MIXED PRECISION TRAINING ON VOLTA GPUS

Boris Ginsburg, Paulius Micikevicius, Oleksii Kuchaiev, Sergei Nikolaev
bginsburg, pauliusm, okuchaiev, snikolaev@nvidia.com
10/17/2017
ACKNOWLEDGMENTS

Michael Houston, Hao Wu, Ahmad Kiswani, Amir Gholaminejad, Ujval Kapasi, Jonah Alben, Alex Fit-Florea, Slawomir Kierat

cuDNN team

Work is based on NVIDIA branch of caffe
https://github.com/NVIDIA/caffe (caffe-0.16)
AGENDA

1. Mixed precision training with Volta TensorOps
2. Training without “master-copy” weights
3. NVCAFFE support for FLOAT16
## TERMINOLOGY

<table>
<thead>
<tr>
<th>Training type</th>
<th>Data type</th>
<th>Matrix-Multiply Accumulator</th>
<th>Weight update</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
</tr>
<tr>
<td>“Pascal” FP16</td>
<td>FP16</td>
<td>FP16</td>
<td>FP16/FP32</td>
<td>Pascal(GP-100)</td>
</tr>
<tr>
<td>Mixed precision</td>
<td>FP16</td>
<td>FP32</td>
<td>FP16/FP32</td>
<td>Volta</td>
</tr>
</tbody>
</table>

“Master” weights - copy of weights used for update (SGD step).

Volta: Mixed precision training with FP32 master weight storage.
HALF-PRECISION FLOAT (FLOAT16)

FLOAT16 has wide range ($2^{40}$) ... but it is not as wide as FP32!

Normal range: $[ 6 \times 10^{-5} , 65504 ]$
Sub-normal range: $[ 6 \times 10^{-8} , 6 \times 10^{-5} ]$
VOLTA: TENSOR CORE 4X4 MATRIX-MULTIPLY ACC

\[
D = \begin{pmatrix}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\
A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\
A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3}
\end{pmatrix} + \begin{pmatrix}
B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\
B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\
B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\
B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3}
\end{pmatrix} + \begin{pmatrix}
C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\
C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\
C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\
C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3}
\end{pmatrix}
\]

FP16 or FP32  FP16  FP16  FP16 or FP32
VOLTA TENSOR OPERATION

FP16 storage/input  Full precision product  Sum with FP32 accumulator  Convert to FP32 result

Also supports FP16 accumulator mode for inferencing
TRAINING FLOW

FORWARD PASS

\[ Y_k = W_k \cdot Y_{k-1} \]

\[ Y_2 = W_2 \cdot Y_1 \]

\[ Y_1 = W_1 \cdot X \]

\[ X \]

Loss

WEIGHT UPDATE

\[ W_k = W_k - \lambda \cdot \frac{dE}{dW_k} \]

\[ \frac{dE}{dY_k} = \frac{dE}{dY_k} \cdot W_k \]

\[ \frac{dE}{dW_k} = \frac{dE}{dY_k} \cdot Y_{k-1} \]

\[ \frac{dE}{dY_1} = \frac{dE}{dY_2} \]

\[ Y_1 = W_1 \cdot X \]

\[ Y_2 = W_2 \cdot Y_1 \]

\[ X \]
TRAINING FLOW

\[ Y_k = W_k Y_{k-1} \]

\[ Y_2 = W_2 Y_1 \]

\[ Y_1 = W_1 X \]

\[ \frac{dE}{dW_k} = \frac{dE}{dY_k} Y_{k-1} \]

\[ \frac{dE}{dW_2} = \frac{dE}{dY_2} Y_1 \]

\[ \frac{dE}{dW_1} = \frac{dE}{dY_1} X \]

\[ \frac{dE}{dX} = \frac{dE}{dY_1} W_1 \]

\[ \frac{dE}{dY_1} = \frac{dE}{dY_2} W_2 \]

\[ \frac{dE}{dY_2} = \frac{dE}{dY_3} W_3 \]

\[ \frac{dE}{dY_k} = \frac{dE}{dY_{k-1}} W_k \]

\[ \frac{dE}{dY_{k-1}} = \frac{dE}{dY_{k-2}} W_{k-1} \]

\[ \frac{dE}{dY_1} = \frac{dE}{dY_2} W_2 \]

\[ \frac{dE}{dY_2} = \frac{dE}{dY_3} W_3 \]

\[ \frac{dE}{dY_3} = \frac{dE}{dY_4} W_4 \]

