Unit 19

Hardware Acceleration

Motivation

- When designing hardware we have nearly unlimited control and parallelism at our disposal
- We can create structures that may dramatically improve performance of an algorithm that would otherwise be executed with software
- Let's look at some common tasks that might be accelerated by hardware implementation

Example

- Take USC fight song and remove ________________ audio from the song (i.e. reduce the “treble”)

Plot of fight song
Low Pass Filter

- We can view the song as samples over time or by taking the Fourier transform, we can see the component frequencies (i.e., the frequency domain representation).
- We would like to remove the high frequency components.

Designing a Low Pass Filter

- Below is a zoomed view.
- Removing high frequency components (parts of the signal that change rapidly) means _______ the signal or finding its basic curve and not the bumpiness (rapid changes).
- One solution: For each input sample, output the _______ of that input sample and its __________ inputs.

Moving Window Filter

- By making each sample equal to the _______ of itself plus neighboring samples we tend to smooth the signal.

Filtered Signal

- Averaging smoothes the waveform and effectively filters out high-frequency components.
### 8-tap Moving Average Filter

- Assume each sample is the average of 8 surrounding samples, we can describe the output as:

\[ y[i] = \frac{1}{8} x[i] + \frac{1}{8} x[i+1] + \ldots + \frac{1}{8} x[i-7] \]

- Example:
  - \[ y[7] = \frac{1}{8} x[7] + \frac{1}{8} x[6] + \ldots + \frac{1}{8} x[0] \]
  - \[ y[8] = \frac{1}{8} x[8] + \frac{1}{8} x[7] + \ldots + \frac{1}{8} x[1] \]

- If we want a weighted average rather than pure average we can generalize from 1/8 to some weight coefficient: \( w_k \)

\[ y[i] = \sum_{k=0}^{7} w_k x[i-k] \]

Weights: \[ w[k] \]

### Software Implementation

- Implementing this filter in software would require:
  - An outer loop to enumerate _______
  - An inner loop to multiple each ________ times its corresponding weight in the sliding ______

- Runtime is \( O(________) \)

```c
const int N = 100000; int x[N]; // input array int y[N] = {0}; // output array /* read input array */ const int WSIZE = 8; int w[WSIZE] = { /* init. array */ }; for(int i=WSIZE; i < N; i++){
    y[i] = 0;
    for(int k=0; k < WSIZE; k++){
        y[i] += x[i-k]*w[k];
    }
}
```

### Digital Implementation

- The system we want to design gets one sample per clock and produces one output sample per clock

![](moving_average_filter.png)

### Storing Last 8 Samples

- Since we only get one sample a clock, but need to use the last 8 samples to do our average, we need to save the last 8 samples
  - To store values we use registers
  - Chain together several registers
  - \( x[d1] = x[i] \) delayed by 1 clock
  - \( x[d2] = x[i] \) delayed by 2 clocks

![](store_last_8_samples.png)
19.13 Time Space Diagram

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X(0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X(1)</td>
<td>X(0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>X(2)</td>
<td>X(1)</td>
<td>X(0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>X(3)</td>
<td>X(2)</td>
<td>X(1)</td>
<td>X(0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>X(4)</td>
<td>X(3)</td>
<td>X(2)</td>
<td>X(1)</td>
<td>X(0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>X(5)</td>
<td>X(4)</td>
<td>X(3)</td>
<td>X(2)</td>
<td>X(1)</td>
<td>X(0)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>X(6)</td>
<td>X(5)</td>
<td>X(4)</td>
<td>X(3)</td>
<td>X(2)</td>
<td>X(1)</td>
<td>X(0)</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>X(7)</td>
<td>X(6)</td>
<td>X(5)</td>
<td>X(4)</td>
<td>X(3)</td>
<td>X(2)</td>
<td>X(1)</td>
<td>X(0)</td>
</tr>
<tr>
<td>8</td>
<td>X(8)</td>
<td>X(7)</td>
<td>X(6)</td>
<td>X(5)</td>
<td>X(4)</td>
<td>X(3)</td>
<td>X(2)</td>
<td>X(1)</td>
</tr>
<tr>
<td>9</td>
<td>X(9)</td>
<td>X(8)</td>
<td>X(7)</td>
<td>X(6)</td>
<td>X(5)</td>
<td>X(4)</td>
<td>X(3)</td>
<td>X(2)</td>
</tr>
</tbody>
</table>

Samples x[i] where i < 0 (negative indices) are equal to 0 since the register will be reset (cleared) at clock 0.

