ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA GRADO DE INGENIERÍA INFORMÁTICA

Simulación de algoritmos de computación cuántica en arquitecturas con extensiones vectoriales

Simulation of quantum computing algorithms in architectures with vector extensions

Realizado por
Javier Molina Montiel

Tutorizado por
Eladio Gutiérrez Carrasco
Óscar Plata González

Departamento
Arquitectura de computadores

MÁLAGA, JUNIO DE 2025

Resumen

La computación cuántica es un área emergente dentro de las ciencias de la computación, que busca revolucionar la manera en que se resuelven ciertos problemas complejos aprovechando los principios fundamentales de la mecánica cuántica, como la superposición, la interferencia y el entrelazamiento cuántico. A diferencia de la computación clásica, que utiliza bits binarios para procesar información, la computación cuántica opera con cúbits, los cuales pueden representar simultáneamente los estados 0 y 1, permitiendo una capacidad de procesamiento exponencialmente mayor en determinadas tareas. Sin embargo, debido a la complejidad tecnológica y física necesaria para construir ordenadores cuánticos reales, su acceso sigue siendo limitado y costoso.

Ante esta realidad, los simuladores cuánticos juegan un papel fundamental en el desarrollo y la experimentación de algoritmos cuánticos. Este trabajo presenta el diseño e implementación de un simulador eficiente que imita el comportamiento de un ordenador cuántico mediante el uso de técnicas avanzadas de optimización a nivel de hardware. En particular, se explota la tecnología AVX-512 (Advanced Vector Extensions 512) de Intel, que permite realizar operaciones vectoriales de alto rendimiento, acelerando así el procesamiento de los estados cuánticos simulados.

Se analizarán las distintas estrategias de optimización empleadas y se realizará una comparativa de los resultados obtenidos frente a una implementación tradicional sin aceleración por hardware. El estudio no solo destaca las mejoras en rendimiento que puede aportar el uso de instrucciones vectoriales, sino que también discute las limitaciones inherentes del enfoque propuesto, como el uso intensivo de memoria y la escalabilidad limitada ante un número elevado de qubits. En conjunto, este trabajo busca contribuir al avance de herramientas que permitan la experimentación accesible en computación cuántica, incluso sin acceso a hardware cuántico real.

Palabras clave: **Quantum computing, Hardware optimization, AVX, Quantum simulator**

Índice

1. Introducción **1**

1.1. Motivación . . . 1

1.2. Objetivos . . . 1

1.3. Estructura del documento . . . 2

1.4. Tecnologías usadas . . . 2

2. Contexto y estado del arte **5**

2.1. Breve introducción a la Computación cuántica . . . 5

2.1.1. El cúbit . . . 5

2.1.2. Vector de estado y producto tensorial . . . 8

2.1.3. Puertas lógicas . . . 10

2.1.4. Chips y ordenadores existentes . . . 13

2.2. Técnicas de simulación de computadores cuánticos . . . 15

2.2.1. Métodos de simulación . . . 15

2.2.2. Simulación basada en vector estado . . . 17

2.3. Vectorización hardware . . . 17

2.3.1. SIMD . . . 17

2.3.2. AVX . . . 18

3. Especificaciones **19**

3.1. Puertas soportadas . . . 19

3.2. Medición . . . 20

3.3. Compilado y uso . . . 21

3.4. Procesador objetivo y repertorio de instrucciones . . . 21

4. Desarrollo e implementación de la librería **25**

4.1. Versión secuencial . . . 25

4.2. Vector de estado . . . 26

4.3. Puertas de 1 cúbit . . . 28

4.3.1. Decomposición de la puerta . . . 30
4.3.2. Localidad de las operaciones . . . 32
4.4. Puertas de 2 cúbit . . . 36
4.5. Permutaciones AVX . . . 39
4.6. Puertas de medición . . . 39
4.7. Transformada cuántica de Fourier . . . 40
4.8. Compilación a Python . . . 40

5. Benchmarks y resultados **45**
5.1. Test de corrección . . . 45
5.2. Benchmarks . . . 45
5.2.1. AVX vs implementación simple . . . 46
5.2.2. AVX vs Qiskit . . . 47

6. Conclusiones y Líneas Futuras **49**
6.1. Conclusiones . . . 49
6.2. Posibles mejoras a la librería . . . 50
6.2.1. Optimizaciones específicas para puertas lógicas . . . 50
6.3. Líneas Futuras . . . 50

7. Bibliografía **51**

Apéndice A. Manual de Instalación **53**

1 Introducción

1.1. Motivación

Las últimas décadas han visto un gigantesco desarrollo en la implementación de ordenadores cuánticos. Sin embargo, actualmente sufren de un gran coste de construcción y mantenimiento, que junto a la escasa disponibilidad, provoca que la ejecución de algoritmos en hardware cuántico real sea limitada y laboriosa. La motivación de este trabajo es doble, primero buscamos ofrecer una herramienta que permita ejecutar algoritmos cuánticos en ordenadores tradicionales y facilmente accesibles, y segundo, queremos estudiar la mejora en rendimiento que ofrece el uso de instrucciones vectoriales. En este caso, consideramos la implementación de un simulador cuántico una aplicación perfecta para el tipo de problemas donde el set de instrucciones AVX es relevante.

1.2. Objetivos

El desarrollo de este trabajo tendrá como resultado lo siguiente:

1. Una librería de C, este será el objeto principal de desarrollo, y será una librería de código eficiente, seguro y preciso, que hará el mejor uso posible de las instrucciones intrínsecas de AVX-512, además de reducir el error en comparación a una ejecución en una computadora cuántica real.

2. Una librería de Python, que actuará como un nivel de abstracción superior, ”Wrapper”, y permitirá ejecutar el simulador en scripts escritos en este lenguaje, con el objetivo de ofrecer una sintaxis fácil de usar.

3. Documentación que permita utilizar esta librería, y que sea amigable con el usuario.

4. Esta memoria, que servirá como fuente de documentación del realización del trabajo, además de reflejar los resultados de rendimiento, y el análisis del problema y desarrollo de solución.

1.3. Estructura del documento

El contenido de este documento no tiene como objetivo suponer un recurso de formación completa en profundidad sobre las materias de computación cuántica o estructuras de computadores, sin embargo, contendrá una sección extensa en la que sí se explicará lo necesario para aprender las nociones más fundamentales en el contexto del resto del trabajo. En caso de que el lector desee aprender en detalle alguno de los temas reflejados, se recomienda comenzar leyendo libros y publicaciones referenciadas en el último capítulo de bibliografía.

Este escrito seguirá las recomendaciones e instrucciones de realización de la memoria del trabajo de fin de grado de la Escuela Técnica Superior de Ingeniería Informática de la Universidad de Málaga.

A lo largo de la memoria aparecerán fragmentos de código, imágenes y extractos de texto. Cuando la procedencia de estos recursos sean terceras personas se agregará una referencia directamente a su lado cuando sea posible, o en la bibliografía al final del documento en su defecto.

1.4. Tecnologías usadas

En esta sección se numeran las tecnologías que han sido empleadas para la realización del proyecto:

1. **C**: Lenguaje de programación de propósito general, de tipos estáticos y débilmente tipado. Es un lenguaje que dispone de construcciones de bajo nivel, lo que lo convierte en una opción perfecta para el desarrollo de código donde el rendimiento es prioritario.

2. **GitHub**: Plataforma de desarrollo colaborativo y control de versiones basado en Git. Utilizado mundialmente para el alojamiento de código fuente y el desarrollo de proyectos.

3. **AVX-512**: Conjunto de extensiones SIMD de instrucciones para arquitecturas x86 propuesto por Intel, con soporte de instrucciones de vectores de 512 bits. Implementado por primera vez en 2016. Cuenta con una gran familia de subconjuntos de instrucciones. En este trabajo utilizamos AVX-512F y AVX-512VL, este último permite que la mayoría de operaciones de AVX-512 trabajen con registros de 128 y 256 bits.

   (Intel® AVX-512 https://www.intel.la/content/www/xl/es/products/docs/accelerator-engines/what-is-intel-avx-512.html)

4. **Python**: Lenguaje de programación de alto nivel, dinámico, interpretado y multiplataforma. Es un lenguaje popular por su legibilidad y su fácil aprendizaje, además de contar con una de las bases de soporte y colección de librerías más extensa de cualquiera de los lenguajes de programación.

5. **Cython**: Lenguaje de programación para la escritura de módulos de extensión de Python y C/C++. Basado en la sintaxis de Python, y con agregados que permiten la invocación de funciones en C/C++. Permite obtener la facilidad de uso de Python junto a la velocidad y eficiencia de código escrito en C.

2

Contexto y estado del arte

2.1. Breve introducción a la Computación cuántica

Explicar un ámbito tan extremadamente amplio, complejo y abstracto como la computación cuántica supone un gran desafío, pues algunas de las propiedades más elementales de un sistema cuántico podrían parecer magia, sin embargo se trata de una tecnología con un potencial inmenso y un concepto realmente fascinante.

¿Qué es la computación cuántica?

La computación cuántica o informática cuántica es un paradigma de computación que busca aprovechar propiedades de la física cuántica para realizar procesos computacionales. Es importante aclarar que los ordenadores cuánticos no son "ordenadores más rápidos" ni "mejores que los ordenadores tradicionales", sino que se trata de un paradigma alternativo, con beneficios y desventajas. Conceptualmente, una tarea puede tener una complejidad distinta si es realizada con un algoritmo convencional o con un algoritmo cuántico, de aquí el posible beneficio de esta tecnología.

