Code Realization in Detail

3.1 TernGrad Algorithm

Code | **Code Explanation**

```
struct compress_without_random{
  float* in_float;
  uint8_t bitwidth;
  uint8_t data_per_byte_lg2;
  float min_val;
  float gap_inverse;
  compress_without_random(
    float* a,
    uint8_t b,
    uint8_t c,
    float d,
    float e
  ){
    in_float = a;
    bitwidth = b;
    data_per_byte_lg2 = c;
    min_val = d;
    gap_inverse = e;
  }
```

This struct is used for building the datatype for the context of compressing without random of Terngrad.

Fields of compress_without_random:
(1) float* in_float:
A pointer to the input array of float
(2)uint8_t bitwidth:
Bit width value of the input array
(3) uint8_t data_per_byte_lg2:
data_per_byte = 8 / bitwidth;
data_per_byte_lg2 = log(2, data_per_byte)
(3) float min_val:
Minimal value of the input array
(4) gap_inverse:
Float gap_inverse = 1 / gap

```
struct compress_with_random{
  float* in_float;
  uint8_t bitwidth;
  uint8_t data_per_byte_lg2;
  float min_val;
  float gap_inverse;
  unsigned long long timestamp;
  compress_with_random(
    float* a,
    uint8_t b,
    uint8_t c,
    float d,
    float e,
    unsigned long long f
  ){
    in_float = a;
    bitwidth = b;
    data_per_byte_lg2 = c;
    min_val = d;
    gap_inverse = e;
    timestamp = f;
  }
```

This struct is used for building the datatype for the context of compressing with random of Terngrad.

Fields of compress_without_random:
(1) float* in_float:
A pointer to the input array of float
(2)uint8_t bitwidth:
Bit width value of the input array
(3) uint8_t data_per_byte_lg2:
data_per_byte = 8 / bitwidth;
data_per_byte_lg2 = log(2, data_per_byte)
(3) float min_val:
Minimal value of the input array
(4) gap_inverse:
Float gap_inverse = 1 / gap
(5) unsigned long long timestamp:
Used in operator(uint_32 I) to generate a random number between 0 and (timestamp + I)

Main functions:

Code **Explanation**

Thrust::minmax_element finds the smallest and largest value between (in_floats.at(i) and in_floats.at(i) +in_float_sizes.at(i))

```
template <typename policy_t>
int terngrad_body(
    std::vector<float*> in_floats,
    std::vector<int32_t>
in_float_sizes,
    std::vector<uint8_t*>
out_uint8_ts,
    std::vector<int32_t> out_uint8_t_sizes,
    std::vector<uint8_t> bitwidths,
    std::vector<int32_t> randoms,
    policy_t policy,
    void *stream)
{

  int comp_size = in_floats.size();
  if(comp_size <= 0){
    return 0;
  }

  std::vector<std::shared_ptr<float>> min_vals;
  std::vector<std::shared_ptr<float>> max_vals;
```

Using memcpy to copy the min_value into (*min_values.at(i)) at copy max_value into (*max values.at(i))

```
for(int i = 0; i < comp_size; i++){
    float min_val, max_val;
    min_vals.push_back(std::make_shared<float>(min_val));
    max_vals.push_back(std::make_shared<float>(max_val));
    auto min_max = thrust::minmax_element(
      policy,
      in_floats.at(i),
      in_floats.at(i)+in_float_sizes.at(i)
    );

get_policy<policy_t>::memcpyOut(&(*min_vals.at(i)),min_max.first,sizeo
f(float),stream);

get_policy<policy_t>::memcpyOut(&(*max_vals.at(i)),min_max.second,size
of(float),stream); }

std::vector<float> gap_inverses;
  std::vector<uint8_t> data_per_byte_lg2s;
  std::vector<uint8_t> data_per_bytes;
  std::vector<uint8_t> tails;
  uint8_t lg2[9] = {0,0,1,1,2,2,2,2,3};
```

```
for(int i = 0; i < comp_size; i++){
    float gap = (*(max_vals.at(i)) - *(min_vals.at(i))) / ((1 <<
bitwidths.at(i)) - 1.0f);
    float gap_inverse = 1. / (gap + 1e-8);
    uint8_t bitwidth_lg2 = lg2[bitwidths.at(i)];

    if (unlikely((1<<bitwidth_lg2)!=bitwidths.at(i))){
      printf("Invalid value of bitwidth, chekc value:
bitwidth=%d\n",bitwidths.at(i)+0);
      return -1;
    }

    uint8_t data_per_byte_lg2 = 3 - bitwidth_lg2;
    uint8_t data_per_byte = 1<<data_per_byte_lg2;
    uint8_t tail = in_float_sizes.at(i) % data_per_byte;
    tail = tail ==  0 ? 0 : data_per_byte - tail;
```

