Final Project
Efficient Deconvolution with HLS

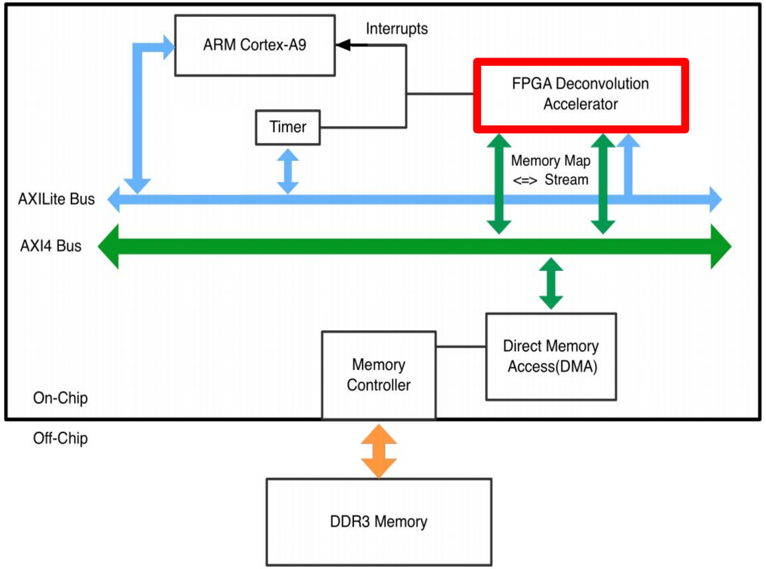
Team 6
廖晨皓
許國讚

Outline

- Target
- Method
- Quantization and RMMD Test
- Roofline model
- Coding patterns
- Reference

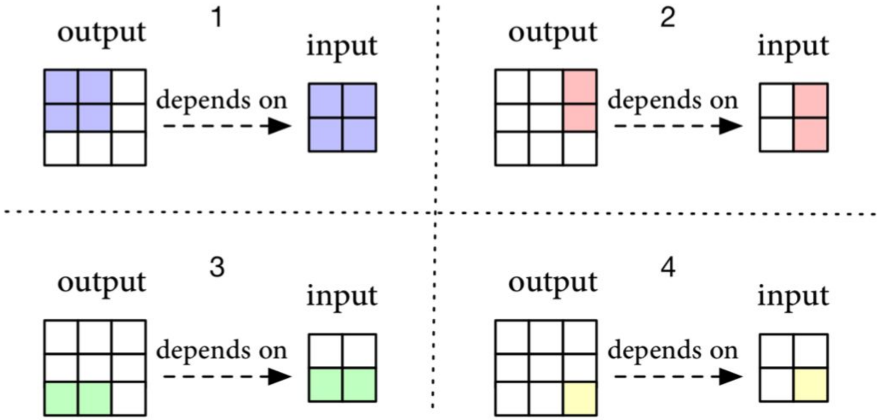
Target

- Design a deconvolution accelerator on FPGA.
- To speed up neural networks which use deconvlolution operation.



Method

- Reverse deconvolution proposed in A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA.
- Use output space to determine which input blocks to deconvolve to avoid overlapping sum problem in deconvolution.



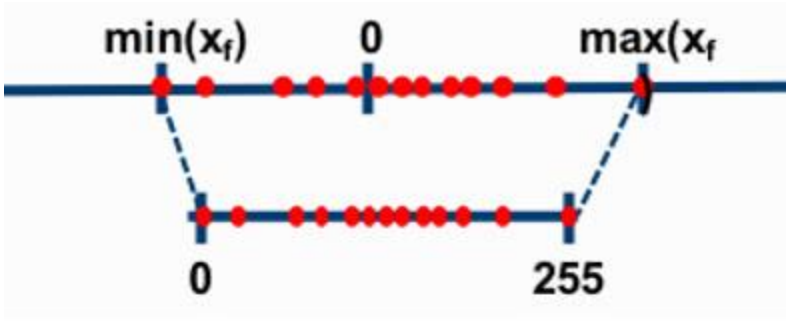
Quantization

- Mapping the neural network's weights and inputs from float to ap_int.
- We use post training quantization method to quantize the pretrained DCGAN model.

$$scale = \frac{2^n - 1}{\max(x_f) - \min(x_f)}$$

$$offset = scale \times \min(x_f) + 2^{n-1}$$

$$x_q = scale \times x_f + offset$$



https://intellabs.github.io/distiller/algo_quantization.html

Quantization (cont.)