El ordenador cuántico sustituye el bit tradicional por el **cúbit**, que será la unidad lógica más pequeña.

2.1.1. El cúbit

Un cúbit es la unidad básica de información. La diferencia más fundamental con un bit tradicional es que, mientras que la versión clásica se encontrará en estado 0 (low) o 1 (high), el cúbit hace uso de un fenómeno de la física cuántica conocido como **superposición**, que permite obtener un estado en el que se encuentra "entre el 0 y el 1", con una determinada

probabilidad de obtener un estado u otro.

Es fundamental entender que esta probabilidad de estar en un estado u otro, no es meramente una falta de conocimiento, y realmente se encuentra ya en un estado determinado, sino que es una propiedad física del qubit, que hace que esta incertidumbre sea parte inherente del estado. Al poder alcanzar estados intermedios, es posible codificar un volumen de datos mucho mayor. Es posible hacer que este cúbit colapse a un estado de 0, o de 1, con la probabilidad descrita por el estado de superposición, para ello será necesario realizar una observación de este cúbit.

El estado de un cúbit se puede representar como una combinación lineal de sus dos vectores base $|0\rangle$ y $|1\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Donde α y β son las amplitudes de probabilidad de cada estado base, y **son números complejos**. Es decir, las amplitudes de probabilidad de cualquier estado de nuestro sistema tendrán un componente real y un componente imaginario. También podría ser útil estudiarlo como un componente amplitud y una fase. Gracias a esto es posible decir que el estado del sistema es una función de onda, por este motivo, gran parte de los algoritmos diseñados para este modelo de computación cuántica guardan similitudes con algoritmos de señales periódicas tradicionales.

Al calcular los cuadrados de estos coeficientes se obtienen las respectivas probabilidades. Esto corresponde con el módulo de dicho número complejo.

Además se tiene que:

$$|\alpha|^2 + |\beta|^2 = 1$$

Puesto que la probabilidad total debe ser igual a la unidad.

Otra propiedad cuántica de la que se aprovechará el cúbit será del **entrelazamiento cuántico**. Según esta propiedad será posible entrelazar dos cúbits de manera que ambos estén en superposición hasta que uno de ellos se mida. Una vez observado, la superposición cuántica de ambos cúbits colapsa y será posible conocer también el estado del cúbit no observado. Esto

permitirá a un sistema cuántico obtener fuertes capacidades para el paralelismo operacional, pudiendo hacer provecho de el entrelazamiento para procesar información de multiples cúbits a la vez.

Es habitual ver la superficie de una esfera como representación del espacio de estados del cúbit, a esto se le conoce como esfera de Bloch. Este modelo geométrico fue ideado para poder visualizar de forma más comprensiva el estado de un sistema cuántico. Cada punto de la superficie de la esfera corresponde a un estado del cúbit. Esta representación es útil para visualizar muchas propiedades del cúbit y de sus operaciones. Por ejemplo observaremos que cuenta con dos grados de libertad, los ángulos θ y ϕ. Y las transformaciones del estado se podrán ver como rotaciones.

De forma intuitiva se puede deducir que el espacio de posibles estados de un cúbit es en realidad infinito. Por lo que a la hora de simularlo será necesario realizar una aproximación.

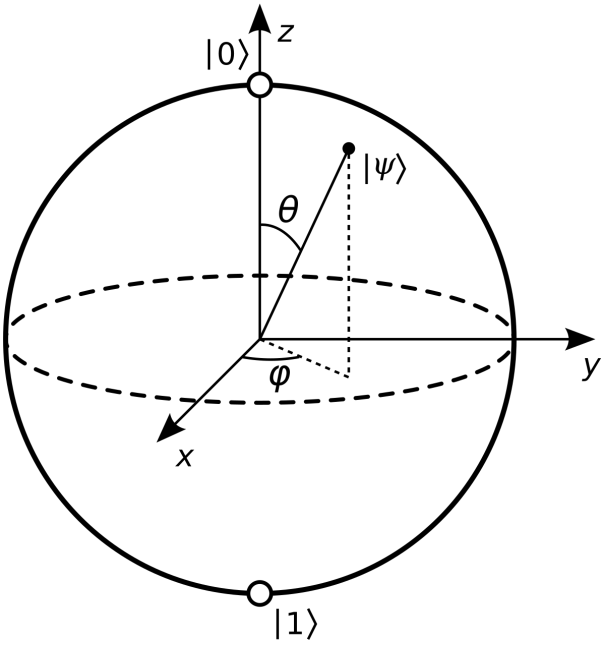

Figura 1: Representación esfera de Bloch. Wikipedia

Este espacio bidimensional descrito con vectores complejos cumple ciertas propiedades adicionales que nos permiten afirmar que es un **espacio de Hilbert**. Esto será de interés más adelante.

2.1.2. Vector de estado y producto tensorial

Sin embargo, este modelo es más apropiado para la visualización, mientras que para la simulación y el estudio de las operaciones implementadas será más práctico trabajar con una representación del estado en forma de vector. Por ejemplo:

$$|\psi_0\rangle = |0\rangle = \underbrace{\begin{bmatrix}1\\0\end{bmatrix}}_{\text{Estado 0}}, \quad |\psi_1\rangle = |1\rangle = \underbrace{\begin{bmatrix}0\\1\end{bmatrix}}_{\text{Estado 1}}$$

Esto es conocido como el **vector estado del sistema**. Será un vector de números complejos, en el que cada elemento representa la probabilidad de obtener un estado concreto. Si tenemos un solo cúbit, solo podrá colapsar en uno de dos posibles estados.

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\end{bmatrix} + \frac{1}{\sqrt{2}}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}\frac{1}{\sqrt{2}}\\\frac{1}{\sqrt{2}}\end{bmatrix}$$

Estado en superposición

Pero un ordenador con un único bit sería un equipo muy limitado, de igual manera, un único cúbit no permitirá hacer mucho, por esto será de gran interés trabajar con sistemas de varios cúbits.

Para poder trabajar con espacios de estado de más de un único cúbit será necesario introducir una operación nueva. Introducimos el producto tensorial. Sin embargo, en el enfoque de este trabajo adoptaremos una definición muy concreta y limitada, aunque de manera rigurosa sea posible definirse de manera mucho más abstracta.

Dado dos espacios de Hilbert $|\phi\rangle$ y $|\psi\rangle$:

$$|\phi\rangle = \sum_{a\in\Sigma} \alpha_a|a\rangle, \quad |\psi\rangle = \sum_{b\in\Gamma} \beta_b|b\rangle$$

El producto tensorial $|\phi\rangle \otimes |\psi\rangle$ es otro espacio de Hilbert tal que:

$$|\phi\rangle \otimes |\psi\rangle = \sum_{(a,b)\in\Sigma\times\Gamma} \alpha_a\beta_b|ab\rangle$$

Por ejemplo, si tenemos un $|v\rangle$ y un $|w\rangle$ tal que:

$$|v\rangle = \begin{bmatrix}a\\b\end{bmatrix}, \quad |w\rangle = \begin{bmatrix}c\\d\end{bmatrix}$$

Entonces tendremos:

$$|v\rangle \otimes |w\rangle = \begin{bmatrix} a \cdot c \\ a \cdot d \\ b \cdot c \\ b \cdot d \end{bmatrix}$$

Formalmente, el estado conjunto de un sistema de N cúbits se describe como un punto en el espacio de Hilbert de dimensión 2^N, el producto tensorial de los N espacios de Hilbert de cada cúbit. Se puede representar el estado compuesto de forma compacta, por ejemplo:

$$|0100\rangle = |0\rangle_1 \otimes |1\rangle_2 \otimes |0\rangle_3 \otimes |0\rangle_4$$

Crucialmente, tendremos entonces que un sistema de n cúbits se encontrará en un estado representado por un vector de 2^n números complejos. Esta será la representación que utilizaremos en adelante.

2.1.3. Puertas lógicas

En la computación tradicional operaremos con los bits utilizando puertas lógicas, por ejemplo la puerta AND, OR, NAND... En la computación cuántica encontraremos un análogo, la **puerta lógica cuántica**.

Estas puertas serán representadas por matrices unitarias. Una puerta de n qubit estará representada por una matriz unitaria de números complejos de dimensión $2^n \times 2^n$.

Podremos aplicar las puertas al estado actual realizando una multiplicación de matriz por vector.

Si tenemos un estado

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

Y la puerta

$$\text{NOT} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Podremos aplicar la puerta de la siguiente manera

$$\text{NOT}|v\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

La aplicación de una matriz identidad devolverá un estado idéntico al original

$$\text{I}|v\rangle = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

Si realizamos el producto tensorial de dos puertas cuánticas obtendremos una puerta cuántica equivalente a la aplicación de estas dos en paralelo. Esto será clave a la hora de aplicar una puerta cuántica de m cúbits a un estado de n cúbits con n >m.