\[ \frac{dE}{dY_k} = \frac{dE}{dY_{k-1}} W_k \]
**TRAINING FLOW**

**FORWARD PASS**
- $Y_k = W_k \cdot Y_{k-1}$
- $Y_2 = W_2 \cdot Y_1$
- $Y_1 = W_1 \cdot X$

**BACKPROP**
- $\frac{dE}{dY_{k-1}} = \frac{dE}{dY_k} \cdot W_k$
- $\frac{dE}{dW_k} = \frac{dE}{dY_k} \cdot Y_{k-1}$
- $\frac{dE}{dY_1} = \frac{dE}{dY_2} \cdot W_2$
- $\frac{dE}{dW_2} = \frac{dE}{dY_2} \cdot Y_1$
- $\frac{dE}{dY_2} = \frac{dE}{dY_1} \cdot W_1$
- $\frac{dE}{dW_1} = \frac{dE}{dY_1} \cdot X$

**WEIGHT UPDATE**
- $W_k = W_k - \lambda \cdot \frac{dE}{dW_k}$
- $W_2 = W_2 - \lambda \cdot \frac{dE}{dW_2}$
- $W_1 = W_1 - \lambda \cdot \frac{dE}{dW_1}$

Loss $E$
TRAINING FLOW

FORWARD PASS

\[ Y_k = W_k \cdot Y_{k-1} \]

\[ Y_2 = W_2 \cdot Y_1 \]

\[ Y_1 = W_1 \cdot X \]

BACKPROP

\[ \frac{dE}{dY_k} = \frac{dE}{dY_{k-1}} \]

\[ \frac{dE}{dY_2} = \frac{dE}{dY_1} \cdot W_2 \]

\[ \frac{dE}{dY_1} = \frac{dE}{dX} \cdot W_1 \]

WEIGHT UPDATE

\[ W_k = W_k - \lambda \cdot \frac{dE}{dW_k} \]

\[ W_2 = W_2 - \lambda \cdot \frac{dE}{dW_2} \]

\[ W_1 = W_1 - \lambda \cdot \frac{dE}{dW_1} \]

FORWARD PASS

\[ Y_k = W_k \cdot Y_{k-1} \]

\[ Y_2 = W_2 \cdot Y_1 \]

\[ Y_1 = W_1 \cdot X \]
VOLTA TRAINING METHOD

Master-W (F32) → W (F16) → W → Y_k → FWD → Y_{k+1} → BWD-A → \frac{dE}{dY_k} → FWD → Y_k → BWD-W → \frac{dE}{dW} → F32 → Weight Update → F32 → Updated Master-W
SOME NETS WORKED OUT OF THE BOX

TensorOp training matched the results of F32 training
  Same solver, hyper-parameters, and training schedule as F32

Image classification (ILSVRC12):
  GoogleNet, VGG-D, Inception v3, ResNet-50
  Solver: SGD with momentum

Language modeling and machine translation
  NMT
  Solver: ADAM
MIXED PRECISION TRAINING

Networks that don’t work “out-of-box”:

Image classification: Alexnet, CaffeNet

Detection nets:
- Multibox SSD with VGG-D backbone: was not learning
- Faster R-CNN with VGG-D backbone: low mAP 68.5% vs 69.1% with F32

RNNs:
- Seq2seq with attention: lagged behind F32 in perplexity
- bigLSTM: diverged after some training
MULTI-BOX SSD: ACTIVATION GRADIENT HISTOGRAM
MULTI-BOX SSD: ACTIVATION GRADIENT HISTOGRAM

Become denormals in F16
Become 0 in F16

activation gradient magnitudes

Percentage of values during training

Upper bound, 2 to the listed exponent
MULTIBOX SSD: ACTIVATION GRADIENT HISTOGRAM

Become 0 in F16

Become denormals in F16

Unused

Overall FP16 range
MIXED PRECISION TRAINING: GLOBAL LOSS SCALING

The problem is small ("vanishing") activation gradients

Solution:
1. scale the loss value by $K$ to "shift" gradients to FP16 range
2. rescale gradients back by $1/K$ before weight update
3. New hyper-parameter: loss scale
SCALING LOSS AND GRADIENTS

Loss scaled by 256

=> gradients get scaled by 256

Benefits:

shift activation gradients into working range

Most weight gradients become normalized values in F16
# AlexNet: Comparison of Results