- Generated image with different quantization bits.

float32

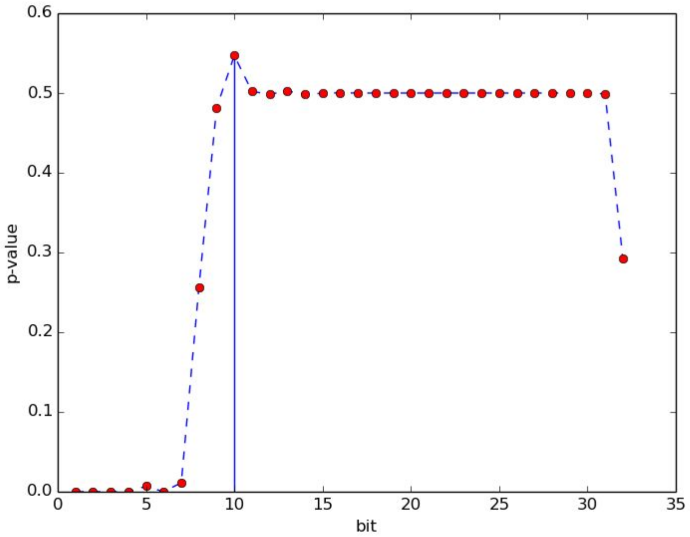
1 bit

8 bit

32 bit

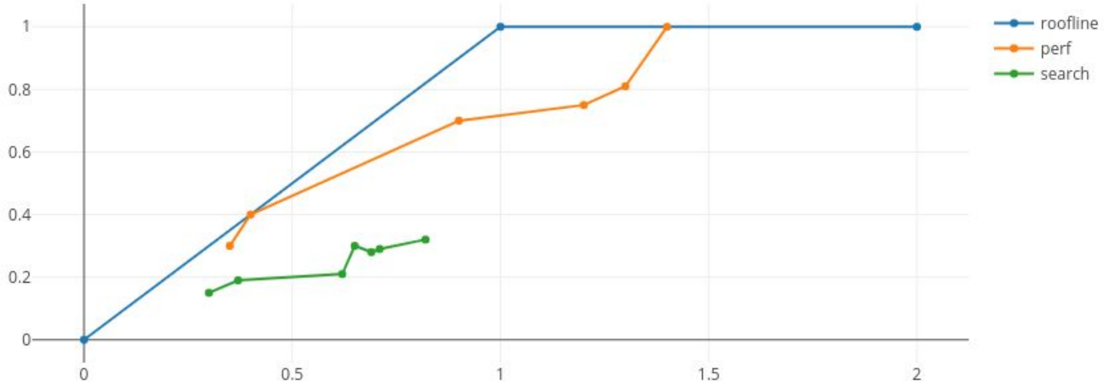
RMMD Test

- Use MMD to estimate the discrepency between training data and and the generated image (from low-bit DCGAN and full precision DCGAN).
- Then use RMMD tests the null hypothesis to determine the quantization bits.



Roofline Model

- Perf:
    - Populate the board with large amount of mundane arithmetic logics that crunches numbers.
    - Keep streaming data to the board and record the computation and memory performance.
- Roofline:
    - The circumscribing roofline of the “perf” curve.
- Parameter search:
    - Alter parameters (bitwidth, precision) that affect template type argument, e.g., ap_fixed<>.
    - Generate new bitstream.
    - Reboot and flash the board.
    - Reiterate.
    - Performance is not actually due to the parameters we originally expected.



Coding Patterns

- In order to test different parameters, we use template to write our deconvolution operation.

```cpp
template <int Stride, int Padding,
          class Tp, // arithmetic type
          int OutHeight, int OutWidth, int OutChannels,
          int InHeight, int InWidth, int InChannels,
          int KerSize>
void deconvolution(Tp (&out)[OutHeight][OutWidth][OutChannels],
                   const Tp (&in)[InHeight][InWidth][InChannels],
                   const Tp (&ker)[KerSize][KerSize][OutChannels][InChannels])
```

Coding Patterns (cont.)

- The templated function must be instantiated by the top level function above in order to be synthesized.
- Use pragma “array_reshape” to access more data in a cycle .

```
void deconv(ap_fixed<12, 6> (&out)[128][128][3],
            const ap_fixed<12, 6> (&in)[64][64][3],
            const ap_fixed<12, 6> (&ker)[12][12][3][3])
{
    #pragma HLS array_reshape variable=out dim=3
    #pragma HLS array_reshape variable=in dim=3
    #pragma HLS array_reshape variable=ker dim=4

    deconvolution<6, 16>(out, in, ker);
}
```

Coding Patterns (cont.)