Supongamos que ahora tenemos 2 cúbits en estado

$$|w\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = |00\rangle$$

Y deseamos aplicar una puerta NOT al segundo cúbit. Hasta ahora hemos visto que una puerta de 1 cúbit es equivalente a una matriz de 2×2, pero en este caso necesitaremos construir una puerta de 4×4, que sea equivalente a aplicar la puerta NOT en el segundo cúbit y no aplicar ninguna operación en el primero. Esto será lo mismo que aplicar en paralelo NOT en el segundo cúbit y la identidad en el primero. Para construir esta puerta compuesta realizaremos el producto tensorial de NOT y 1 puerta identidad.

$$NOT \otimes I = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$(NOT \otimes I) \cdot |w\rangle = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = |10\rangle$$

El orden de las puertas al aplicar el producto tensorial influirá en la puerta que obtendremos, por ejemplo, si realizamos el producto tensorial con la puerta NOT en la primera posición desde la derecha, se aplicará al primer cúbit:

$$(I \otimes NOT) \cdot |w\rangle = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = |01\rangle$$

Para crear una puerta aplicable a 3 cúbits bastará con volver a realizar el producto tensorial con la identidad una vez más. Es decir, para aplicar una puerta de m cúbit a un estado de n cúbits realizaremos el producto tensorial con la identidad n - m veces.

Existen puertas de dos o más cúbits en las que uno o más cúbits actúan como control de la operación, haciendo uso de la propiedad de entrelazamiento cuántico, estas serán conocidas como puertas controladas (controlled gates), y solo se aplicarán cuando los cúbits de control colapsen a un estado positivo (1/high).

$$\text{CNOT} \cdot |w\rangle = |00\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = |00\rangle$$

$$\text{CNOT} \cdot |w\rangle = |10\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = |11\rangle$$

2.1.4. Chips y ordenadores existentes

A día de hoy no hay un diseño único de implementación de cúbits, existen muchas tecnologías distintas, con ventajas e inconvenientes diferentes. Sin embargo, no llegaremos a ver implementaciones físicas, en su lugar nos centraremos en ver algunos de los sistemas cuánticos más avanzados y representativos actuales.

1. **IBM**

   IBM es una de las compañías más comprometidas con el desarrollo de tecnologías cuánticas a gran escala. En el evento *IBM Quantum Summit 2023*, presentó dos de sus procesadores más ambiciosos: el **IBM Condor**, que cuenta con 1,121 cúbits, y el **IBM Heron**, con 156 cúbits. Condor representa un hito significativo en cuanto al número de cúbits alcanzado, aunque su conectividad y fidelidad aún deben ser evaluadas en aplicaciones prácticas. Por otro lado, Heron se centra en ofrecer una menor tasa de error y una arquitectura más optimizada para la ejecución de algoritmos, siendo el primero en ser implementado en el sistema *IBM Q System Two*, el nuevo marco de hardware modular que aspira a ser la base de una computación cuántica escalable.

   Además, IBM ha anunciado una hoja de ruta con planes para desarrollar procesadores con decenas de miles de cúbits en los próximos años, utilizando tecnologías de interconexión entre chips y controladores criogénicos avanzados.

2. **Google**

   Google, a través de su división *Google Quantum AI*, ha sido una de las pioneras en demostrar la llamada ”supremacía cuánticaçon su procesador *Sycamore* en 2019. A finales de 2024, la compañía anunció el desarrollo de su nuevo procesador cuántico denominado **Willow**, con una capacidad de 105 cúbits. Aunque aún se encuentra en una etapa temprana de pruebas, Willow ha sido diseñado con una arquitectura que prioriza la reducción de errores y la mejora de la coherencia cuántica, aspectos clave para la ejecución de algoritmos cuánticos con valor práctico.

   Google afirma haber logrado avances importantes en técnicas de corrección de errores y en la implementación de puertas lógicas más estables, lo que podría allanar el camino hacia computadoras cuánticas tolerantes a fallos. Su objetivo a largo plazo es desarrollar una arquitectura que combine un gran número de cúbits físicos con cúbits lógicos altamente fiables mediante códigos de corrección cuántica.

3. **IonQ**

   En 2023, IonQ presentó su sistema **IonQ Forte**, con una capacidad de hasta 32 cúbits físicos y una fidelidad excepcional en operaciones multi-cúbit.

   Además, IonQ ha diseñado sus sistemas para ser accesibles a través de plataformas en la nube como Amazon Braket, Azure Quantum y Google Cloud, lo cual ha facilitado su integración en entornos de investigación y desarrollo industrial. Su hoja de ruta proyecta llegar a más de 1,000 cúbits en 2028, con avances significativos en escalabilidad y estabilidad de los iones atrapados.

4. **Quantinuum**

   Quantinuum, surgida de la fusión entre Honeywell Quantum Solutions y Cambridge Quantum, ha emergido como una de las compañías más avanzadas del sector. Su procesador cuántico, **H1**, ha mostrado avances notables en corrección de errores mediante la implementación práctica de códigos de superficie y cúbits lógicos.

   En 2024, Quantinuum lanzó su sistema **H1-1E**, una evolución de su arquitectura modular, con soporte para hasta 32 cúbits físicos y mejoras significativas en fidelidad de compuertas y tiempos de coherencia. Además, ha destacado por su integración con herramientas de software cuántico como *TKET*, y por la ejecución de experimentos pioneros en simulación de moléculas y materiales complejos. Su estrategia combina hardware robusto con una fuerte orientación a algoritmos cuánticos prácticos en química, criptografía y optimización.

2.2. Técnicas de simulación de computadores cuánticos

2.2.1. Métodos de simulación

1. **Simulación con estado completo (Full-state simulation)**: Esta categoría de simulación consiste en modelar el estado cuántico completo del sistema y aplicar paso a paso las transformaciones definidas por el circuito. Aunque es el enfoque más directo y exacto, también es el más demandante en términos de recursos computacionales.

   - **Vector de estado (State-vector)**: El sistema se describe mediante un vector de amplitudes complejas de tamaño 2^N, donde N es el número de qubits. Es una simulación precisa, pero su coste en memoria crece exponencialmente, lo que restringe su aplicabilidad a un número limitado de qubits.

- **Matriz de densidad (Density-matrix)**: Adecuada para estudiar efectos de decoherencia y ruido, representa el estado como una matriz de dimensiones $2^N \times 2^N$. Este enfoque es aún más costoso que el vector de estado, por lo que se utiliza en contextos muy específicos y con pocos qubits.

2. **Simulación aproximada**: Cuando se busca una mayor eficiencia, es posible recurrir a técnicas que no conservan toda la información del sistema, pero permiten obtener resultados útiles con un coste computacional más bajo.

   - **Estados de producto matricial (MPS/MPO)**: Utiliza una representación basada en tensores conectados localmente, lo que resulta útil para sistemas con entrelazamiento limitado. La memoria requerida escala como $O(N\chi^2)$, siendo χ la dimensión de enlace. Es eficaz para circuitos poco profundos o con estructura regular.
   - **Modelos basados en redes neuronales**: Emplea arquitecturas como las máquinas de Boltzmann restringidas para aproximar estados cuánticos. Estos modelos ofrecen ventajas en ciertos algoritmos (como QAOA), aunque su entrenamiento puede ser complejo debido a la naturaleza no convexa del problema.

3. **Redes de tensores (Tensor networks)**: En este enfoque, el circuito cuántico se traduce a una red de tensores, y se simula mediante operaciones de contracción. Dependiendo del orden en que se contraigan los tensores, la simulación puede ser exacta o aproximada. Este método permite incluir efectos de ruido. Su eficiencia depende de propiedades como el ancho de árbol del circuito.

4. **Otros enfoques**: Existen técnicas adicionales que exploran distintas estrategias para mejorar la eficiencia de la simulación.

   - **Diagramas de decisión**: Representan el circuito como estructuras gráficas que permiten identificar patrones repetitivos y reducir el uso de memoria y tiempo de cómputo. Combinan elementos de redes de tensores con árboles de decisión.
   - **Simuladores adaptados a algoritmos variacionales**: Herramientas específicas para simulaciones en las que el circuito cambia ligeramente en cada iteración, como ocurre en algoritmos variacionales. Estos simuladores aprovechan la reutilización de cálculos entre diferentes ejecuciones para ganar eficiencia.

2.2.2. Simulación basada en vector estado

En este trabajo se optará por la simulación de estado completo a partir de un vector de estado. Esta decisión se ha tomado considerando que es la representación más cercana al conjunto de instrucciones AVX que se utilizará. Muchas librerías de simulación cuántica populares han sido escritas basándose en este paradigma:

1. **Qiskit**

   Qiskit es un kit de desarrollo lanzado en 2017, por IBM. Usa Python y puede ser empleado para programar ordenadores cuánticos reales. Existen implementaciones que usan otro modelo.

2. **Cirq/ Qsim**

   Framework opensource desarrollado por Google. Qsim es el backend que usa simulación de vectores de estado.

3. **QuTiP**

   Soporta distintos modelos de simulación, entre ellos el vector de estado.

2.3. Vectorización hardware

2.3.1. SIMD

SIMD (Single Instruction, Multiple Data) es un tipo de paralelismo a nivel de datos. Basado en la ejecución de una única operación en multiple bloques de datos, comúnmente a nivel hardware, y accesible con conjuntos especiales de instrucciones. El uso de instrucciones SIMD para la optimización es también conocido como vectorización, ya que esta arquitectura se basa en el uso de registros de vectores, y se puede considerar que las operaciones se aplican al vector.