<table>
<thead>
<tr>
<th>Mode</th>
<th>Top1, %</th>
<th>Top5, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fp32</td>
<td>58.62</td>
<td>81.25</td>
</tr>
<tr>
<td>FP16 training, no scaling</td>
<td>56.70</td>
<td>78.12</td>
</tr>
<tr>
<td>FP16 training, scale = 1000</td>
<td>58.9</td>
<td>81.10</td>
</tr>
<tr>
<td>Mixed precision training, scale = 1000</td>
<td>59.1</td>
<td>81.2</td>
</tr>
</tbody>
</table>

Nvcaffe-0.16, DGX1, SGD with momentum, 100 epochs, batch=1024, no augmentation, 1 crop, 1 model
ALEXNET: TRAINING WITH SCALING

FP16 training matches other training curves (TensorOp and FP32)
OBJECTS DETECTION

Multibox-SSD mAP:

F32: 76.9%
TensorOp: 77.1%, loss scaled by 256;
Without scaling: doesn’t learn

Faster-RCNN mAP:

F32: 69.1%
TensorOp: 69.7%, loss scaled by 256,
without scaling: 68.5%
SEQ2SEQ TRAINING

Neural machine translation: Ge-En

1. [https://github.com/NVIDIA/OpenSeq2Seq](https://github.com/NVIDIA/OpenSeq2Seq)

2. NMT_ONE model:
   1. 2-layer bi-directional encoder (512 LSTM)
   2. Normalized Bahdanau attention
   3. 4-layer decoder (512 LSTM)
SEQ2SEQ TRAINING

OpenSeq2Seq: training loss

Scaling recovered accuracy
SEQ2SEQ TRAINING

scaling factor: 32,768

How to reduce scaling factor?
OPEN SEQ2SEQ LOSS

Logits shape: [batch_size, time, dimension]

Targets shape: [batch_size, time]

\[
\text{Loss}_{\text{avg}} = \text{Avg}_{\text{batch}}(\text{Avg}_{\text{timesteps}}(\text{crossentropy}(\text{logits}, \text{targets})))
\]

\[
\text{Loss}_{\text{sum}} = \text{Avg}_{\text{batch}}(\text{SUM}_{\text{timesteps}}(\text{crossentropy}(\text{logits}, \text{targets})))
\]

More numerically-friendly loss function

LARS algorithm allows smaller scale factors
SEQ2SEQ TRAINING

OpenSeq2Seq training: Eval BLEU score

With sum loss and LARS scaling factor: 512
SEQ2SEQ TRAINING

With sum loss and LARS scaling factor: 1024

GNMT-like

- 8-layer bi-directional encoder (1024 LSTM)
- GNMT-style normalized Bahdanau attention
- 8-layer decoder (1024 LSTM)
- residual connections

1 Billion Word Language Benchmark

BigLSTM:

2x8192 LSTM, 1024 Projection

800K word vocabulary

Adagrad solver

BIGLSTM: 2X8192 LSTM, 1024 PROJECTION

Graph showing the performance of different models over iterations. The x-axis represents the number of iterations (in thousands), ranging from 0K to 2,000K, and the y-axis represents a value ranging from 2.5 to 5.0. The graph compares F32, Volta, scale=1, and Volta, scale=128 models.
GUIDELINES FOR TRAINING
WITH MIXED PRECISION / TENSOROPS
VOLTA MIXED PRECISION TRAINING

Mixed Precision Training:
• FP16 storage and TensorOps for fwd/bwd pass:
  – weights, activations, gradients
• Batch Normalization: FP16 data, FP32 math
• FP32 “master-copy” of weights for weights update

WARNING
• Gradient may become too small for FP16 range
• Solved with new hyper parameter “loss scaling”
OBSERVATIONS ON GRADIENT VALUES

FP16 range: $2^{40}$ including denoms

Gradient range is biased low vs standard FP16 range

Max magnitude we’ve seen was $O(2^3)$

“shift” values without overflowing

Maximum magnitudes:

weight-grad $>>$ activation-grad

* for all the nets we’ve looked at
MIXED PRECISION TRAINING
WITHOUT MASTER WEIGHT COPY
ALEXNET: FP16 MASTER WEIGHT STORAGE

Can we avoid two weights copies? Can FP16 be used for weight update?

“Vanilla” SGD weights update:

$$W(t+1) = W(t) - \lambda \Delta W(t)$$

If we use FP16 for $\Delta W$, the product $\lambda \Delta W(t)$ can become too small:

Initially gradients $\Delta W(t)$ are very small. They are multiplied by learning rate $\lambda$ which is $< 1$, so $\lambda \Delta W(t)$ can go into subnormal float16 range.