19.14 Averaging the Samples

- Multiple each sample by the appropriate weight (in this case each $w_k = 1/8$)
- Add up all values

\[ y(\ldots) = \frac{1}{8} \sum_{k} x(k) \]

19.15 Another Example: Image Compression

- Images are just 2-D arrays (matrices) of numbers
- Each number corresponds to the color or a pixel in that location
- Image store those numbers in some way
  - For an $n \times n$ image, requires at least $n^2$ operations and thus time if we process it sequentially with software
**Image Compression**

1. Break Image into small blocks of pixels

2. Store the difference of each pixel and the upper left (or some other representative pixel)

3. We can save more space by rounding numbers to a smaller set of options (i.e. only even # differences)

**Video Compression**

- Video is a sequence of still frames
  - 24-30 frames per second (fps)
- How much difference is expected between frames?
- Idea:
  - Store 1 of every N frames (aka key frame or I-frame), with other N-1 frames being differences from previous or next frame

**JPEG Conversion Process**

1. Break into 8x8 Tiles
2. Perform Discrete Cosine Transform on each 8x8 Block
3. Quantize
4. Huffman Coding
5. Storage (as .jpeg)
Huffman Code

- Compression algorithm
- Variable-length code
  - Each character can be coded with a different number of bits
- Prefix code
  - No two codes start with the same prefix
- Assignment of codes to characters is based on frequency of the code in the message

Encryption

- To encrypt data over a (communications) channel we need to perform some transformation of the data before transmission and the inverse at the receiver

A Simple System: LFSR

- Idea: Generate a sequence of "reproducible" random (i.e. pseudo-random) numbers and use them to modify/transform each data byte as it is sent
- Reproduce the same random sequence at the receiver and perform the inverse transformation
A Simple System: LFSR

- Linear Feedback Shift Registers can help generate the pseudo-random sequence of numbers
  - Start with a secret "key" and continuously modify it by shifting bits in one direction and putting in a "random" bit on the other side
  - This forms a sequence of random numbers

Transforming the Data

- XOR each random number from the LFSR with the next data byte (referred to as in_byte_q below) and send that value (out_byte) over the channel

Recovering the Data

- If the receiver starts with the same secret key and which "tap" bit will be chosen, it can reproduce the same pseudo-random sequence as the transmitter
- We rely on the fact that:
  - \( A \oplus B \oplus A = \) ___________________________
  - \((\text{random}_{\text{trans}} \oplus \text{data} \oplus \text{random}_{\text{recv}}) = \) _______
- The receiver just XORs the received data byte with the random number it generates (which was the same one as the transmitter) and it will have the original data

Encryption Example

<table>
<thead>
<tr>
<th>Original Data</th>
<th>Transmitter LFSR Value</th>
<th>Encrypted Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'=0x61=01100001</td>
<td>[1010100001]</td>
<td>01010100=0x54</td>
</tr>
<tr>
<td>'b'=0x62=01100010</td>
<td>[0101010010]</td>
<td>00010000=0x08</td>
</tr>
<tr>
<td>'c'=0x63=01100011</td>
<td>[1010100010]</td>
<td>10110111=0xb7</td>
</tr>
<tr>
<td>'d'=0x64=01100100</td>
<td>[1101011010]</td>
<td>11001101=0xcd</td>
</tr>
</tbody>
</table>
Decryption Example

<table>
<thead>
<tr>
<th>Encrypted Data</th>
<th>Receiver LFSR Value</th>
<th>Original Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010100=0x54</td>
<td>1 0 0 1 1 0 1 0</td>
<td>'a'=0x61=01100001</td>
</tr>
<tr>
<td>00010000=0x08</td>
<td>0 0 1 1 0 1 0 1</td>
<td>'b'=0x62=01100010</td>
</tr>
<tr>
<td>10110111=0xb7</td>
<td>1 0 1 1 0 1 0 1</td>
<td>'c'=0x63=01100011</td>
</tr>
<tr>
<td>11001101=0xcd</td>
<td>1 1 0 1 1 0 1 0</td>
<td>'d'=0x64=01100100</td>
</tr>
</tbody>
</table>

Hardware vs. Software

- To do this in software on a buffer of n bytes would require us to use instructions that sequentially performed:
  - Repeat n times:
    - Get the data from memory
    - XOR the data byte with the LFSR value
    - Shift the LFSR key left 1 spot
    - XOR the tap bit to find the new bit & add it to arrive at the next pseudo-random number
- In hardware we could perform the entire loop body in a single clock cycle and thus encrypt our data in n clocks

Structure of an LFSR Engine

Make a 4-bit Shift Register

<table>
<thead>
<tr>
<th>CLK</th>
<th>RESET</th>
<th>LOAD</th>
<th>SHIFT</th>
<th>Q3<em>Q2</em>Q1<em>Q0</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Q3,Q2,Q1,Q0</td>
</tr>
<tr>
<td>PosEdge</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0000</td>
</tr>
<tr>
<td>PosEdge</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D3,D2,D1,D0</td>
</tr>
<tr>
<td>PosEdge</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Q2,Q1,Q0,D_IN</td>
</tr>
<tr>
<td>PosEdge</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Q3,Q2,Q1,Q0</td>
</tr>
</tbody>
</table>

\[ \text{DIN} \rightarrow \text{DFF} \rightarrow \text{CLK} \rightarrow \text{RESET} \rightarrow \text{LOAD} \rightarrow \text{SHIFT} \rightarrow \text{Q}\text{n} \]