A nivel lógico es similar a lenguajes como APL o J, ya que se trabaja con el array como tipo de datos, y se aplican las operaciones a nivel de array, sin embargo, estos lenguajes de programación de array utilizan abstracción para conseguir este efecto, y las instrucciones a nivel bajo y hardware continuan siendo de elementos de datos individuales. Mientras que por otro lado, la programación SIMD actúa a nivel de registros físicos con conjuntos de datos simultáneamente.

Existen diversas implementaciones de esta arquitectura en función del procesador y del fabricante:

1. **ARM NEON** Disponible para las series de procesadores Arm Cortex-A y Cortex-R, con operaciones para enteros de 16x8, 8x16, 4x32 y 2x64 bits. Y operaciones de punto flotante de 8x16, 4x32 y 2x64 bits.

2. **SVE** SVE (Scalable vector extension) es el sucesor de Neon para la arquitectura Arm, con un rango de 128 a 2048 bits.

3. **Intel SSE** SSE (Streaming SIMD Extensions) fue lanzado en 1999 por Intel para los procesadores Pentium III. Más adelante fueron saliendo los sucesores SSE2, SSE3 y el más reciente SSE4, lanzado en 2007.

4. **Intel AVX** Implementado en una gran gama de procesadores Intel desde 2008, fue inicialmente implementado con instrucciones para vectores de 128 bits. Más adelante se expandió a 256 bits, y finalmente con la salida de AVX-512 se añadió soporte a las instrucciones de 512 bits.

2.3.2. AVX

Como se ha descrito, AVX-512 es el sucesor de una larga familia de conjuntos de instrucciones vectoriales. LLevan siendo implementados en los procesadores intel durante casi dos décadas, y es una tecnología que ya está muy extendida. La mayoría de compiladores modernos tienen configuraciones de optimización automática que se encargan de hacer uso de estas extensiones sin necesidad de reescribir el código o invocarlas de forma manual. De hecho esta es la práctica habitual, es poco común diseñar e implementar código específicamente usando estas instrucciones, sin embargo, también es posible hacer optimizaciones que el compilador no es capaz de realizar, partiendo de un diseño que haga mejor uso de la vectorización. A cambio se sacrificará portabilidad del código, ya que será imposible ejecutarlo en procesadores que no cuenten con las instrucciones necesarias.

3

Especificaciones

Antes de comenzar a analizar la implementación de la librería se discutirán aspectos del diseño cruciales, como el conjunto de instrucciones usadas, las puertas incluidas, y la lista de sistemas soportados.

3.1. Puertas soportadas

En esta sección se enumerarán las puertas cuánticas implementadas en la librería. Es importante aclarar que el algoritmo de aplicación de las puertas al estado fue diseñado de forma genérica, de tal manera que añadir nuevas es extremadamente trivial.

Se ha buscado crear una librería que cuente con un set de puentas cuánticas universales, esto quiere decir que el conjunto de operaciones disponibles es la totalidad de las operaciones que cualquier computador cuántico puede realizar.

Un ejemplo de esto es el conjunto de puertas Clifford (CNOT, H, S) con el operador T.

Además se añadirán las puertas más comúnmente utilizadas, para facilitar la creación de circuitos.

Operadores implementados:

Puerta X (NOT):

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

Puerta Y:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

Puerta Z:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Puerta H (Hadamard):

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Puerta S:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$$

Puerta T:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$$

Puerta CNOT:

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Puerta SWAP:

$$\text{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Puerta de fase controlada (CPHASE):

$$\text{CPHASE} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\phi} \end{bmatrix}$$

3.2. Medición

Al igual que será esencial poder almacenar una representación del estado del sistema sin colapsar (vector de estado), será también fundamental disponer de algún método para observar el sistema y hacerlo colapsar.

Para ello dispondremos de una función para realizar una medición en base computacional, es decir, obtendremos 0 o 1 dependiendo de los estados propios Pauli-Z. Es importante añadir,

que es posible y a menudo de interés algorítmico realizar medidas en distintas bases o sistemas, sin embargo, solo será necesario implementar medición en la base Pauli-Z o Z, ya que será posible realizar una medición genérica a partir de ella.

3.3. Compilado y uso

Se ha decidido implementar el simulador en forma de librería de código en vez de optar por el desarrollo de una aplicación o alguna interfaz gráfica o por terminal. Principalmente para poder facilitar la integración en proyectos sin necesidad de desarrollar una API adicional. Se ha incluido en el repositorio una versión de la librería compilada en Linux GCC. Para poder compilarla en otros sistemas operativos es posible que sea necesario hacer leves cambios a la alocación de memoria dinámica. Sin embargo la mayor parte del código no necesitaría ser modificado.

En linux se han utilizado las siguientes instrucciones:

```
gcc -fPIC -c main.c state\_vector.c qgates.c circuits.c testcases.c vector\
    _masks.c -lm -mavx512vl -mavx512dq

ar rcs lib/libcquantum.a lib/*.o
```

Para compilar la librería de python adicionalmente se ha usado

```
python3 setup.py build\_ext --inplace
```

3.4. Procesador objetivo y repertorio de instrucciones

La librería se ha compilado y testeado en un procesador 11th Gen Intel(R) Core(TM) i5-1135G7. Sin embargo cualquier procesador Intel que cuente con el repertorio de instrucciones AVX necesario podrá usar la librería sin problema.

Se han empleado funciones de dos sets de instrucciones AVX512 concretos:

AVX-512VL Esta extensión de AVX-512 expande varias instrucciones para operar también sobre vectores de 128 y 256 bits.

AVX-512DQ Añade instrucciones nuevas de 32 y 64 bits.

Se ha optado por hacer uso de estos conjuntos de instrucciones ya que ofrecían un gran beneficio en cuanto a rendimiento en funciones específicas.

Designer	Microarchitecture	Year	Support Level																	
			F	CD	ER	PF	BW	DQ	VL	FP16	IFMA	VBMI	VBMI2	BITALG	VPOPCNTDQ	VP2INTERSECT	4VNNIW	4FMAPS	VNNI	BF16
Intel	Knights Landing	2016	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘
	Knights Mill	2017	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✔	✘	✔	✔	✘	✘
	Skylake (server)	2017	✔	✔	✘	✘	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘
	Cannon Lake	2018	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘
	Cascade Lake	2019	✔	✔	✘	✘	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✔	✘
	Cooper Lake	2020	✔	✔	✘	✘	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✔	✔
	Tiger Lake	2020	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔	✘	✘	✔	✘
	Rocket Lake	2021	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✘	✘	✘	✔	✘
	Alder Lake	2021	✔	✔	✘	✘	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✘	✘	✔	✔
	Ice Lake (server)	2021	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✘	✘	✘	✔	✘
	Sapphire Rapids	2023	✔	✔	✘	✘	✔	✔	✔	✔	✔	✔	✔	✔	✔	✘	✘	✘	✔	✔
AMD	Zen 4	2022	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✘	✘	✘	✔	✔
Centaur	CHA		✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘

Figura 2: www.wikichip.org. *AVX set CPU support* [Infographic].

Por otro lado, a continuación se listarán las funciones intrínsecas de AVX que se han utilizado en el código C:

1. **_mm512_loadu_ps()**

   Carga 16 valores flotantes (32 bits) desde memoria no alineada en un registro `__m512`.

2. **_mm512_store_ps()**

   Almacena 16 valores flotantes desde un registro `__m512` en memoria alineada.

3. **_mm512_permutexvar_ps()**

   Permuta los elementos de un vector `__m512` de acuerdo a un índice vectorial `__m512i`.

4. **_mm512_permutex2var_ps()**

   Permuta y mezcla elementos desde dos registros `__m512` usando un vector índice `__m512i`.

5. **_mm512_mul_ps()**

   Multiplica de forma paralela 16 valores flotantes en dos registros `__m512`.

6. **_mm512_add_ps()**

   Suma de forma paralela 16 valores flotantes en dos registros `__m512`.

7. **_mm512_castps512_ps256()**

   Realiza una conversión de tipo (sin mover datos) de un `__m512` a `__m256`.

8. **_mm512_extractf32x8_ps()**

   Extrae un segmento de 8 flotantes (`__m256`) de un registro `__m512` en la posición indicada (0 o 1).

9. **_mm256_hadd_ps()**

   Realiza suma horizontal adyacente en un registro `__m256`; útil para reducción parcial.

10. **_mm512_insertf32x8()**

    Inserta un registro `__m256` en un `__m512` en una posición determinada (0 o 1).

11. **_mm512_castps256_ps512()**

    Convierte un registro `__m256` a un `__m512` sin modificar los bits (casteo bit a bit).

4

Desarrollo e implementación de la librería

A continuación se describirán detalles de la implementación final del código del proyecto. Además se analizarán las motivaciones que fueron consideradas para las decisiones de diseño técnico tomadas.

4.1. Versión secuencial

Primero se ha desarrollado una implementación de varias de las funciones esenciales de la librería sin usar instrucciones vectoriales, con código simple sin aceleraciones hardware. Este código no es incluido ni utilizado para la versión final de la librería, sino que servirá como un punto de referencia y comparación para estudiar el desempeño de una simulación sin emplear estas técnicas. Esto permitirá estudiar si realmente existe beneficio tangible al usar un conjunto de instrucciones SIMD. Es importante añadir que se han desabilitado optimizaciones concretas en el compilado, esto se debe a que la mayoría de compiladores modernos actuales realizan, si es posible, vectorización del código por defecto. Es decir, aunque no se llame de forma explicita, es posible que el compilador sustituya ciertas funciones por implementaciones SIMD. Esto es habitual por ejemplo en iteraciones de vectores usando bucles, que a menudo son ”desenroscadosz sustituidos.