Calculating the data_per_byte_lg2s, data_per_bytes, the tail and the header before further stepping into the compression.

```
    uint8_t header[10];
    ((float*)(header+2))[0] = *(min_vals.at(i));
    ((float*)(header+6))[0] = *(max_vals.at(i));
    header[0] = bitwidths.at(i);
```

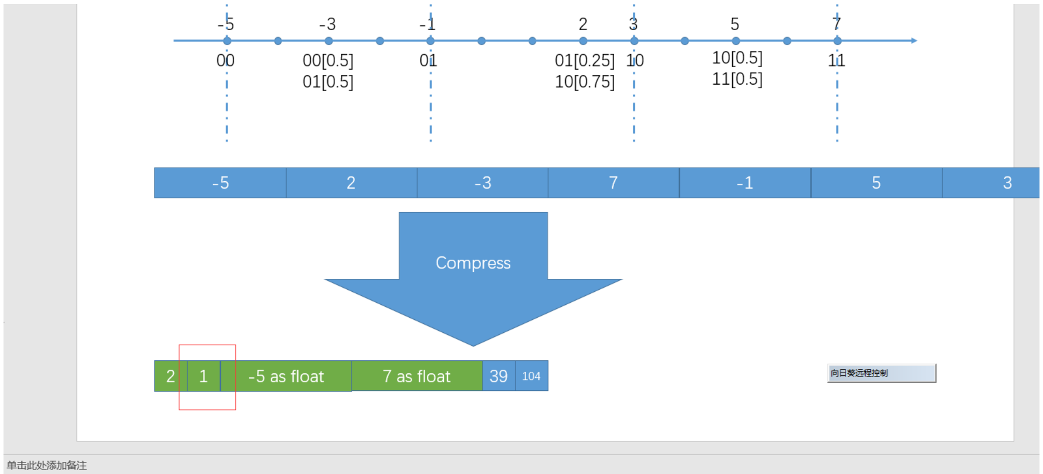

Figure 2. {Header}

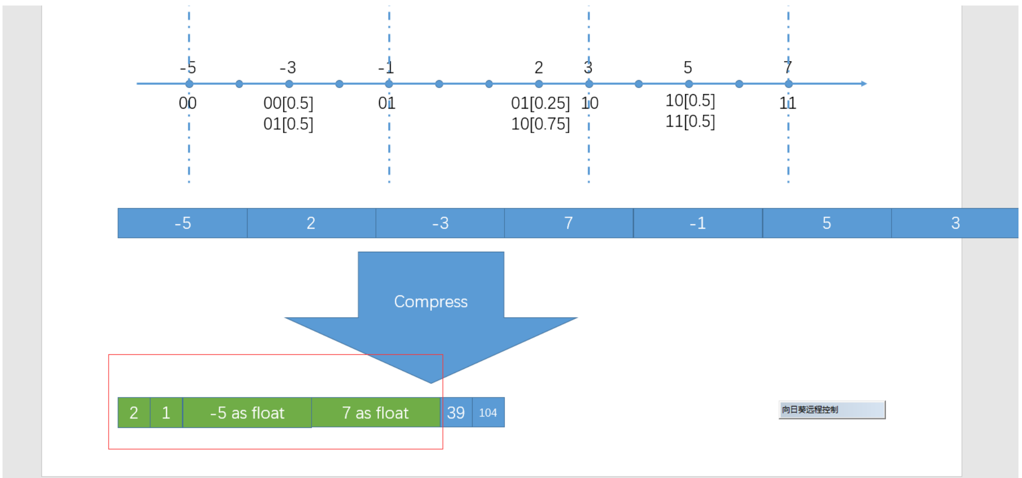

Figure 3. {Tail}

```
for(int i = 0; i < comp_size; i++){
    thrust::counting_iterator<int32_t> index_sequence_begin(0);
    if (randoms.at(i)){
      thrust::transform(
        policy,
        index_sequence_begin,
        index_sequence_begin + (in_float_sizes.at(i) >>
data_per_byte_lg2s.at(i)),
        out_uint8_ts.at(i)+10,
```

```
else{
      thrust::transform(
        policy,
        index_sequence_begin,
        index_sequence_begin + (in_float_sizes.at(i) >>
data_per_byte_lg2s.at(i)),
        out_uint8_ts.at(i)+10,
        compress_without_random(
          in_floats.at(i),
          bitwidths.at(i),
          data_per_byte_lg2s.at(i),
          *(min_vals.at(i)),
          gap_inverses.at(i)
        )
      );
    }
  }
```

sequence. out_uint8_ts.at(i)+10 is the beginning of the output sequence.
We then pass the input values into struct compress_with_random.
If the randoms.at(i) is equal to null, we pass the input values into the struct compress_without_random.

```
compress_with_random(
          in_floats.at(i),
          bitwidths.at(i),

data_per_byte_lg2s.at(i),
          *(min_vals.at(i)),
          gap_inverses.at(i),
          static_cast<unsigned
long long>(

std::chrono::high_resolution_clock::now()
     .time_since_epoch()
     .count()
    )
   )
  );
 }
```

