Huffman Encoding and Decoding

Group Members:

Salem Ait Ami (V00932871)
salemaitami@uvic.ca

Adam Tidball (V00936605)
adamtidball@uvic.ca

PRESENTATION SECTIONS

01

BACKGROUND & SPECIFICATIONS

02

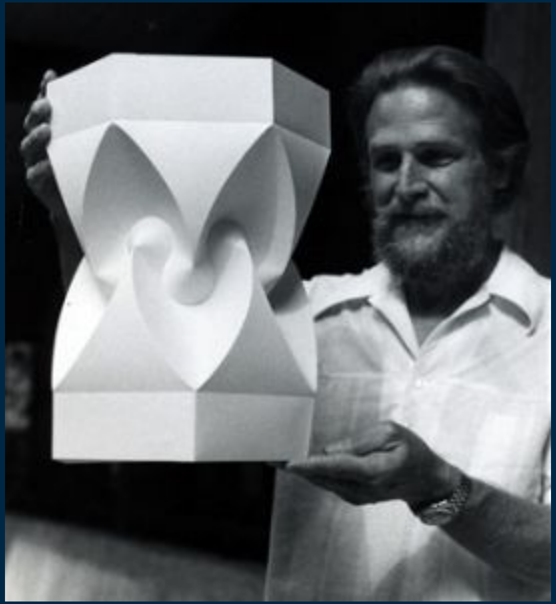
OPTIMIZATION TECHNIQUES

03

RESULTS & CONCLUSION

David Huffman & Huffman Coding

- David Huffman invited Huffman Coding in 1952 while studying at MIT
- Huffman Coding is a lossless data compression technique that is still used today (MPEG)
- Compressing data is useful for data storage and transmission purposes



David Huffman 1978 - UC Santa Cruz [1]

Huffman Encoding

- Huffman Encoding creates codes based on each of the symbols' probability of occurrence
- Common symbols have shorter code lengths

Example of a 5 Symbol Alphabet:

- Without Huffman 3 bits are needed per symbol
- With Huffman ≈ 2 bits are needed per symbol

$$(0.4*1)+(0.25*2)+(0.2*3)+(0.1*4)+(0.05*4) = 2.1 \approx 2$$

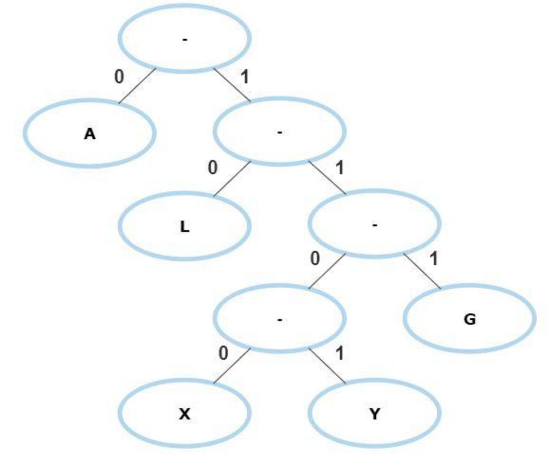
Symbol	Probability	Code Length
A	0.4	1
L	0.25	2
G	0.2	3
Y	0.1	4
X	0.05	4

Alphabet Probabilities & length

Huffman Decoding

Huffman Decoding can be done in different ways:

1. Brute force approach using bit-by-bit search with the Huffman Tree
2. Look up Table (LUT) search using a maximum code length barrel shifter