Esta implementación tradicional del simulador cuenta con funciones para aplicar puertas lógicas cuánticas genéricas. Se basa en un vector de estados que hace uso del tipo de número complejos incorporado en la librería estándard del lenguaje çomplex.h”. La versión vectorial de la librería no hace uso de este tipo de datos, sin embargo en la versión tradicional sí, esto

facilitará en gran medida las operaciones.

Por otro lado, en la aplicación de las puertas lógicas se ha aplicado una técnica de decomposición de las puertas, y aplicación usando una máscara de bits, para optimizar la ejecución. Más adelante se discutirá esta técnica en detalle.

Esta librería usada en las pruebas se adjuntará en el repositorio público.

4.2. Vector de estado

A continuación se comenzará a analizar la implementación vectorial. Para el uso de las instrucciones intrínsecas de AVX en el código, será necesario incluir el header **"immintrin.h"**. Esta cabecera dará acceso al repertorio de instrucciones vectoriales avx e incluirá nuevos tipos de datos como __m512 que representarán el registro vectorial avx de 512 bits.

El primer paso necesario para la implementación de la librería es el diseño de una estructura de datos capaz de representar el estado del sistema de una forma eficiente, pero que también permita el manejo de manera cómoda e intuitiva. El simulador diseñado sigue un modelo de simulación de vector de estado, y nuestro objetivo es recrear esta estructura de una manera fiel. Sin embargo encontramos un obstáculo.

Si tenemos un sistema de n cúbits, el vector de estado tendrá un tamaño 2^n, es decir, que dependerá del número de cúbits, mientras que para hacer uso de las instrucciones de AVX-512, será de interés emplear vectores __m512, que cuentan con un tamaño fijo.

Específicamente queremos codificar 2^n números complejos. Serán representados por un par de números flotantes, la parte real y la parte imaginaria. En caso de desear una mayor precisión podría haberse implementado con doubles, sin embargo, se ha decidido priorizar el menor uso de memoria y tiempo computacional que ofrece el uso de floats. En la mayoría de implementaciones de C, los floats ocupan un espacio en memoria de 32 bits. Esto significa que en cada vector avx será posible almacenar 16 números flotantes. Como hemos explicado antes, será necesario almacenar una parte real y una parte imaginaria, por lo que en cada registro vector podremos almacenar 8 números complejos, es decir, 8 probabilidades. Y como para n cúbits necesitamos 2^n probabilidades, tendremos un máximo de 3 cúbits en un vector __m512, por lo que será necesario una representación algo más capaz.

Con este objetivo se ha diseñado la estructura de datos primaria de esta librería, el struct StateVector. Esta será una estructura de C con tres campos:

```
typedef struct {
    __m512 *state_vec;
    int n_qubits;
    int n_vectors;
} StateVector;
```

1. Un puntero a un array de vectores AVX-512. Esto servirá como almacenamiento de los 2^n estados.

2. Un entero que indica el número de cúbits del sistema.

3. Un entero que indica el número de vectores AVX-512 a los que apunta el puntero. Este número no es necesario ya que se puede obtener a partir del valor anterior, sin embargo, ya que es necesario en muchas operaciones, la redundancia de almacenarlo directamente es óptima.

Esta estructura nos permitirá almacenar un número indefinido de estados, y por tanto obtener un sistema de n cúbits. Más concretamente, para n cúbits, el puntero apuntará a un espacio de memoria con $2^{(n-3)}$ vectores __m512. En cada vector __m512 se almacenarán 8 probabilidades en orden, y seguirá un formato de parte real seguida de parte imaginaria, de tal manera que cada número complejo ocupe dos espacios consecutivos del vector.

Este array de vectores se creará haciendo uso de malloc, es decir, se almacena de forma dinámica en el heap.

Adicionalmente se suministran distintas funciones auxiliares para el manejo de esta estructura de datos:

1. **StateVector init_state_vector(int qubits)**

   Esta función permite al usuario la creación de un vector estado $|0\rangle$, es decir, todas sus probabilidades serán 0.0, excepto, la primera probabilidad, que todos los cúbits sean 0.

2. **void state_vector_to_string(StateVector *state_vect)**

   Esta función permitirá al usuario escribir en pantalla el vector de estados en forma de números complejos.

3. **void state_vector_probability_to_string(StateVector *state_vect)**

   Esta función permitirá al usuario escribir en pantalla el vector de estados en forma de porcentaje, para ello calculamos la amplitud haciendo el módulo al cuadrado.

4. **void state_vector_polar_to_string(StateVector *state_vect)**

   También le puede resultar interesante al usuario obtener las probabilidades en base polar, al tratarse de números complejos. Esta función lo permite.

5. **void free_state_vector(StateVector *state_vect)**

   Como se menciona antes, esta estructura reside en el heap del programa, por lo que finalmente será necesario habilitar una función para liberar el espacio con free().

4.3. Puertas de 1 cúbit

La operación más sencilla posible será la aplicación de una puerta de 1 único cúbit. Se ha diseñado e implementado un algoritmo capaz de aplicar una puerta lógica cuántica cualquiera, a partir de la representación matricial de la misma. A continuación se detalla el algoritmo:

Llamaremos a este algoritmo o función äpply_gate()̈. Y tomará tres parámetros de entrada; el vector de estados del sistema, un entero que indica sobre que cúbit se debe aplicar la operación, y como tercer parámetro una matriz, la cual representa la puerta a aplicar.

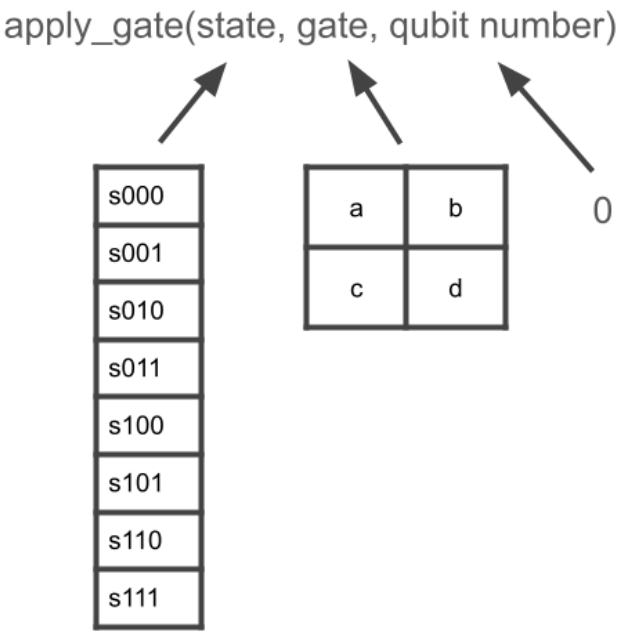

Figura 3: Aplicación puerta lógica cuántica

```
void apply_single_qubit_gate(StateVector *state_vect, int qubit, float *mat
    ){
     int n_qubits_total = state_vect->n_qubits;
     int n_vects = state_vect->n_vectors;
     __m512 *s_vect = state_vect->state_vec;
     qubit = qubit+1;

     __m512 gate[2] = {
         _mm512_set_ps(mat[4], mat[5], -mat[5], mat[4], mat[4], mat[5], -mat
    [5], mat[4], mat[0], mat[1], -mat[1], mat[0], mat[0], mat[1], -mat[1],
    mat[0]),
         _mm512_set_ps(mat[6], mat[7], -mat[7], mat[6], mat[6], mat[7], -mat
    [7], mat[6], mat[2], mat[3], -mat[3], mat[2], mat[2], mat[3], -mat[3],
    mat[2])
     };

     if(qubit > n_qubits_total || qubit < 0){
         fprintf(stderr, "Single qubit gate error!\n");
```

```
        exit(EXIT_FAILURE);
    }
    if(qubit > 3){
        non_local_apply_gate(s_vect, n_vects, qubit, gate);
    }else{
        local_apply_gate(s_vect, n_vects, qubit, gate);
    }
}
```

La matriz que pasamos como parámetro representa una operación de 1 cúbit, y tendrá una dimensión de 2x2 números complejos. En C será representada por un array de 8 flotantes, 4 números complejos (con un número flotante para la parte real y otro para la parte imaginaria). Esta matriz se pasará como parámetro a una función de aplicación local o no local (esto lo veremos más adelante) con sus elementos permutados en dos vectores __m512. Esto será una optimización que reducirá el número de pasos necesarios para aplicar esta puerta.