- We improved upon the original code of the paper, which was overly complex.

- We deduced a way to compute all loop bounds at compile time. This potentially allows each loop to be pipelined, unrolled, or flattened.

- We tried several optimizations, they are either too aggressive (resulting in unstable synthesis), or doesn't help much. This is the one that worked best. Note that the order of the loops are important for performance.

```
#pragma HLS INLINE
    constexpr auto I = ratio_ceiling(Padding - KerSize + 1, Stride);
    constexpr auto IH = min(InHeight, (OutHeight + 2 * Padding - KerSize) / Stride + 2);
    constexpr auto IW = min(InWidth, (OutWidth + 2 * Padding - KerSize) / Stride + 2);

loop_ih:
    for (auto ih = I; ih < IH; ++ih) {
    loop_iw:
        for (auto iw = I; iw < IW; ++iw) {
        loop_kh:
#pragma HLS PIPELINE
            for (auto kh = 0; kh < KerSize; ++kh) {
                const auto oh = Stride * ih + kh - Padding;
                if (oh < 0 || oh >= OutHeight) {
                    continue;
                }
            loop_kw:
                for (auto kw = 0; kw < KerSize; ++kw) {
                    const auto ow = Stride * iw + kw - Padding;
                    if (ow < 0 || ow >= OutWidth) {
                        continue;
                    }
                loop_oc:
                    for (auto oc = 0; oc < OutChannels; ++oc) {
#pragma HLS UNROLL
                    loop_ic:
                        for (auto ic = 0; ic < InChannels; ++ic) {
#pragma HLS UNROLL
// #pragma HLS loop_flatten
                            out[oh][ow][oc] += in[ih][iw][ic] * ker[kh][kw][oc][ic];
```

Coding Patterns (cont.)

Template recursion that accepts arrays of arbitrary dimension.

Normally, we use a special template argument to record recursion depth.

Instead, we can use template type parameters as a record of recursion depth.

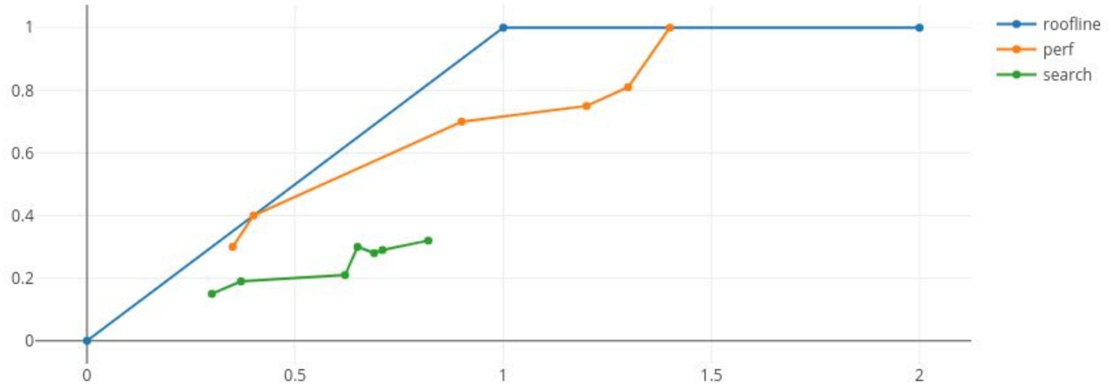
Example:
(ap_int<10>[128][128][3], float[128][128][3])
↓
(ap_int<10>[128][3], float[128][3])
↓
(ap_int<10>[3], float[3])

```
template <class Tp, int Bits, int N>
void quantize(ap_int<Bits> (&out)[N], Tp (&in)[N], float scale, float offset)
{
#pragma HLS INLINE
    for (int i{}; i != N; ++i) {
#pragma HLS loop_flatten
        out[i] = ap_int<Bits>(hls::round(scale * in[i] - offset));
    }
}

template <class ToSubarray, class FromSubarray, int N, int M>
void quantize(ToSubarray (&to_subarray)[N][M],
                     FromSubarray (&from_subarray)[N][M],
                     float scale, float offset)
{
#pragma HLS INLINE
    for (int i{}; i != N; ++i) {
        quantize(to_subarray[i], from_subarray[i], scale, offset);
    }
}

void deconv(ap_int<10> (&quant_input)[128][128][3],
            const float (&input)[128][128][3]) {
    quantize(quant_input, input, 2, -1.3);
}
```

This actually synthesizes correctly, and is equivalent to 3 layers of nested loops.