Later gradients becomes larger, but $\lambda$ becomes smaller, so $\lambda \Delta W(t)$ becomes even smaller.
ALEXNET: FP16 MASTER WEIGHT STORAGE

There are a number of solutions for this “vanishing update” problem.

**Option 1 “ FP32 master copy of weights”**

- float $\overline{W}_{32}$ “master” copy for updates,
- fp16 $\overline{W}_{16}$ weights used for forward-backward pass:

  Compute $\Delta W_{16}(t)$ using forward-backward pass

  Convert gradients to float: $\Delta W_{32}(t) = \text{half2float}(\Delta W_{16}(t))$

  Update weights in float: $W_{32}(t+1) = W_{32}(t) - \lambda \Delta W_{32}(t)$

  Make float16 copy of weights: $W_{16}(t+1) = \text{float2half}(W_{32}(t+1))$

  Do forward-backward with $W_{16}$ ...

So $\overline{W}_{32}$ will accumulate small weights updates.
ALEXNET: FP16 MASTER WEIGHT STORAGE

Consider SGD with momentum:

1. Compute momentum $H$:
   \[ H(t+1) = m*H(t) - \lambda*\Delta W(t) \]

2. Update weights with $H$:
   \[ W(t+1) = W(t) + H(t+1) \]

$\lambda$ is small, so $\lambda*\Delta W(t)$ can be very small and it can vanish if we compute momentum in float16. Can we fix this?

Denote $D(t) = \Delta W(t)$. Assume for simplicity that $\lambda = \text{const}$. Then

\[ H(t+1) = m*H(t) - \lambda*D(t) = m*(H(t-1) - \lambda*D(t-1)) - \lambda*D(t) = -\lambda*[D(t) + m*D(t-1) + m^2*D(t-2) + m^k*D(t-k) + ...] \]

Moment works as average of gradients!
ALEXNET: FP16 MASTER WEIGHT STORAGE

Let’s modify the original momentum schema:

1. Compute momentum $H$: $H(t+1) = m \times H(t) - \lambda \times \Delta W(t)$
2. Update weights with $H$: $W(t+1) = W(t) + H(t+1)$
ALEXNET: FP16 MASTER WEIGHT STORAGE

Let’s modify the original momentum schema:

1. Compute momentum G:
   \[ G(t+1) = m \cdot G(t) + \lambda \cdot \Delta W(t) \]

2. Update weights with G:
   \[ W(t+1) = W(t) - \lambda \cdot G(t+1) \]

Now \( G \) will accumulate average of \( \Delta W(t) \) which don’t vanish!

Weights update in float16:

Compute \( \Delta w_{16}(t) \) using forward-backward pass

Compute momentum:
\[ G_{16}(t+1) = m \cdot G_{16}(t) + \Delta w_{16}(t) \]

Update in float math:
\[ W=\text{half2float}(W_{16}(t))-\lambda\cdot\text{half2float}(G_{16}(t+1)) \]

Convert result to float16:
\[ W_{16}(t+1)=\text{float2half}(W) \]

Do forward-backward with \( W_{16} \) …
ALEXNET: FP16 MASTER WEIGHT STORAGE

With this fix we can have only one copy of weights in float16:
## ALEXNET: COMPARISON OF RESULTS

<table>
<thead>
<tr>
<th>Mode</th>
<th>Top1, %</th>
<th>Top5, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fp32</td>
<td>58.62</td>
<td>81.25</td>
</tr>
<tr>
<td>FP16 training, no scaling</td>
<td>56.70</td>
<td>78.12</td>
</tr>
<tr>
<td>FP16 training, scale = 1000</td>
<td>58.9</td>
<td>81.10</td>
</tr>
<tr>
<td>Mixed precision training, scale =1000</td>
<td>59.1</td>
<td>81.2</td>
</tr>
<tr>
<td>Mixed precision training, scale =1000</td>
<td>58.50</td>
<td>81.2</td>
</tr>
</tbody>
</table>
## INCEPTION-V3 RESULTS

Scale loss function by 100x...