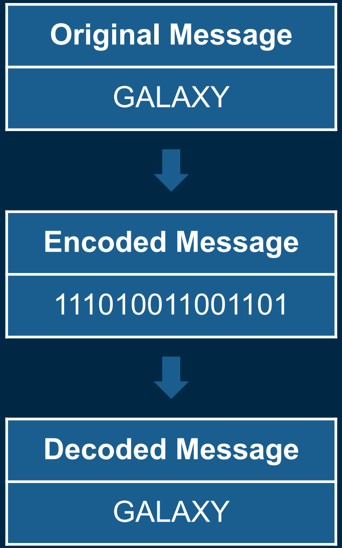
Huffman Tree

Index	Symbol	Code Length
0000	A	1
0001	A	1
0010	A	1
⋮	⋮	⋮
1111	G	3

Section of LUT

Huffman Decoding 2

- Encoded data does not contain specific guard bits separating codewords
- Decoding is a strictly sequential process
- The slow decoding process is the penalty for reducing the bit-rate of the encoded message



Project Specifications

Alphabet Size: We currently support 128 ASCII symbols and alphabets containing that number of characters or less. The code can also be extended to accommodate extended ASCII.

Input Text Size: The program accepts varying file lengths, provided they are less than 256 kB.

Probabilities: The program determines the probabilities of each symbol based on the input text prior to encoding.

Implementation: Huffman decoding is implemented using a brute force approach and LUT approach, metrics for both are compared. Metrics are acquired with gprof and cachegrind.

ARM Machine Information

- Uvic provided ARM machine: S3C2440A Microcontroller
- ARM920T core which is a 16/32 bit RISC processor
- 16KB data cache
- 16KB instruction cache
- Address space: 128Mbytes per bank (total 1GB/8 banks)

Mnemonic	Instruction	Action
ADC	Add with carry	Rd: = Rn + Op2 + Carry
ADD	Add	Rd: = Rn + Op2
AND	AND	Rd: = Rn AND Op2
B	Branch	R15: = address
BIC	Bit clear	Rd: = Rn AND NOT Op2
BL	Branch with link	R14: = R15, R15: = address
BX	Branch and exchange	R15: = Rn, T bit: = Rn[0]
CDP	Coprocessor data processing	(Coprocessor-specific)
CMN	Compare Negative	CPSR flags: = Rn + Op2
CMP	Compare	CPSR flags: = Rn - Op2
EOR	Exclusive OR	Rd: = (Rn AND NOT Op2) OR (Op2 AND NOT Rn)
LDC	Load coprocessor from memory	Coprocessor load
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	Rd: = (address)
MCR	Move CPU register to coprocessor register	cRn: = rRn {<op>cRm}
MLA	Multiply accumulate	Rd: = (Rm × Rs) + Rn
MOV	Move register or constant	Rd: = Op2

ARM Instruction Set [2]

Project Goals

Alphabet Probability Calculation

- 128 ASCII characters can be in the input alphabets including control characters

File Encoding & File Decoding

- The encoded file should be smaller than the decoded/original file

Optimizations

- Explore optimization techniques

Optimization #1 – Declaring Register Variables

Original Code

```
int i;
for(i = 0; i < (*tables_needed); i++){
    free(all_luts[i]);
}
```

Optimized Code

```
register uint32_t i;
for(i = 0; i < (*tables_needed); i++){
    free(all_luts[i]);
}
```

Assembly

```
main:
        str     fp, [sp, #-4]!
        add     fp, sp, #0
        sub     sp, sp, #12
        mov     r3, #10
        str     r3, [fp, #-12]
        mov     r3, #0
        str     r3, [fp, #-8]
        b       .L2
.L3:
        ldr     r3, [fp, #-8]
        add     r3, r3, #1
        str     r3, [fp, #-8]
.L2:
        ldr     r3, [fp, #-12]
        ldr     r2, [r3, #0]
        ldr     r3, [fp, #-8]
        cmp     r2, r3
        bgt     .L3
        mov     r0, r3
        add     sp, fp, #0
        ldmfd   sp!, {fp}
        bx      lr
```

