Unit 21

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
Finding and exploiting patterns in raw data

HARDWARE ACCELERATION
Example

- Take USC fight song and remove high frequency 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

![Fourier Series of Sound](image1)

![Fourier Series of Filtered Sound](image2)

Before Filter

After Filter
Designing a Low Pass Filter

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

• By making each sample equal to the average of itself plus neighboring samples we tend to smooth the signal

Weights: 1/3 each

Original signal, \( x[i] \)

Filtered Signal, \( y[i] \)
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:

\[ y[i] = \sum_{k=0}^{7} \frac{1}{8} \times x[i - k] \]

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

• If we want a weighted average rather than pure average we can generalize from \( \frac{1}{8} \) to some weight coefficient: \( w_k \)

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

Weights:

\[ w[k] \]
Software Implementation

• Implementing this filter in software would require:
  – An outer loop to enumerate each output sample
  – An inner loop to multiple each input sample times its corresponding weight in the sliding window

• Runtime is $O(N \times WSIZE)$

```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
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.
### Time Space Diagram

<table>
<thead>
<tr>
<th>Clock</th>
<th>X[i]</th>
<th>Xd1</th>
<th>Xd2</th>
<th>Xd3</th>
<th>Xd4</th>
<th>Xd5</th>
<th>Xd6</th>
<th>Xd7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X(0)</td>
<td>X(-1)=0</td>
<td>X(-2)=0</td>
<td>X(-3)=0</td>
<td>X(-4)=0</td>
<td>X(-5)=0</td>
<td>X(-6)=0</td>
<td>X(-7)=0</td>
</tr>
<tr>
<td>1</td>
<td>X(1)</td>
<td>X(0)</td>
<td>X(-1)=0</td>
<td>X(-2)=0</td>
<td>X(-3)=0</td>
<td>X(-4)=0</td>
<td>X(-5)=0</td>
<td>X(-6)=0</td>
</tr>
<tr>
<td>2</td>
<td>X(2)</td>
<td>X(1)</td>
<td>X(0)</td>
<td>X(-1)=0</td>
<td>X(-2)=0</td>
<td>X(-3)=0</td>
<td>X(-4)=0</td>
<td>X(-5)=0</td>
</tr>
<tr>
<td>3</td>
<td>X(3)</td>
<td>X(2)</td>
<td>X(1)</td>
<td>X(0)</td>
<td>X(-1)=0</td>
<td>X(-2)=0</td>
<td>X(-3)=0</td>
<td>X(-4)=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>X(-1)=0</td>
<td>X(-2)=0</td>
<td>X(-3)=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>X(-1)=0</td>
<td>X(-2)=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>X(-1)=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 there register will be reset (cleared) at clock 0.
Averaging the Samples

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

Note: EE 483 (Digital Signal Processing) will teach many ways to implement efficient filters.

$O(N+WSIZE) = O(N)$
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 nxn image, requires at least $n^2$ operations and thus time if we process it sequentially with software

Image taken from the photo "Robin Jeffers at Ton House" (1927) by Edward Weston
Image Compression
# 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)**

### Input Image

![Image](image.jpg)

### 4x4 Blocks of Pixels

<table>
<thead>
<tr>
<th>129</th>
<th>131</th>
<th>130</th>
<th>133</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
<td>130</td>
<td>131</td>
<td>129</td>
</tr>
<tr>
<td>132</td>
<td>131</td>
<td>130</td>
<td>132</td>
</tr>
<tr>
<td>134</td>
<td>132</td>
<td>131</td>
<td>132</td>
</tr>
</tbody>
</table>

### Differences

<table>
<thead>
<tr>
<th>129</th>
<th>2</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

### Rounding Differences

<table>
<thead>
<tr>
<th>129</th>
<th>2</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

### Output Image

![Output Image](output.jpg)
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
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)

Note: EE 569 (Image Processing) will cover many related concepts.
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
Linear Feedback Shift Register

ENCRYPTION EXAMPLE
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.

Original data pixels

 Pixels after encryption
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

The tap bit (in your design we will use a mux to allow any of the lower 8-bits to be selected)
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:
  - \( A \oplus B \oplus A = A \oplus A \oplus B = 0 \oplus B = B \)
  - \((\text{random}_{\text{trans}} \oplus \text{data} \oplus \text{random}_{\text{recv}}) = \text{data}\)

• 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

**Original Data**

- 'a' = 0x61 = 01100001
- 'b' = 0x62 = 01100010
- 'c' = 0x63 = 01100011
- 'd' = 0x64 = 01100100

**Transmitter LFSR Value**

- Initial value: 10011010110101
- After shift: 01101010101100
- Output: 1

**Encrypted Data**

- 'a' = 0x54 = 01010100
- 'b' = 0x08 = 00001000
- 'c' = 0xb7 = 10110111
- 'd' = 0xcd = 11001101
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>01001101010101</td>
<td>'a'=0x61=01100001</td>
</tr>
<tr>
<td>00001000=0x08</td>
<td>00110101010101</td>
<td>'b'=0x62=01100010</td>
</tr>
<tr>
<td>10110111=0xb7</td>
<td>01101010101010</td>
<td>'c'=0x63=01100011</td>
</tr>
<tr>
<td>11001101=0xcd</td>
<td>11010101001010</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

[Diagram of an LFSR Engine]

{0,KEY[7:0]}

LOAD
D[8:0]
D_IN
SHIFT
valid
RESET
Q[8:0]

Shift Reg.
clk

lfsr[8:0]

8-to-1 Mux

S[2:0]
lfsr[7]
lfsr[0]

in_byte[7:0]
in_en
valid
out_en

out_byte[7:0]
x8

REG

D[7:0] Q[7:0]
reset
clk

DFF

D
RST
CLK
reset
clk

lfsr_fsm

running

shload

start
stop

STOPPED

LOAD_KEY

STARTED

RESET

~start

start

~start~stop

~start~stop

running ← 1

~start~stop

~start

LOAD_KEY

shload ← 1
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>

CLK, RESET, LOAD, SHIFT, Q3*Q2*Q1*Q0*