<table>
<thead>
<tr>
<th>Mode</th>
<th>Top1, %</th>
<th>Top5, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fp32</td>
<td>73.85</td>
<td>91.44</td>
</tr>
<tr>
<td>Mixed precision training</td>
<td>73.6</td>
<td>91.11</td>
</tr>
<tr>
<td>FP16 training</td>
<td>71.36</td>
<td>90.84</td>
</tr>
<tr>
<td>FP16 training, loss scale = 100</td>
<td>74.13</td>
<td>91.51</td>
</tr>
<tr>
<td>FP16 training, loss scale = 100, FP16 master weights</td>
<td>73.52</td>
<td>91.08</td>
</tr>
</tbody>
</table>
INCEPTION-V3 RESULTS

![Graph showing top 1 accuracy over iteration for different models: Inception_v3_baseline, Inception_v3_dfp16, Inception_v3_mfp16_x100, Inception_v3_nfp16_x100.](image)
**RESNET-50 RESULTS**

No scale

<table>
<thead>
<tr>
<th>Mode</th>
<th>Top1, %</th>
<th>Top5, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fp32</td>
<td>73.2</td>
<td>91.2</td>
</tr>
<tr>
<td>FP16 training, FP32 master weights</td>
<td>73.2</td>
<td>90.9</td>
</tr>
<tr>
<td>FP16 training, FP16 master weight</td>
<td>72.7</td>
<td>91.5</td>
</tr>
<tr>
<td>Mixed, FP16 master weight</td>
<td>73.5</td>
<td>91.4</td>
</tr>
</tbody>
</table>

Nvcaffe-0.16, DGX-1, SGD with momentum, 100 epochs, batch=512, min augmentation, 1 crop, 1 model
RESNET-50 RESULTS

FP16 training is ok

FP16 storage has a small dip at the end (noise?)
1. **Mixed precision with FP32 master weight storage:**
   Good results on with a variety of networks
   - Applying a global scaling to the loss input is needed for some networks
   - Wide range of loss scaling values work well
   - float master copy of weights

2. **FP16 master weight storage** worked for convnets after refactoring the solver
TRAINING WITH MIXED PRECISION IN NVCAFFE-0.16
NVIDIA/CAFFE-0.16

- Full float16 support
- Mixed precision:
  - Different data types for Forward and Backward
  - Different math type
  - Solver_type (for weight update in float16)
- Automatic type conversion

https://github.com/NVIDIA/caffe/tree/caffe-0.16
name: "AlexNet_fp16"

default_forward_type: FLOAT16
default_backward_type: FLOAT16

# default_forward_math: FLOAT  # GP100 only
# default_backward_math: FLOAT  # GP100 only

global_grad_scale: 1000.

layer {
    forward_math: FLOAT16
    backward_math: FLOAT

    ...
}

solver_data_type: FLOAT16

https://github.com/NVIDIA/caffe/tree/caffe-0.16
default_forward_type:  FLOAT16
default_backward_type:  FLOAT16

template<typename Ftype, typename Btype>
class Layer : public LayerBase {…

default_forward_math:  FLOAT

forward_math_ = this->layer_param().forward_math();
...
setConvolutionDesc(forward_math_, fwd_conv_descs_[i],
    pad_h, pad_w, stride_h, stride_w);
enum Type {
    DOUBLE = 0,
    FLOAT = 1,
    FLOAT16 = 2,
...

class Blob {
    mutable shared_ptr<Tensor> data_tensor_;  
mutable shared_ptr<Tensor> diff_tensor_;  
...  

class Tensor {
    Type type_;  
    shared_ptr<vector<shared_ptr<SyncedMemory>>> synced_arrays_;  
...  

template< typename Dtype >
class TBlob : public Blob {
...
template<typename Gtype, typename Wtype>
__global__ void SGDRegUpdateAllAndClear(int N, Gtype* g, Wtype* w, Wtype* h,
float momentum, float local_rate, float local_decay, bool reg_L2, bool clear_grads) {
    CUDA_KERNEL_LOOP(i, N) {
        Wtype reg = reg_L2 ? w[i] : Wtype((Wtype(0) < w[i]) - (w[i] < Wtype(0)));
        Wtype gr = Wtype(g[i]) + reg * local_decay;
        gr = h[i] = momentum * h[i] + local_rate * gr;
        w[i] -= gr;
        g[i] = clear_grads ? Gtype(0) : Gtype(gr);
    }
}

template<> __global__ void SGDRegUpdateAllAndClear<__half, float>(int N,__half* g,float* w,
float* h, float momentum, float l_rate, float l_decay, bool reg_L2, bool clear_grads) {
    __half hz; hz.x = 0;
    CUDA_KERNEL_LOOP(i, N) {
        float reg = reg_L2 ? w[i] : (0.F < w[i]) - (w[i] < 0.F);
        float gr = __half2float(g[i]) + reg * l_decay;
        gr = h[i] = momentum * h[i] + l_rate * gr;
        w[i] -= gr;
        g[i] = clear_grads ? hz : float2half_clip(h[i]);
    }
}