Quplexity: A faster modern quantum computing library written in Assembly.

By Jacob Liam Gill

July 2024

1. Abstract & Overview

Quantum Computer Simulators (QCS's) are often complex pieces of technology where performace is essential. Most modern QCS's like Qrack and Qiskit are written in either Python or C/C++. C++ being the more popular and suitable option for performant simulators. Despite C++ being performant and fast in nature, x86 and ARM/AMR64 Assembly, when written and utilised correctly proves to be significantly faster than C++ whilst also being extremely lightweight in nature. This paper looks at how I have successfully utilised the Assembly language to provide performance and ”weight” benefits to QCS's and the like. All the code for this project can be viewed on the Quplexity github.

2. Matrix Math

In the field of Quantum Computing (QC) mathematical methods involving the manipulation and computation of matrices are very prominent in QCS's. With this in mind I set out to optimaize matrix math in Assembly for Intel and ARM/ARM64 processors.

2.1 Standard Multiplication of 2x2 Matrices

This section will explore the benefits of a standard matrices multiplication function written in Assembly opposed to the traditional C++. The Math that the Assembly function performed can be seen below:

$$\begin{bmatrix} a_A & a_B \\ a_C & a_D \end{bmatrix} \times \begin{bmatrix} b_A & b_B \\ b_C & b_D \end{bmatrix} = \begin{bmatrix} c_A & c_B \\ c_C & c_D \end{bmatrix}$$

I wrote a C++ function that mirrored my Assembly function, I compared the execution times for both using the C++ ”chrono” library. You can view this function in ./x86/math.s of the Quplexity github repository, the function is named ”_gills_matrix2x2”. Below are the calculation and execution time results for Intel & x86:

```
C++ Result: 19 22 43 50
Assembly Result: 19 22 43 50
C++ Duration: 0.531 ms
Assembly Duration: 0.108 ms
```

2.2 Inverse of a 2x2 Matrix

This is an example of how to use the Quplexity function "gills_inv_matrix2x2". It involves finding the determinant of the matrix $\frac{1}{det(A)}$ if $det(a) \neq 0$ then there is a inverse of that matrix. After the determinant has been calculated the function the multiplys the numbers inside the matrix by $\frac{1}{det(A)}$ to produce the resultant matrix. The mathematics behind this function can be viewed below:

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} a_D & a_B \\ a_C & a_A \end{bmatrix}$$

2.3 Pauli-X Matrix

In classical computing, the NOT gate flips the state of a bit from 0 to 1 or from 1 to 0. Similarly, the Pauli-X gate flips the state of a qubit. For instance, if a qubit is in the $|0\rangle$ state, applying the Pauli-X gate will change it to the $|1\rangle$ state, and vice versa. Matrix representation:

$$PX = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Full C++ code implementation (for ASM64 code see ./ARM/gates.s):

```
#include <iostream>
#include <vector>
#include <array>
#include <chrono>
#include <cstdio>

extern "C" {
  void pauli_X(double A[2], double B[2]);
}

int main(){
  std::array<double, 2> A = {1.0, 0.0};
  std::array<double, 2> C = {0.0, 0.0};

  std::cout << "Pauli_X Gate:" << std::endl;
  auto start_asm4 = std::chrono::high_resolution_clock::now();
  pauli_X(A.data(), C.data());
```

```cpp
  auto end_asm4 = std::chrono::high_resolution_clock::now();
  std::chrono::duration<double, std::micro> duration_asm4 = end_asm4 - start_asm4;
  std::cout << "Assembly Duration: " << duration_asm4.count() << " microseconds\n";
  std::cout << "Resultant Vector:" << std::endl;
  std::cout << "[" << C[0] << "]" << "\n" << "[" << C[1] << "]" << "\n\n";

  return 0;
}
```

Listing 1: Pauli-X matrix application with timing.

2.4 Pauli-Z Matrix

The Pauli-Z gate applies a phase flip to the quantum state. Its effect on the basis states $|0\rangle$ and $|1\rangle$ is:

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix} \; |1\rangle = \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\-1\end{bmatrix}$$

Full C++ code implementation (for ASM64 code see ./ARM/gates.s):

```cpp
#include <iostream>
#include <vector>
#include <array>
#include <chrono>
#include <cstdio>

extern "C" {
  void pauli_Z(double A[2], double B[2]);
}

int main(){
  std::array<double, 2> A = {1.0, 0.0};
  std::array<double, 2> C = {0.0, 0.0};

  std::cout << "Pauli_Z Gate:" << std::endl;
  auto start_asm5 = std::chrono::high_resolution_clock::now();
  pauli_Z(A.data(), C.data());
  auto end_asm5 = std::chrono::high_resolution_clock::now();
  std::chrono::duration<double, std::micro> duration_asm5 = end_asm5 - start_asm5;
  std::cout << "Assembly Duration: " << duration_asm5.count() << " microseconds\n";
  std::cout << "Resultant Vector:" << std::endl;
  std::cout << "[" << C[0] << "]" << "\n" << "[" << C[1] << "]" << "\n\n";

  return 0;
```

```
    }
```

Listing 2: Pauli-Z matrix application with timing.

