HWK3 – Lookup Table vs. Regression

Jaime Emmanuel Valenzuela Valdivia

Septiembre 2025

Resumen

En este trabajo se compara el tiempo de ejecución entre dos enfoques para calcular una función polinómica en el microprocesador ESP32: la evaluación directa mediante regresión y el uso de una tabla de búsqueda (LUT). El polinomio se obtuvo en el HWK2, y en esta práctica se generó una LUT con `sympy` en Python, se exportó a un archivo `.h`, y se implementaron pruebas en ESP-IDF midiendo el tiempo con `esp_timer_get_time`.

1 Metodología

1.1 Generación de la LUT

A partir de los coeficientes obtenidos en HWK2:

$$f(x) = -11.2885 + 0.014886 \cdot x + 3.8746 \times 10^{-6} \cdot x^2$$

Se utilizó un script en Python con `sympy` para calcular 4096 valores de la función y exportarlos a un archivo `lookuptable.h`.

```
!pip install sympy

Requirement already satisfied: sympy in /usr/local/lib/python3.12/dist-packages (1.13.3)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.12/dist-packages (from sympy) (1.3.0)

import sympy as sp
import numpy as np

x = sp.Symbol('x')

poly = 3.87457794e-06*x**2+1.48860811e-02*x-11.2885192301927
display(poly)

3.87457794 · 10^-6 x^2 + 0.0148860811 x − 11.2885192301927

f = sp.lambdify(x,poly,"numpy")

N = 2**12
display(N)

4096

values = f(np.arange(N))

values[0]

np.float64(-11.2885192301927)

c_array = "uuint32_t percentage_values = {{\n".format(N)
for i in range(0,N,16):
  row = ", ".join(str(int(v)) for v in values [i:i+16])
  c_array += "   " + row + ",\n"
c_array += "};"

print(c_array)

uuint32_t percentage_values = {
    -11, -11, -11, -11, -11, -11, -11, -11, -11, -11, -11, -11, -11, -11, -11, -11,
    -11, -11, -11, -11, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10,
    -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10,
    -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10,
    -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10,
    -10, -10, -10, -10, -10, -9, -9, -9, -9, -9, -9, -9, -9, -9,
    -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9,
    -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9,
    -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9, -9,
    -9, -9, -9, -9, -9, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8,
    -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8,
    -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8,
    -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8,
    -8, -8, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7,
    -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7,
    -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7,
    -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -6, -6,
    -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6,
    -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6,
    -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6, -6,
    -6, -6, -6, -6, -6, -6, -6, -6, -5, -5, -5, -5, -5, -5, -5, -5,
```

Figure 1: Valores de la función.

Listing 1: lookuptable.py

```
from sympy import symbols, lambdify

LUT_SIZE = 4096
x = symbols('x')

# ---- Polinomio HWK2 ----------------------------
intercept = -11.288519230192662
a1 = 1.48860811e-02
a2 = 3.87457794e-06
poly_expr = intercept + a1*x + a2*(x**2)
# ------------------------------------------------

to_int = lambda v: int(round(v))
f = lambdify(x, poly_expr, "math")
lookup = [to_int(f(i)) for i in range(LUT_SIZE)]

header_name = "lookuptable.h"
guard = "LOOKUPTABLE_H"

with open(header_name, "w", encoding="utf-8") as h:
    h.write(f"#ifndef␣{guard}\n#define␣{guard}\n\n")
    h.write("#include␣<stdint.h>\n\n")
    h.write(f"#define␣LUT_SIZE␣{LUT_SIZE}\n\n")
    h.write("static␣const␣int32_t␣lookup_table[LUT_SIZE]␣=␣{\n")

    per_line = 8
    for i, val in enumerate(lookup):
        end = "\n" if (i + 1) % per_line == 0 else "␣"
        h.write(f"{val},{end}")
    if LUT_SIZE % per_line != 0:
        h.write("\n")
    h.write("};\n\n")

    h.write(
        "static␣inline␣int32_t␣lut_get(int␣idx)␣{\n"
        "␣␣␣␣if␣(idx␣<␣0)␣idx␣=␣0;\n"
        "␣␣␣␣if␣(idx␣>=␣LUT_SIZE)␣idx␣=␣LUT_SIZE␣-␣1;\n"
        "␣␣␣␣return␣lookup_table[idx];\n"
        "}\n\n"
    )
    h.write(f"#endif␣//␣{guard}\n")

print(f"Generado␣{header_name}␣con␣{LUT_SIZE}␣entradas.")
```