If randoms.at(i) is not null, thrust::transform applies a unary function to each element of an input sequence and stores the result in the corresponding position in an output sequence. In this specific case, we apply out_uint8_ts.at(i)+10. index_sequence_begin is the beginning of the input sequence and index_sequence_begin + (in_float_sizes.at(i) is the ending of the input

```
for(int i = 0; i < comp_size; i++){
    float* tail_data = new float[8];
    if (tails.at(i)){

get_policy<policy_t>::memcpyOut(tail_data,in_floats.at(i)+(in_fl
oat_sizes.at(i)-(data_per_bytes.at(i)-
tails.at(i))),sizeof(float)*(data_per_bytes.at(i)-
tails.at(i)),stream);
    }
    tail_datas.push_back(tail_data);
  }
get_policy<policy_t>::streamSynchronize(stream);
```

Calculates the tail_datas vector.

```
  for(int i = 0; i < comp_size; i++){
    if (tails.at(i)){
      uint8_t qval = 0;
      auto tail_data = tail_datas.at(i);
      for (auto j = 0; j < data_per_bytes.at(i) - tails.at(i);
j++){
        uint8_t t = nearbyint((tail_data[j] -
*(min_vals.at(i)))*gap_inverses.at(i));
        qval = qval | ( t << (bitwidths.at(i)*j));
      };
```

IV. Conclusion:

HiPress driven by CaSync and CompLL addresses fundamentaltal tensions imposed by gradient compression. CaSync in- novates a general, composable, and adaptive gradient synchronization architecture that is compression-aware. CompLL facilitates easy development of highly-optimized on-GPU gradient compression and an automated integration into modern DNN systems with minimal manual efforts. HiPress is open-sourced and achieves a scaling efficiency of up to 0.91 and a training speed improvement up to 106.4% over the state-of-the-art baselines across six popular DNN models.

Reference

1. Wen, Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., & Li, H. (2017). TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning.