Quplexity-ARM and ARM64 Assembly Manual/User Guide

Jacob Liam Gill

Saturday 27 July 2024

What is Quplexity?

You might be wondering what Quplexity is and in what ways your project can benefit from using it. Quplexity is an exceptionally fast and extremely lightweight modular library for providing Quantum Computer simulators with their complex and precise mathematical logic and other essential functions that these projects need. Quplexity is written in the "Assembly" or "Assembler" language, which is one of the contributing factors to its impressive speeds, extensive hardware support and extremely lightweight nature.

Quplexity-ARM is verified to work on `MacBooks` with `M2 or M1` chips as well as `Raspberry Pi's(3, 4, 5 [64bit!])`, thats right even a Raspberry Pi can run Quplexity with ease.

Navigating the Quplexity Project

The Quplexity codebase is split up into two main parts: the code/tools for Intel CPUs found in the `./Intel` directory and the code/tools for ARM CPUs found in the `./ARM` directory. It is important to note that Quplexity is compiled with the assembler `NASM (x86 Intel)` or `as (ARM and ARM64)`.

Using Quplexity in Your Project

Before you can use the magic that Quplexity offers, you must set up your project correctly to use Quplexity tools/functions. You must compile your C/C++ file with the Quplexity object file linked. For ARM and ARM64, you can do this as follows:

```
# Compile Assembly (for ARM) file:
as math.s -o math.o

# Link:
g++ -no-pie a.cpp math.o -o test
```

Matrix2x2 Example.

Math behind the `_gills_matrix2x2` function:

$$\begin{bmatrix} a_A & a_B \\ a_C & a_D \end{bmatrix} \times \begin{bmatrix} b_A & b_B \\ b_C & b_D \end{bmatrix}$$

After successfully compiling and linking a Quplexity file with a C++ file, you can now begin to use Quplexity functions/logic in that C++ file. In your C++ file, you will need to declare the Assembly functions as external:

```
extern "C" {
  void _gills_matrix2x2(double matrixA[2][2], double matrixB[2][2],
                        double matrixC[2][2]);
}
```

Then you can call that Assembly function from within your C++ code.
An example (Matrix2x2) can be found below:

```
  //Define the Matrix's for this example.
  double matrixA[2][2] = {{8.0, 5.0}, {2.0, 3.0}};
  double matrixB[2][2] = {{19.0, 1.0}, {0.1, 4.0}};
  double matrixC[2][2] = {{0.0, 0.0}, {0.0, 0.0}};

  std::cout << "MUL two 2x2 Matrix:" << std::endl;

  _gills_matrix2x2(matrixA, matrixB, matrixC); //Call ARM Assembly function.

  std::cout << matrixC[0][0] << " " << matrixC[0][1] << std::endl;
  std::cout << matrixC[1][0] << " " << matrixC[1][1] << "\n\n";
```

Inverse of a 2x2 Matrix Example:

This is an example of how to use the Quplexity function "_gills_inv_matrix2x2". Note that you should only pass floats into this function, instead of passing int num1 = 1; pass float num1 = 1.0;

```
  extern "C" {
    void _gills_inv_matrix2x2(double* num1, double* num2, double* num3,
                              double* num4, double* out_matrix);
  }
```

```
  //Make sure the external function is declared.
  double num1 = 1.0;
  double num2 = 2.0;
  double num3 = 3.0;
  double num4 = 4.0;
  double out_matrix[4] = {0.0, 0.0, 0.0, 0.0};

  _gills_inv_matrix2x2(&num1, &num2, &num3, &num4, out_matrix);

  std::cout << "Inverted matrix:" << std::endl;
  std::cout << out_matrix[0] << " " << out_matrix[1] << std::endl;
  std::cout << out_matrix[2] << " " << out_matrix[3] << "\n\n";
```

The determinant or "det(A)" is found by: $det(A) = ad - bc$, if $ad - bc = 0$, mathematically there is no inverse of the matrix 'A'.
The output for the above will be: $-211.5 - 0.5$

Multiplication of two 2x1 Matrix.

This is an example of how to use the Quplexity function "_gills_matrix2x1".
The mathematics behind this function and my example can be viewed below:

$$\begin{bmatrix}0\\1\end{bmatrix} \times \begin{bmatrix}1\\0\end{bmatrix} = [0]$$

This function is great for math involving Qubits at very fast speeds and with very little overhead.
Note that you should only pass floats into this function, instead of passing int num1 = 1; pass float num1 = 1.0;

An example using "_gills_matrix2x1" is as follows:

```
extern "C" {
  void _gills_matrix2x1(double matrixA[2], double matrixB[2],
                        double matrixC[1]);
}
```

```
  //Make sure the external function is declared.

  std::array<double, 2> A = {1.0, 0.0};
  std::array<double, 2> B = {10.0, 299.0};
  std::array<double, 1> C = {0.0};

  std::cout << "MUL-two-2x1-Matrix:" << std::endl;
  _gills_matrix2x1(A.data(), B.data(), C.data());
  std::cout << "[" << C[0] << "]" << "\n\n";
```

The output for the above will be: 0