4.3.1. Decomposición de la puerta

Se hará uso de una optimización esencial para poder aplicar puertas a estados de gran cantidad de cúbits. En la sección de estado del arte, se explicó que era posible componer puertas de n cúbits a partir del producto tensorial de puertas de 1 cúbit paralelas. Pero es posible también revertir este proceso, descomponiendo esta n-cúbit puerta en puertas simples. Pues la matriz necesaria para realizar esta operación es una matriz de 2^nx2^n, sin embargo esto sería un gran consumo de memoria y procesamiento que no es realmente necesario. Es posible descomponer esta operación en pasos más pequeños, 2^{n-1} multiplicaciones de la matriz 2x2 a los pares de elementos del vector de estado que solo difieran en el cúbit m. Una visualización de este proceso:

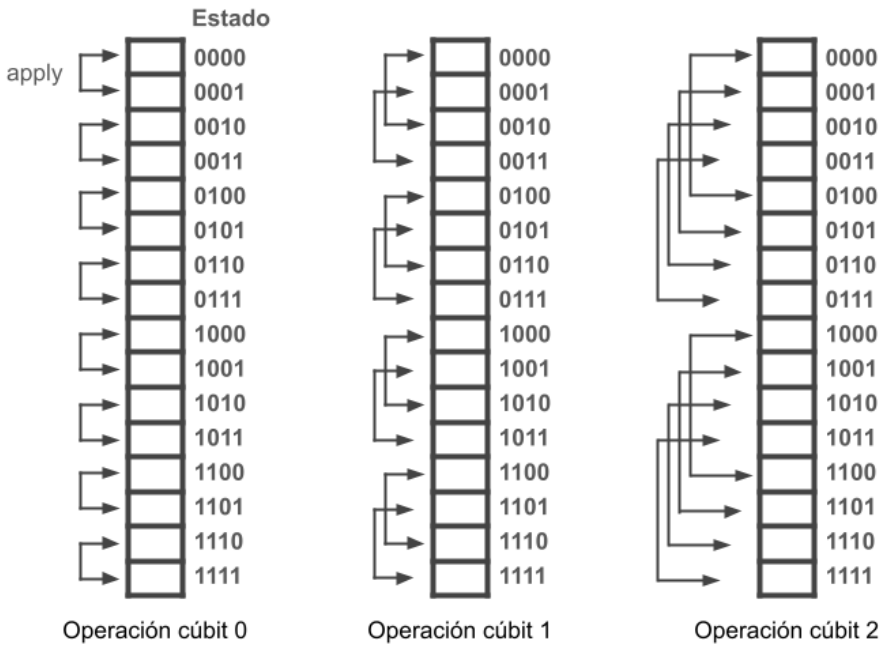

Figura 4: Decomposición de la puerta en operaciones de 2 estados.

Este método es preferible, ya que permite reducir críticamente el espacio de memoria necesario, lo cuál representa de forma consistente la mayor limitación a la hora de diseñar simulaciones.

Teniendo en cuenta que este es el algoritmo que se desea implementar, será de interés usar técnicas de vectorización para la multiplicación de las posiciones deseadas del vector de estado con la matriz 2^nx2^n de la puerta lógica. Primero consideraremos un caso base, en el que las posiciones a multiplicar del vector estado son adyacentes y ordenadas. Esto solo ocurrirá cuando se desee aplicar la puerta lógica sobre el primer cúbit, el cúbit 0:

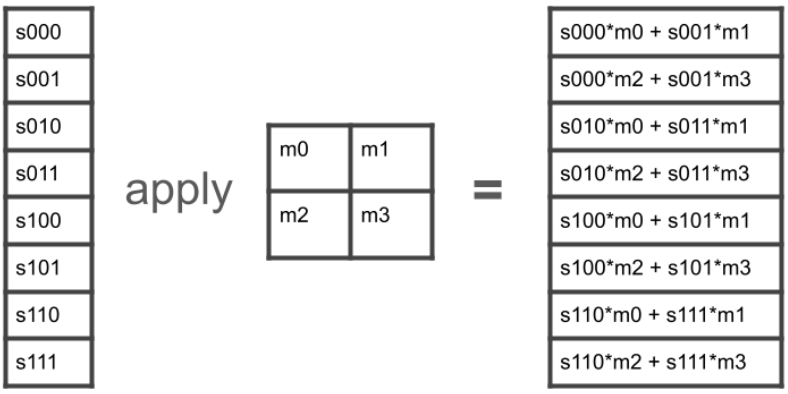

Figura 5: Caso base.

El objetivo de nuestro algoritmo de aplicación de puertas es convertir cualquier estado del vector en el caso base, y aplicar un único algoritmo de multiplicación de matriz.

Para esto se realizarán permutaciones entre los elementos del vector hasta reducirlo al caso base.

4.3.2. Localidad de las operaciones

Sin embargo estas permutaciones no serán uniformes a lo largo del vector de estados completo. Recordemos que los vectores de AVX-512 tendrán una capacidad de almacenar 8 números complejos. Y cuando esta permutación se aplique entre estos 8 números complejos se dirá que es una operación local. Será una operación mucho más sencilla y rápida, pues todos los elementos a permutar se encuentran en el mismo vector, y será posible saber que elementos juntar únicamente a partir de una máscara de bits de la posición. Este es el caso por ejemplo al aplicar la puerta lógica al cúbit 1 o 2:

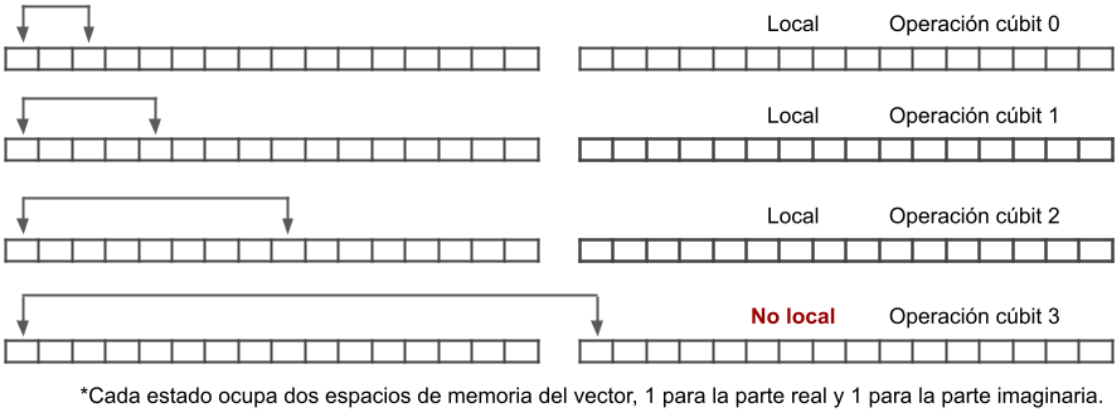

Figura 6: Operaciones locales y no locales.

Sin embargo, cuando se desee aplicar a un cúbit en una posición mayor, la operación no será local, y será necesario permutar entre elementos ubicados en distintos vectores AVX-512. Una vez más es posible determinar que vectores y posiciones permutar a partir del cúbit sobre el que se desea operar. Sin embargo ahora será necesario acceder a multiples vectores AVX a la vez y reorganizar elementos entre ellos, también calculando la distancia entre vectores y posiciones, o leap, a partir de una operación de bits obtenida a partir del índice del cúbit sobre el que queremos aplicar.

```
void local_apply_gate(__m512 *st_vec, int n_vec, int qubit, __m512 *gate){
    __m512i mask1;
    __m512i mask2;
    __m512i mask3;
    __m512i mask4;
    // Máscaras para la permutación
    mask1 = get_mask(MULT2X2_1);
    mask2 = get_mask(MULT2X2_2);
    mask3 = get_mask(MULT2X2_3);
    mask4 = get_mask(MULT2X2_4);
    for(int i = 0; i < n_vec; i++){
        permute_vector(&st_vec[i], qubit);
```

```
        st_vec[i] = complex_matrix_vector_mul_avx512_2x2(st_vec[i], gate,
    mask1, mask2, mask3, mask4);
        permute_vector(&st_vec[i], qubit);
    }
}

void non_local_apply_gate(__m512 *st_vec, int n_vec, int qubit, __m512 *
    gate){

    int factor = qubit - 3;
    __m512i mask1 = get_mask(MULT2X2_1);
    __m512i mask2 = get_mask(MULT2X2_2);
    __m512i mask3 = get_mask(MULT2X2_3);
    __m512i mask4 = get_mask(MULT2X2_4);

    int pointer = 0;
    while(pointer < n_vec){
        swap_half(&st_vec[pointer], &st_vec[pointer+factor]);
        permute_vector(&st_vec[pointer], 3);
        permute_vector(&st_vec[pointer + factor], 3);
        st_vec[pointer] = complex_matrix_vector_mul_avx512_2x2(st_vec[
    pointer], gate, mask1, mask2, mask3, mask4);
        st_vec[pointer+factor] = complex_matrix_vector_mul_avx512_2x2(
    st_vec[pointer+factor], gate, mask1, mask2, mask3, mask4);
        permute_vector(&st_vec[pointer], 3);
        permute_vector(&st_vec[pointer + factor], 3);
        swap_half(&st_vec[pointer], &st_vec[pointer+factor]);
        pointer++;
        if(pointer % factor == 0){
            pointer += factor;
        }
    }
}
```

Una vez se haya realizado toda permutación necesaria, el siguiente paso será la operación de multiplicación matriz-vector de la puerta con cada par de estados. Este será el paso donde el uso de AVX permita optimizar el flujo operacional. Se ha diseñado un algoritmo de mul-

tiplicación especialmente optimizado para realizar esta operación rápidamente gracias a las instrucciones vectoriales.

```
__m512 complex_matrix_vector_mul_avx512_2x2(__m512 st_vec, __m512 *gate,
   __m512i mask1, __m512i mask2, __m512i mask3, __m512i mask4){

    __m512 q1 = _mm512_permutexvar_ps(mask1, st_vec);
    __m512 q2 = _mm512_permutexvar_ps(mask2, st_vec);
    __m512 q3 = _mm512_permutexvar_ps(mask3, st_vec);
    __m512 q4 = _mm512_permutexvar_ps(mask4, st_vec)
```

Primero se crearán cuatro vectores con los elementos del vector original reorganizados. Si los elementos del vector original son: [a+bi, c+di, e+fi, g+hi, j+ki, l+mi, n+oi, p+qi]

Entonces los vectores resultantes serán:

[a+bi, e+fi, a+bi, e+fi],

[c+di, g+hi, c+di, g+hi],

[j+ki, n+oi, j+ki, n+oi],

[l+mi, p+qi, l+mi, p+qi]

```
    __m512 gate_row1 = gate[0];
    __m512 gate_row2 = gate[1];

    __m512 mult1 = _mm512_mul_ps(q1, gate_row1);
    __m512 mult2 = _mm512_mul_ps(q2, gate_row2);
    __m512 mult3 = _mm512_mul_ps(q3, gate_row1);
    __m512 mult4 = _mm512_mul_ps(q4, gate_row2);
```

En el siguiente paso se multiplicarán estos vectores por las columnas de la matriz de la puerta lógica.

```
    __m256 l1 = _mm512_castps512_ps256(mult1);
    __m256 h1 = _mm512_extractf32x8_ps(mult1,1);
    __m256 l2 = _mm512_castps512_ps256(mult2);
    __m256 h2 = _mm512_extractf32x8_ps(mult2,1);
    __m256 l3 = _mm512_castps512_ps256(mult3);
    __m256 h3 = _mm512_extractf32x8_ps(mult3,1);
    __m256 l4 = _mm512_castps512_ps256(mult4);
    __m256 h4 = _mm512_extractf32x8_ps(mult4,1);
```

```
    __m256 a1 = _mm256_hadd_ps(l1,h1);
    __m256 a2 = _mm256_hadd_ps(l2,h2);
    __m256 a3 = _mm256_hadd_ps(l3,h3);
    __m256 a4 = _mm256_hadd_ps(l4,h4);

    __m512 half1 = _mm512_insertf32x8(_mm512_castps256_ps512(a1), a3, 1);
    __m512 half2 = _mm512_insertf32x8(_mm512_castps256_ps512(a2), a4, 1);
```

Se realiza un preprocesado en el que cada vector de 512 bits se convierten en dos mitades de 256 bits, para poder realizar una suma horizontal vectorial (no existe instrínseco de esta operación para vectores de 512 bits). Tras esta operación se unirán los vectores resultado de tal manera que tengamos dos vectores con los resultados de la multiplicación de números complejos fila-columna.

```
    __m512 res = _mm512_add_ps(half1, half2);
    return res;
}
```

Finalmente bastará con sumar estos dos vectores y tendremos el resultado.

4.4. Puertas de 2 cúbit

Gran parte de los conceptos desarrollados para la aplicación de puertas de 1 solo cúbit serán aplicables a las puertas de 2 cúbits, aunque será necesario realizar grandes modificaciones a la implementación.

En el caso de las puertas de dos cúbits, estas estarán representadas por matrices de 4x4 números complejos.

a+Ai	b+Bi	c+Ci	d+Di
e+Ei	f+Fi	g+Gi	h+Hi
q+Qii	j+Ji	k+Ki	l+Li
m+Mi	n+Ni	o+Oi	p+Pi

Figura 7: Puerta lógica de 2 cúbits.

Volveremos a realizar una descomposición y aplicar la puerta a subcomposiciones menores del vector de estado. Sin embargo esta vez no operaremos a pares que difieran por un único cúbit. Sino que al aplicar la puerta con un cúbit de control c, y a un cúbit m, tendremos que aplicarla a los cuatro elementos del vector que difieran en estos dos cúbits, por ejemplo:

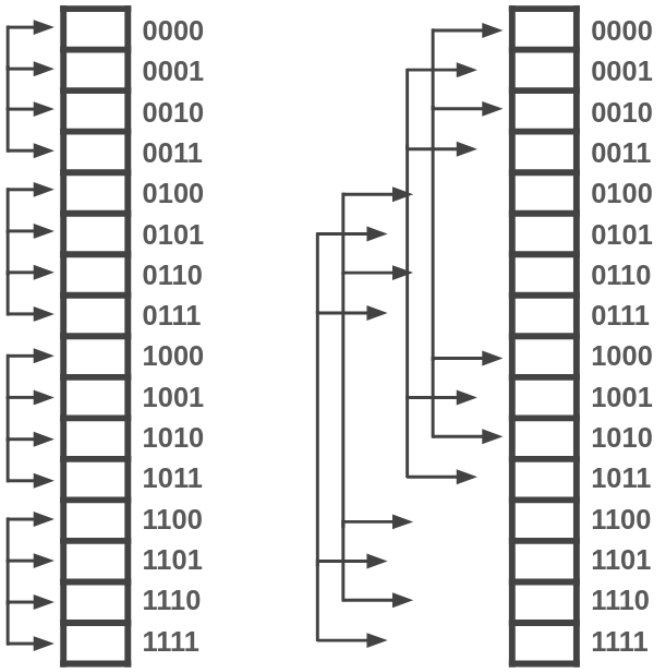

Figura 8: Aplicación puerta 2 cúbits, para cúbits (0, 1) y (1, 3) respectívamente.

Esto aumenta considerablemente la complejidad de las permutaciones tanto en operaciones locales como en operaciones no locales. Además será necesario diferenciar cuál es el cúbit de control y cuál es el que operamos, ya que dependiendo de cuál sea cada uno la permutación será distinta.

Cuando tenemos una puerta de 2 cúbits es además posible que la operación sea parcialmente local, y parcialmente no local:

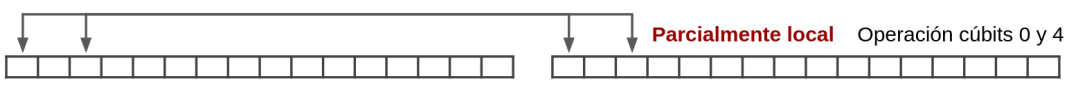

Figura 9: Aplicación puerta 2 cúbits parcialmente local.

Para cada uno de estos casos será necesario aplicar permutaciones diferentes, pero una vez aplicada se realizará siempre la multiplicación de estados por la matriz de la puerta lógica. Antes, para la aplicación de puertas de 1 cúbit fué necesario idear un algoritmo para la multiplicación del vector avx-512 por una matriz 2x2, ambas de números complejos. Ahora tendremos que diseñar un nuevo algoritmo para la multiplicación con matrices de 4x4.

```
    __m512 complex_matrix_vector_mul_avx512_4x4(__m512 st_vec, __m512 *
   gate1, __m512 *gate2, __m512i mask1, __m512i mask2, __m512i mask3,
   __m512i mask4, __m512i mask5, __m512i mask6, __m512i mask7, __m512i
   mask8){
    __m512 l1_1 = _mm512_permutexvar_ps(mask1, st_vec);
    __m512 l2_1 = _mm512_permutexvar_ps(mask3, st_vec);
    __m512 l3_1 = _mm512_permutexvar_ps(mask5, st_vec);
    __m512 l4_1 = _mm512_permutexvar_ps(mask7, st_vec);
    __m512 l1_2 = _mm512_permutexvar_ps(mask2, st_vec);
    __m512 l2_2 = _mm512_permutexvar_ps(mask4, st_vec);
    __m512 l3_2 = _mm512_permutexvar_ps(mask6, st_vec);
    __m512 l4_2 = _mm512_permutexvar_ps(mask8, st_vec);

    __m512 mult1_1 = _mm512_mul_ps(l1_1, gate1[0]);
    __m512 mult2_1 = _mm512_mul_ps(l2_1, gate2[0]);
    __m512 mult3_1 = _mm512_mul_ps(l3_1, gate1[1]);
    __m512 mult4_1 = _mm512_mul_ps(l4_1, gate2[1]);
    __m512 mult1_2 = _mm512_mul_ps(l1_2, gate1[2]);
    __m512 mult2_2 = _mm512_mul_ps(l2_2, gate2[2]);
    __m512 mult3_2 = _mm512_mul_ps(l3_2, gate1[3]);
    __m512 mult4_2 = _mm512_mul_ps(l4_2, gate2[3]);

    __m512 row1 = _mm512_add_ps(mult1_1, mult1_2);
    __m512 row2 = _mm512_add_ps(mult2_1, mult2_2);
    __m512 row3 = _mm512_add_ps(mult3_1, mult3_2);
    __m512 row4 = _mm512_add_ps(mult4_1, mult4_2);

    __m512 result = _mm512_add_ps(_mm512_add_ps(row1, row2), _mm512_add_ps(
   row3, row4));
    return result;
}
```

Este algoritmo se ha diseñado tomando como base el proceso descrito para las operaciones de 1 cúbit, aunque ahora será necesario duplicar el número de vectores temporales construidos y multiplicarlos por el doble de columnas de la matriz que describe la puerta lógica.