Optimized Assembly

```
main:
        stmfd   sp!, {r4, fp}
        add     fp, sp, #4
        sub     sp, sp, #8
        mov     r3, #10
        str     r3, [fp, #-8]
        mov     r4, #0
        b       .L2
.L3:
        add     r4, r4, #1
.L2:
        ldr     r3, [fp, #-8]
        ldr     r3, [r3, #0]
        cmp     r3, r4
        bgt     .L3
        mov     r0, r3
        sub     sp, fp, #4
        ldmfd   sp!, {r4, fp}
        bx      lr
```

Optimization #2 – Bit Shifting Operations

<table>
<tr><th>Original Code</th><th>Optimized Code</th></tr>
<tr><td><pre><code>int size;
size = size / sizeof(uint16_t);</code></pre></td><td><pre><code>uint16_t size;
size >>= 1;</code></pre></td></tr>
<tr><th>Assembly</th><th>Optimized Assembly</th></tr>
<tr><td><pre><code>main:
        str   fp, [sp, #-4]!
        add   fp, sp, #0
        sub   sp, sp, #12
        mov   r3, #2
        str   r3, [fp, #-8]
        ldr   r3, [fp, #-8]
        mov   r3, r3, lsr #1
        str   r3, [fp, #-8]
        mov   r0, r3
        add   sp, fp, #0
        ldmfd   sp!, {fp}
        bx   lr</code></pre></td><td><pre><code>main:
        str   fp, [sp, #-4]!
        add   fp, sp, #0
        sub   sp, sp, #12
        mov   r3, #2
        str   r3, [fp, #-8]
        ldr   r3, [fp, #-8]
        mov   r3, r3, lsr #1
        str   r3, [fp, #-8]
        mov   r0, r3
        add   sp, fp, #0
        ldmfd   sp!, {fp}
        bx   lr</code></pre></td></tr>
</table>

Optimization #3 – Branch Reduction

Original Code

```
for (int i = 0;i < barrel shifter; i++) {
    encoded section <<= 1;
    if (encoded[i]  == 1) {
        encoded_section |= 1;
    }
    else if(encoded[i] == 0) {
        encoded_section |= 0;
    }
    else {
        printf("Tried to convert non 1 or 0 to
bit!\n");
        exit(1);
    }}
```

Optimized Code

```
register uint32 t j;
for (j = decoded bit count; j < barrel_shifter +
decoded_bit_count; j++) {

    if (compressed[cur index] & (1U << bit pos )) {
        compressed section |= 1u << ((barrel_shifter -
1) - (j - decoded_bit_count));
    }
    bit pos++;
    if(bit pos >= 16){
        bit pos = 0;
        cur_index++;}}
```

Assembly

```
        mov     r3, #0
        str     r3, [fp, #-12]
        b       .L2
.L6:
        ldr     r3, [fp, #-8]
        mov     r3, r3, asl #1
                ...
        str     r3, [fp, #-8]
        b       .L4
.L3:
        ldr     r3, [fp, #-12]
        mov     r3, r3, asl #2
                ...
        mov     r0, #1
        bl      exit
.L7:
        mov     r0, r0    @ nop
.L4:
        ldr     r3, [fp, #-12]
        add     r3, r3, #1
        str     r3, [fp, #-12]
.L2:
        ldr     r2, [fp, #-12]
        ldr     r3, [fp, #-16]
                ...
        sub     sp, fp, #4
        ldmfd   sp!, {fp, pc}
.L8:
        .word   .LC0
```

Optimized Assembly

```
        ldr     r4, [fp, #-24]
        b       .L2
.L5:
        ldr     r3, [fp, #-12]
        mov     r3, r3, asl #2
                ...
        orr     r3, r2, r3
        str     r3, [fp, #-8]
.L3:
        ldr     r3, [fp, #-16]
        add     r3, r3, #1
                ...
        add     r3, r3, #1
        str     r3, [fp, #-12]
.L4:
        add     r4, r4, #1
.L2:
        ldr     r2, [fp, #-20]
        ldr     r3, [fp, #-24]
                ...
        ldmfd   sp!, {r4, fp}
        bx      lr
```

