AI85 Discussion List

6/10/2019

Top Goal: Increase capacity

Since we’re weight constrained, this primarily means optimizing weights in any way possible

# 1. Increase weight #

* Increase pure capacity
* Also, allow 4-bit weights (plus 1, 2 if feasible)

# 2. Per-layer scale factor

In addition to a scale factor or shift value that is applied at the output (before reduction to 8-bit space), this could be extended to support Batch Normalization (integer scale factor, shift + addition)

# 3. Increase avg pooling accumulator

Currently, 2048 is the maximum (11 bits, supports roughly 3x3 or 4x4 after activation).   
I’d like to allow 32x32 (= 18 bits) if possible

# 4. Empty filter removal (per-filter)?

This could save some space but would require bit(s) in *each* filter prior or next filter indicating that one or more filter(s) should be skipped. Probably too much trouble?

# 5. Fully connected layer (in digital)

A fairly low priority in my opinion, but many networks use it as the classification layer. If we do it, it doesn’t have to be very big (the number of weights just blows up).

*Update:* While it’s OK to quantize weight and bias to 8-bit, I see a large performance drop when restricting the output to 8-bit (e.g., from 84% down to 77% on CIFAR-10 – and that only with a custom shift factor, otherwise it’s only 39%). Since this is the last output before the SoftMax decision, this last layer does not perform a ReLU. In this case (no activation, last layer), it would be very helpful to output 16 bits instead of 8.

# 6. 9-bit data mode

Since the output of the ReLU is always positive, we could add two configuration bits to each layer as follows:

* Output format bit: Tells the ReLU to output 8-bit unsigned (implied 9th bit as zero).
* Input format bit: Tells the processor that there is an extra implied 9th bit in the input which is zero (this could tie in with the per-layer scale factor/shift).

*Update:* Some experiments with the CIFAR-10 code on AI84 show that this does not significantly improve anything. However, it may be helpful for elementwise addition (see below).

# 7. New operators *(Updated)*

## Fire

Add “Skip” (used in Fire, see Tiny SSD or Fire).

The simple "Fire" operator should already be possible using current hardware.  
 x = nn.ReLU(self.squeeze(x))  
 x = torch.cat([nn.ReLU(self.expand1x1(x)), nn.ReLU(self.expand3x3(x))], 1)  
where  
 squeeze and expand1x1 have a 1x1 kernels:  
 self.squeeze = nn.Conv2d(inplanes, squeeze\_planes, kernel\_size=1)  
 self.expand1x1 = nn.Conv2d(squeeze\_planes, expand1x1\_planes, kernel\_size=1)  
 expand3x3 has a 3x3 kernel:  
 self.expand3x3 = nn.Conv2d(squeeze\_planes, expand3x3\_planes, kernel\_size=3,  
 padding=1)

By using the same input processors for both expand operators and running them in separate layers serially, the output of those layers can then be arranged to be "concatenated" in memory (i.e., contiguous).

## Element-wise addition

ResNet requires new hardware: <https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035>

ResNet and similar networks require a “combination” of two different prior layer outputs. Typically, this is an element-wise addition, denoted as a (+) in the pictures. /var/folders/lc/4cgck9kd3p93xvkmr4f1w9s40000gp/T/com.microsoft.Word/WebArchiveCopyPasteTempFiles/cid05465d42-0cc5-4712-a6b8-4fad009a3f94

There are some papers about this, for example <https://arxiv.org/pdf/1603.05027.pdf>

1. We don’t have an element wise addition, and we don’t have anything yet that pulls data from two different sources.
2. When we add in 8-bit space, we need to worry about overflow. Perhaps this is where implementing the “9-bit data” option could help.
3. Batch Normalization seems to be used as well, and we don’t have that either.

The element-wise addition could have to have an optional ReLU, but we might be able to live without (quote: “This is just an empirical result. I mean they try to justify their actions with some hand wavy stuff, but it’s not a sound theory yet. There are not many theoretical works on skip connections. – Thomas Pinetz”).

## 1x1 Convolution

Add 1x1 Convolution (for Fire, as used in many new nets such as Tiny SSD).

## Pooling without convolution

Allow pooling without convolution (important for last layer).

## 1D convolution

Add 1D convolution (for sensor data)?

## Dilation support

Add dilation support (2 — would allow the 3x3 kernel to span a 5x5 field)?

## Padding for pooling

Add optional padding to pooling? Low priority.

## Simpler concatenation?

Concatenating two outputs is currently possible if the channels are a multiple of four (due to the HWC/ “little data” mode). It’s worth discussion ways to allow this for other numbers of channels.

# 8. Fix bias

The programmed bias value needs to be shifted left 7 in the addition to be mathematically correct AND useful at the same time. It's currently mathematically correct, but the bias needs to be quantized with the same scale as the weight. The bias is applied either pre-shift or post-shift and so the weight scale is different from the bias scale, rendering it useless (using the bias could increase network Top1 by 0.3% for FashionMNIST).

# 9. Various

## Modify average pooling to round instead of truncate

*Update:* Some experiments with the CIFAR-10 code on AI84 show that this does not significantly improve anything. It should probably still be done if it’s easy enough to fix.

## Improve average pooling to use higher internal resolution

This doesn’t sound feasible?

## Fixes & verify existing operators

* Allow weight read back for channel >= 4.
* Check stride != 1 (this probably works).
* Check 4x4 pooling with stride of 2, 2x2 stride 1 (these probably work for even inputs), also 3x3 stride 2 (unknown).
* Check pooling that’s not a clean divider of the input dimensions (e.g., 17x17, pool\_size=2, pool\_stride=2).

# 10. Simplify accelerator unloading

Currently, unloading the accelerator and transferring the data into a standard order is a bit involved and requires a lot of byte writes. A hardware helper for this would be helpful.

*Example code:*

uint32\_t val, \*addr, offs;

addr = (uint32\_t \*) 0x50110000;

val = \*addr++;

offs = 0x0000;

out\_buf[offs] = val & 0xff;

out\_buf[offs+0x10] = (val >> 8) & 0xff;

out\_buf[offs+0x20] = (val >> 16) & 0xff;

out\_buf[offs+0x30] = (val >> 24) & 0xff;

val = \*addr++;

offs++;

out\_buf[offs] = val & 0xff;

out\_buf[offs+0x10] = (val >> 8) & 0xff;

out\_buf[offs+0x20] = (val >> 16) & 0xff;

out\_buf[offs+0x30] = (val >> 24) & 0xff;

val = \*addr++;

offs++;

out\_buf[offs] = val & 0xff;

out\_buf[offs+0x10] = (val >> 8) & 0xff;

out\_buf[offs+0x20] = (val >> 16) & 0xff;

out\_buf[offs+0x30] = (val >> 24) & 0xff;

val = \*addr++;

offs++;

etc. etc. etc. etc. etc. etc., but offs needs reset when reaching the end of a 4-channel bundle, and addr needs reset when crossing instances.