Result

- Random deconvolution of mnist with noise.



Reference

- A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA, arXiv 2017
- A test of relative similarity for model selection in generative models, arXiv 2015
- GitHub
    a. joeylitalien/celeba-gan-pytorch
    b. chl218/DCNN-on-FPGA

GitHub

https://github.com/learningstud/hls_deconvolution

Auto deduce array dimension size at compile time, and the use of “array_reshape”.

```
void deconv(ap_fixed<12, 6> (&out)[128][128][3],
            const ap_fixed<12, 6> (&in)[64][64][3],
            const ap_fixed<12, 6> (&ker)[12][12][3][3])
{
    #pragma HLS array_reshape variable=out dim=3
    #pragma HLS array_reshape variable=in dim=3
    #pragma HLS array_reshape variable=ker dim=4

    deconvolution<6, 16>(out, in, ker);
}
```

The templated function below must be instantiated by the top level function above in order to be synthesized.

```
template <int Stride, int Padding,
          class Tp, // arithmetic type
          int OutHeight, int OutWidth, int OutChannels,
          int InHeight, int InWidth, int InChannels,
          int KerSize>
void deconvolution(Tp (&out)[OutHeight][OutWidth][OutChannels],
                   const Tp (&in)[InHeight][InWidth][InChannels],
                   const Tp (&ker)[KerSize][KerSize][OutChannels][InChannels])
```

random deconvolution of mnist with noise

```cpp
#pragma HLS INLINE
    constexpr auto I = ratio_ceiling(Padding - KerSize + 1, Stride);
    constexpr auto IH = min(InHeight, (OutHeight + 2 * Padding - KerSize) / Stride + 2);
    constexpr auto IW = min(InWidth, (OutWidth + 2 * Padding - KerSize) / Stride + 2);

loop_ih:
    for (auto ih = I; ih < IH; ++ih) {
    loop_iw:
        for (auto iw = I; iw < IW; ++iw) {
        loop_kh:
#pragma HLS PIPELINE
            for (auto kh = 0; kh < KerSize; ++kh) {
                const auto oh = Stride * ih + kh - Padding;
                if (oh < 0 || oh >= OutHeight) {
                    continue;
                }
            loop_kw:
                for (auto kw = 0; kw < KerSize; ++kw) {
                    const auto ow = Stride * iw + kw - Padding;
                    if (ow < 0 || ow >= OutWidth) {
                        continue;
                    }
                loop_oc:
                    for (auto oc = 0; oc < OutChannels; ++oc) {
#pragma HLS UNROLL
                    loop_ic:
                        for (auto ic = 0; ic < InChannels; ++ic) {
#pragma HLS UNROLL
// #pragma HLS loop_flatten
                            out[oh][ow][oc] += in[ih][iw][ic] * ker[kh][kw][oc][ic];
```

We improved upon the original code of the paper, which was overly complex, and not necessarily faster.

We deduced a way to compute all loop bounds at compile time. This potentially allows each loop to be pipelined, unrolled, or flattened.

We tried several optimizations, they are either too aggressive (resulting in unstable synthesis), or doesn't help much. This is the one that worked best. Note that the order of the loops are important for performance.

```cpp
template <class Tp, int Bits, int N>
void quantize(ap_int<Bits> (&out)[N], Tp (&in)[N], float scale, float offset)
{
#pragma HLS INLINE
    for (int i{}; i != N; ++i) {
#pragma HLS loop_flatten
        out[i] = ap_int<Bits>(hls::round(scale * in[i] - offset));
    }
}

template <class ToSubarray, class FromSubarray, int N, int M>
void quantize(ToSubarray (&to_subarray)[N][M],
              FromSubarray (&from_subarray)[N][M],
              float scale, float offset)
{
#pragma HLS INLINE
    for (int i{}; i != N; ++i) {
        quantize(to_subarray[i], from_subarray[i], scale, offset);
    }
}

void deconv(ap_int<10> (&quant_input)[128][128][3],
            const float (&input)[128][128][3]) {
    quantize(quant_input, input, 2, -1.3);
}
```

This actually synthesizes correctly, and is equivalent to 3 layers of nested loops.

Template recursion that accepts arrays of arbitrary dimension.

Normally, we use a special template argument to record recursion depth.

Instead, we can use template type parameters as a record of recursion depth.

(ToSubarray, FromSubarray)

(ap_int<10>[128][128][3], float[128][128][3])

(ap_int<10>[128][3], float[128][3])

(ap_int<10>[3], float[3])