Quplexity-Intel Assembly Manual/User Guide

Jacob Liam Gill

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.

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`.

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 Intel x86-64, you can do this as follows:

```
# Compile Assembly file:
nasm -f elf64 math.asm -o math.o

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

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 matrix2x2(int64_t* A, int64_t* B, int64_t* C);
  void gills_inv_matrix2x2(float* num1, float* num2,
                           float* num3, float* num4, float* inv_matrix_C);
  float fast_inverse_sqrt(float num);
}
```

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

```
matrix2x2(A.data(), B.data(), C_asm.data()); //Will return a resulting
matrix in the following format: num1 num2 num3 num4
```

OR:

```
float sqrt_ans = fast_inverse_sqrt(20.28865f);
std::cout << "Inverse square root (Assembly): " << sqrt_ans << std::endl;
```

Fast Inverse Square Roots.

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

```
//Make sure the external function is declared.
float my_cool_float = 20.28865f;

float result = fast_inv_sqrt(my_cool_float);
std::cout << result << std::endl;
```

Inverse of a 2x2 Matrix.

This is an example of how to use the Quplexity function "gills_inv_matrix2x2".
The mathematics behind this function can be viewed below:

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

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'.
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_inv_matrix2x2" is as follows:

```
//Make sure the external function
//is declared: void gills_inv_matrix2x2(float* num1
//,float* num2, float* num3, float* num4, float* inv_matrix_C);

std::vector<float> inv_matrix_C(4); //Output matrix

float num1 = 1.0f;
float num2 = 2.0f;
float num3 = 3.0f;
float num4 = 4.0f;

gills_inv_matrix2x2(&num1, &num2, &num3, &num4, inv_matrix_C.data());
for (const auto& val : inv_matrix_C) std::cout << val << " ";
std::cout << std::endl;
```

The output for the above will be: $-0.5 - 1 - 1.5 - 2$

Multiplication of two 1x2 Matrix.

This is an example of how to use the Quplexity function ”gills_matrix1x2”.
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 could be great for math involving Qubits and its blazingly fast and lightweight 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_matrix1x2” is as follows:

```
//Make sure the external function is declared.
//float gills_matrix1x2(float* num1, float* num2, float* num3, float* num4);
float num1 = 0.0f;
float num2 = 1.0f;
float num3 = 1.0f;
float num4 = 0.0f;

float answ = gills_matrix1x2(&num1, &num2, &num3, &num4);
std::cout << answ << std::endl;
```

The output for the above will be: 0