3. Additional Mathematical Methods

This section covers additional mathematical functions and methods that the Quplexity library offers. Some of these functions are specific to a project Quplexity was integrated in.

3.1 Esigx [00, 01, 10] Functions

The esigx function was specifically written for the Quantum Quokka simulator. The esigx function currently has three versions; esigx00, esigx01, esigx10 respectively. The identifier ”00” means the functions takes in two positive vars, ”01” meaning first var should be pos and second neg hence ”10” means the first number parsed should be neg and the second pos. The esigx function is currently only implemented for ARM CPU’s. The esigx function offers minimal improvements speed wise but its still slightly better than its C/C++ counterpart. Math/C code behind the esigx function (_esgix01() used as an example):

$$\boxed{zPtr1- > x = c * z1.x - s * z2.y;}$$

Full C code implementation (for ASM64 code see ./ARM/math.s):

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

typedef struct {
    double x;
    double y;
} ds_Complex;

extern double esigx01(double c, double s, double z1, double z2);
extern double esigx00(double c, double s, double z1, double z2);
extern double esigx10(double c, double s, double z1, double z2);

void c_ds_esigx(ds_Complex *zPtr1, ds_Complex *zPtr2, double theta) {
    double c = cos(theta / 2);  // Compute cosine of theta/2
    double s = sin(theta / 2);  // Compute sine of theta/2
    ds_Complex z1 = *zPtr1;     // Dereference zPtr1 to get the value of the first complex number
    ds_Complex z2 = *zPtr2;     // Dereference zPtr2 to get the value of the second complex number

    zPtr1->x = c * z1.x - s * z2.y;
    zPtr1->y = c * z1.y + s * z2.x;
    zPtr2->x = -s * z1.y + c * z2.x;
    zPtr2->y = s * z1.x + c * z2.y;
}

void asm_ds_esigx(ds_Complex *zPtr1, ds_Complex *zPtr2, double theta) {
    double c = cos(theta / 2);  // Compute cosine of theta/2
    double s = sin(theta / 2);  // Compute sine of theta/2
    ds_Complex z1 = *zPtr1;     // Dereference zPtr1 to get the value of the first complex number
    ds_Complex z2 = *zPtr2;     // Dereference zPtr2 to get the value of the second complex number

    zPtr1->x = esigx01(c, s, z1.x, z2.y);
    zPtr1->y = esigx00(c, s, z1.y, z2.x);
    zPtr2->x = esigx10(-s, c, z1.y, z2.x);
    zPtr2->y = esigx00(s, c, z1.x, z2.y);
}


int main() {
    ds_Complex z1_asm = {1.0, 2.0};  // Example values
    ds_Complex z2_asm = {3.0, 4.0};  // Example values
    ds_Complex z1_c = {1.0, 2.0};    // Same initial values for C
    ds_Complex z2_c = {3.0, 4.0};    // Same initial values for C

    double theta = M_PI / 4;         // 45 degrees

    // Measure execution time of the C function
    clock_t startC = clock();
    c_ds_esigx(&z1_c, &z2_c, theta);
    clock_t endC = clock();
    double cpu_time_usedC = ((double) (endC - startC)) / CLOCKS_PER_SEC;

    // Measure execution time of the assembly function
    clock_t startASM = clock();
    asm_ds_esigx(&z1_asm, &z2_asm, theta);
    clock_t endASM = clock();
    double cpu_time_usedASM = ((double) (endASM - startASM)) / CLOCKS_PER_SEC;

    // Print results for the assembly function
    printf("Assembly ds_esigx:\n");
    printf("Time taken: %f seconds\n", cpu_time_usedASM);
    printf("z1: (%f, %f)\n", z1_asm.x, z1_asm.y);
    printf("z2: (%f, %f)\n", z2_asm.x, z2_asm.y);

    // Print results for the C function
    printf("\nC ds_esigx:\n");
    printf("Time taken: %f seconds\n", cpu_time_usedC);
    printf("z1: (%f, %f)\n", z1_c.x, z1_c.y);
    printf("z2: (%f, %f)\n", z2_c.x, z2_c.y);

    return 0;
}
```

Listing 3: esigx(00, 01, 10) example with timing.

Tested results - Average of 5 trials (tests ran on a Apple M2 8GB RAM, MacOS 13.0.1 (22A400)):

```
Quplexity-ARM

Assembly ds_esigx:
Time taken(AVG): 0.0000022 seconds
z1: (-0.606854, 2.995809)
z2: (2.006272, 4.078202)

C ds_esigx:
Time taken(AVG): 0.0000046 seconds
z1: (-0.606854, 2.995809)
z2: (2.006272, 4.078202)
```