Optimization #4 – Memory Organization

- Custom_fwrite function implemented
- As well as '#defines' and 'consts' added when applicable

```
void custom_fwrite(const void* array, uint32_t element_size, uint32_t total_elements, FILE* file) {
    uint32_t elements_written = 0;
    uint32_t chunk_size = 32000;

    while (elements_written < total_elements) {
        uint32_t remaining_elements = total_elements - elements_written;
        uint32_t elements_to_write = (remaining_elements < chunk_size) ? remaining_elements : chunk_size;
        uint32_t bytes_written = fwrite(array + (elements_written * element_size), element_size, elements_to_write, file);
        assert(bytes_written == elements_to_write);

        fflush(file);
        elements_written += elements_to_write;
    }
}
```

Custom fwrite Function From Optimized Code

Optimization #5 – Assembly inlining

Original Code

```
table_index = compressed_section % max;
table_num = compressed_section / max;
```

Optimized Code

```
__asm__ __volatile__ (
    "mov r0, %[dividend]\n\t"
    "mov r1, %[divisor]\n\t"
    "mov r2, #0\n\t"
    "udiv r3, r0, r1\n\t"
    "mov %[quotient], r3\n\t"
    "mov %[remainder], r2\n\t"
    : [quotient] "=r" (table_index), [remainder] "=r" (table_num)
    : [dividend] "r" (compressed_section), [divisor] "r" (max)
    : "r0", "r1", "r2", "r3" );
```

Assembly

```
ldr     r3, [fp, #-16]
mov     r0, r3
ldr     r1, [fp, #-20]
bl      __aeabi_idivmod
mov     r3, r1
str     r3, [fp, #-8]
ldr     r0, [fp, #-16]
ldr     r1, [fp, #-20]
bl      __aeabi_idiv
mov     r3, r0
str     r3, [fp, #-12]
```

Optimized Assembly

```
ldr     ip, [fp, #-24]
ldr     r4, [fp, #-28]
mov r0, ip
mov r1, r4
mov r2, #0
udiv r3, r0, r1
mov r5, r3
mov r4, r2

str     r5, [fp, #-16]
str     r4, [fp, #-20]
```

Result Summary

The following metrics were collected using results from 9 different input files:

- Comparing Compression & Decompressed File Sizes
- Comparing Cache Efficiency Metrics (using cachegrind)
- Profiling Time Spent in Each Function (using gprof)

Note: input files were made intenaily so that they included varying alphabet sizes and text lengths

Comparing Compression & Decompressed File Sizes

Name	Size
sample1	1 KB
sample2	3 KB
sample3	4 KB
sample4	32 KB
sample5	67 KB
sample6	127 KB
sample7	228 KB
sample8	98 KB
sample9	123 KB

Decoded File Sizes

Name	Size
sample1.huf	1 KB
sample2.huf	2 KB
sample3.huf	2 KB
sample4.huf	17 KB
sample5.huf	36 KB
sample6.huf	83 KB
sample7.huf	121 KB
sample8.huf	25 KB
sample9.huf	30 KB

Encoded File Sizes

Comparing Cache Efficiency Metrics (using cachegrind)

```
Definitions:
Ir: Number of instruction reads (fetches) from the instruction cache for that line.
I1mr: Number of instruction cache misses for that line.
Dr: Number of data reads from the data cache for that line.
D1mr: Number of data cache misses for that line.
Dw: Number of data writes to the data cache for that line.
D1mw: Number of data cache misses due to writes for that line.

Cache Efficiency (CE) = (Cache Hits / (Cache Hits + Cache Misses))
```

-------- sample6 --------

Ir	I1mr	Dr	D1mr	Dw	D1mw	CE
25,387,939 (31.0%)	7 (0.4%)	4,551,422 (29.6%)	56,839 (83.5%)	2,210,607 (46.5%)	2,037 (7.6%)	0.991368300