Listing 2: lookuptable.h

```
from google.colab import files
files.download("lookuptable.h")
```

1.2 Código en ESP32

En el ESP-IDF se implementó la comparación entre regresión y LUT, midiendo tiempos con `esp_timer_get_time()`:

Listing 3: main.c

```c
#include <stdio.h>
#include <inttypes.h>
#include "esp_timer.h"
#include "lookuptable.h"

// y(x) = -11.288519230192662 + 0.0148860811*x + 3.87457794e-06*x^2
static inline int32_t regression_func(int32_t x)
{
    const double intercept = -11.288519230192662;
    const double a1 = 1.48860811e-02;
    const double a2 = 3.87457794e-06;

    const double xd = (double)x;
    const double y  = intercept + a1*xd + a2*xd*xd;

    return (int32_t)(y + (y >= 0 ? 0.5 : -0.5));
}

void app_main(void)
{
    const int base = 1234;
    const int N = 10000;

    volatile int32_t sink = 0;
    int64_t t0, t1;

    int32_t r_demo = regression_func(base);
    int32_t l_demo = lut_get(base);
    printf("Demo x=%d -> regression=%" PRId32 ", lut=%" PRId32 "\n",
           base, r_demo, l_demo);

    // --- Regresion ---
    t0 = esp_timer_get_time();
    for (int i = 0; i < N; ++i) {
        sink ^= regression_func(base + (i & 7));
    }
    t1 = esp_timer_get_time();
    int64_t dt_reg = t1 - t0;

    // --- LUT ---
    t0 = esp_timer_get_time();
    for (int i = 0; i < N; ++i) {
        sink ^= lut_get(base + (i & 7));
    }
    t1 = esp_timer_get_time();
    int64_t dt_lut = t1 - t0;

    printf("Regression: %" PRId64 " us / %d calls   => %.3f ns/call\n",
           dt_reg, N, (dt_reg * 1000.0) / N);
    printf("LUT:        %" PRId64 " us / %d calls   => %.3f ns/call\n",
           dt_lut, N, (dt_lut * 1000.0) / N);

    printf("Sink=%" PRId32 "\n", sink);
}
```

2 Resultados

Durante la ejecución en la tarjeta ESP32 se obtuvo la siguiente salida:

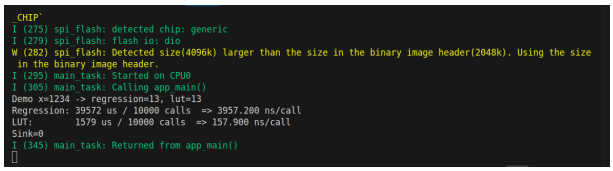

Figure 2: Valores de la terminal al completar el proceso.

```
Demo x=1234 -> regression=13, lut=13
Regression: 39572 us / 10000 calls  => 3957.2 ns/call
LUT:        1579 us / 10000 calls   => 157.9 ns/call
Sink=0
```

2.1 Tabla comparativa

Método	Tiempo total (μs)	ns/llamada	Aceleración
Regresión (polinomio)	39,572	3,957.2	1.0×
LUT (4096 entradas)	1,579	157.9	25.1×

Table 1: Comparativa de tiempo de ejecución en ESP32 (N=10,000, $x = 1234$).

3 Observación

- La LUT ocupa ≈16 KB de memoria (4096 enteros de 4 bytes).
- La precisión fue exacta para el caso de prueba ($x = 1234$, ambos dieron 13).
- El método LUT es ∼25 veces más rápido, lo cual es relevante en aplicaciones de tiempo real.
- La regresión no ocupa memoria adicional, pero es mucho más lenta.

4 Conclusión

El uso de tablas de búsqueda es altamente recomendable cuando se requiere rapidez en cálculos repetitivos en sistemas embebidos, siempre que la memoria disponible lo permita. En este caso, la LUT de 4096 entradas permitió reducir el tiempo de cómputo de 3,957 ns a 158 ns por llamada, logrando una aceleración de 25× sin pérdida de exactitud.