-------- sample7 --------

Ir	I1mr	Dr	D1mr	Dw	D1mw	CE
37,895,530 (37.8%)	7 (0.4%)	6,751,020 (35.4%)	4,917 (26.1%)	3,259,030 (51.7%)	3,642 (16.9%)	0.998905360

-------- sample8 --------

Ir	I1mr	Dr	D1mr	Dw	D1mw	CE
5,377,837 (30.5%)	7 (0.4%)	900,197 (26.9%)	514 (7.4%)	400,012 (23.7%)	1,565 (9.7%)	0.998403578

-------- sample9 --------

Ir	I1mr	Dr	D1mr	Dw	D1mw	CE
8,236,771 (35.9%)	7 (0.4%)	1,375,197 (31.4%)	603 (7.5%)	625,012 (30.8%)	1,956 (11.5%)	0.998722268

Profiling Time Spent in Each Function (using gprof)

Sample 7

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
44.12	0.15	0.15	1	150.00	150.00	lutDecoding
26.47	0.24	0.09	1	90.00	90.00	prologue
14.71	0.29	0.05	1	50.00	50.00	treeDecodingBitByBit
11.76	0.33	0.04	1	40.00	40.00	generateEncodedFileBitified
2.94	0.34	0.01	1	10.00	10.00	lutPopulate
0.00	0.34	0.00	2	0.00	0.00	printCodesBitified
0.00	0.34	0.00	1	0.00	0.00	buildHuffmanEncTree
0.00	0.34	0.00	1	0.00	0.00	freeHuffmanTree
0.00	0.34	0.00	1	0.00	0.00	generatehuffmanCodesBitified
0.00	0.34	0.00	1	0.00	0.00	initMinHeap
0.00	0.34	0.00	1	0.00	10.00	lutCreation
0.00	0.34	0.00	1	0.00	0.00	maxTreeDepth

Sample 9

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.33	0.02	0.02	1	20.00	20.00	lutDecoding
33.33	0.04	0.02	1	20.00	20.00	treeDecodingBitByBit
16.67	0.05	0.01	1	10.00	10.00	generateEncodedFileBitified
16.67	0.06	0.01	1	10.00	10.00	prologue
0.00	0.06	0.00	2	0.00	0.00	printCodesBitified
0.00	0.06	0.00	1	0.00	0.00	buildHuffmanEncTree
0.00	0.06	0.00	1	0.00	0.00	freeHuffmanTree
0.00	0.06	0.00	1	0.00	0.00	generatehuffmanCodesBitified
0.00	0.06	0.00	1	0.00	0.00	initMinHeap
0.00	0.06	0.00	1	0.00	0.00	lutCreation
0.00	0.06	0.00	1	0.00	0.00	lutPopulate
0.00	0.06	0.00	1	0.00	0.00	maxTreeDepth

What Went Well & What Could Improve

What Went Well:

- Comparing Huffman Tree decoding vs LUT decoding was helpful for gauging relative performance

What Could Improve:

- Exploring more complexe decoding algorithms such as MPEG: Entropy decoding
- More detailed hardware investigation
- Scaling code to handle larger files

Future Work

- Investigate ways to keep the processor working while data is being decoded by giving it a parallel task.
- Implementing physical hardware improvements such as a microcontroller with faster data cache memory access speed.
- A smart solution that determines whether to use LUT decoding or tree decoding based on maximum code length and projected results.

THANK YOU!

QUESTIONS?

Sources

[1] https://www.maa.org/press/periodicals/convergence/discovery-of-huffman-codes

[2] https://www.keil.com/dd/docs/datashts/samsung/s3c2440_um.pdf

[3] M. Sima "SENG 440 Embedded Systems - Lesson 105: Huffman Decoding", 2023

[4] https://godbolt.org/

[5] https://valgrind.org/