4.5. Permutaciones AVX

La mayoría de operaciones implementadas de forma vectorial requieren de la aplicación de multiples permutaciones, casi todas determinadas por máscaras indexadas. Estas máscaras deben están compuestas por vectores AVX-512 con enteros utilizados de index. Estas máscaras son constantes que deben ser calculadas en el diseño de algoritmo y no dependen de valores del tiempo de ejecución. Esto causa que durante la implementación algorítmica al usar estas máscaras el código se vuelva poco legible y redundante. Para evitar esto, se ha tomado la decisión, desde el ámbito del diseño de software, de abstraer gran parte de estas permutaciones, creando una estructura de datos auxiliar que almacene dichas máscaras como campos propios. Se ha seguido un patrón de diseño **Singleton** para ofrecer una única instancia global de esta estructura, con un método para ofrecer acceso a las máscaras desde las funciones que así lo requieran.

```
typedef struct AVXVectorMasks {
    __m512i mult2x2_1;
    __m512i mult2x2_2;
    __m512i mult2x2_3;
    __m512i mult2x2_4;
    __m512i mult4x4_1;
    __m512i mult4x4_2;
    __m512i mult4x4_3;
    __m512i mult4x4_4;

    ...
}
```

4.6. Puertas de medición

La única puerta de medición que será necesaria implementar es la medición Z, que colapsará el estado y devolverá un 1 o un 0 de manera aleatoria siguiendo la probabilidad descrita por

el vector estado. Para simular este comportamiento utilizaremos un generador pseudoaleatorio, pues aunque no sea perfectamente aleatorio, es una aproximación suficiente. Esta puerta será una función que tome como parámetro el cúbit sobre el que se desea realizar la medición, y sumará las probabilidades de que este cúbit colapse a 1, a partir de todos los estados del vector. Finalmente con un número generado pseudoaleatoriamente entre 0 y 1 se realizará una comparación y si la probabilidad sumada total es mayor o igual se devolverá 1, o 0 en el caso contrario.

4.7. Transformada cuántica de Fourier

Después de la implementación de las puertas lógicas y la medición se ha diseñado una función que permite construir el circuito de la transformada cuántica de Fourier generalizado para n cúbits.

Este es un ejemplo de los circuitos que se pueden construir haciendo uso de las herramientas disponibles en la libería.

4.8. Compilación a Python

Tras escribir la librería C y compilarla como una librería estática, se decidió crear un wrapper en Python que permita ejecutar el código C, con todas las aceleraciones AVX desde Python.

Para usar Cython con este objetivo es necesario añadir dos archivos al proyecto: cquantum.pyx y setup.py

En el archivo cquantum.pyx se escribirán las definiciones de las funciones y clases que incluirá esta librería.

Como todo este código ha sido escrito ya en C, podremos importarlo directamente de la librería estática y los headers, haciendo uso de cdef:

```
cdef extern from "state_vector.h":

ctypedef struct StateVector:
    __m512* state_vec
    int n_qubits
    int n_vectors
```

```
    StateVector init_state_vector(int qubits)
    void state_vector_to_string(StateVector* state_vect)
    void state_vector_probability_to_string(StateVector* state_vect)
    void free_state_vector(StateVector* state_vect)
    void state_vector_polar_to_string(StateVector* state_vect)

cdef extern from "qgates.h":

    void debug_gate(StateVector* state_vect, int qubit)
    void debug_gate_d(StateVector* state_vect, int c_qubit, int qubit)

    void X_gate(StateVector* state_vect, int qubit)
    void Y_gate(StateVector* state_vect, int qubit)
    void Z_gate(StateVector* state_vect, int qubit)
    void H_gate(StateVector* state_vect, int qubit)
    void S_gate(StateVector* state_vect, int qubit)
    void T_gate(StateVector* state_vect, int qubit)
    void CNOT_gate(StateVector* state_vect, int control_qubit, int qubit)
    void SWAP_gate(StateVector* state_vect, int control_qubit, int qubit)
    void C_Phase_gate(StateVector* state_vect, int control_qubit, int qubit
   , float phi)

    int Z_measure(StateVector* state_vect, int qubit)

cdef extern from "circuits.h":
    void general_QFT(StateVector* state_vect)
```

Adicionalmente será necesario importar algunas definiciones de immintrin.h:

```
    cdef extern from "immintrin.h":
    ctypedef float __m512
    ctypedef float __m256
    ctypedef int __m256i
    ctypedef int __m512i
    __m512 _mm512_mul_ps(__m512, __m512)
    __m512 _mm512_add_ps(__m512, __m512)
    __m512 _mm512_permutexvar_ps(__m512i, __m512)
    __m512i _mm512_set_epi32(...)  # if used
```

```
    __m256 _mm512_castps512_ps256(__m512)
    __m256 _mm512_extractf32x8_ps(__m512, const int)
    __m256 _mm256_hadd_ps(__m256, __m256)
    __m512 _mm512_insertf32x8(__m512, __m256, const int)
    __m512 _mm512_castps256_ps512(__m256)
```

Pero esto no será suficiente para poder operar correctamente con el vector de estados. Para ello se ha optado por implementar una clase de Python llamada PyStateVector. Esta clase representará el vector de estados del sistema, y contará con métodos para realizar todas las operaciones incluidas por la librería:

```
cdef class PyStateVector:
    cdef StateVector sv
    def __cinit__(self, int qubits):
        self.sv = init_state_vector(qubits)

    def to_string(self):
        state_vector_to_string(&self.sv)

    def probability_to_string(self):
        state_vector_probability_to_string(&self.sv)

    def polar_to_string(self):
        state_vector_polar_to_string(&self.sv)

    def apply_X_gate(self, int qubit):
        X_gate(&self.sv, qubit)

    def apply_Y_gate(self, int qubit):
        Y_gate(&self.sv, qubit)

    def apply_Z_gate(self, int qubit):
        Z_gate(&self.sv, qubit)

    def apply_H_gate(self, int qubit):
        H_gate(&self.sv, qubit)
```

```
    def apply_S_gate(self, int qubit):
        S_gate(&self.sv, qubit)

    def apply_T_gate(self, int qubit):
        T_gate(&self.sv, qubit)

    def apply_CNOT_gate(self, int control_qubit, int qubit):
        CNOT_gate(&self.sv, control_qubit, qubit)

    def apply_SWAP_gate(self, int control_qubit, int qubit):
        SWAP_gate(&self.sv, control_qubit, qubit)

    def apply_C_Phase_gate(self, int control_qubit, int qubit, float phi):
        C_Phase_gate(&self.sv, control_qubit, qubit, phi)

    def debug_gate(self, int qubit):
        debug_gate(&self.sv, qubit)

    def debug_gate_d(self, int c_qubit, int qubit):
        debug_gate_d(&self.sv, c_qubit, qubit)

    def measure_Z(self, int qubit):
        return Z_measure(&self.sv, qubit)

    def apply_general_QFT(self):
        general_QFT(&self.sv)
```

5

Benchmarks y resultados

5.1. Test de corrección

Para garantizar el correcto funcionamiento de los algoritmos y las puertas se ha realizado numerosas pruebas. Se han realizado test con todas las puertas incluidas por defecto, aplicado tanto local como no local. Adicionalmente se han hecho test de circuitos específicos. Y todos los resultados se han comparado directamente con los obtenidos usando la librería de Qiskit. Sin embargo existe una ligera imprecisión al hacer uso de números flotantes de precisión simple. Por lo que los resultados no son exactos sin redondear.

5.2. Benchmarks

¿Es realmente útil el uso de optimizaciones AVX? La forma más práctica de estudiarlo y demostrarlo será el diseño de varios test de Benchmarking que permitan establecer una comparativa directa entre el rendimiento de las implementaciones.

Se han diseñado tres distintos test con objetivos diferentes.

1. **Test de aplicación de puertas simples** Para comprobar la eficiencia del algoritmo de aplicación de puertas de 1 cúbit, se ha escrito un test que aplicará un total de 1000 puertas Hadamard de forma equitativa a todos los cúbits del vector de estados.

2. **Test de aplicación de puertas controladas (2 cúbit)** Al igual que con el test anterior, un aspecto clave a testear será la velocidad al aplicar puertas de 2 cúbits. Se aplicarán 1000 puertas CNOT de manera equitativa tanto respecto a los cúbits de control como los cúbits a aplicar.

3. **Test QFT** Adicionalmente en algunos casos se realizarán test probando la velocidad de ejecución del circuito de QFT.

5.2.1. AVX vs implementación simple

Para comprobar la eficiencia de las instrucciones AVX se comparará las funciones de aplicación de puertas con la implementación simple (no vectorial) de estos algoritmos, aquí se observa el tiempo promedio obtenido:

	Puerta 1 cúbit (NO AVX)	Puerta 1 cúbit (AVX)	Puerta 2 cúbit (NO AVX)	Puerta 2 cúbit (AVX)
NºCúbits				
3	0.00026s	0.00024s	0.00084s	0.00167s
8	0.00531s	0.00438s	0.00758s	0.00184s
15	0.30195s	0.19854	0.87178s	0.10424s
22	48.26190s	24.94301s	102.4904s	12.22360s

Si observamos estos resultados veremos que la implementación de AVX obtiene mejoras del rendimiento exitósamente. Es importante mencionar como aclaración que el algoritmo de puertas de 2 cúbits sin AVX no hace uso óptimo de la memoria, lo que provoca que el tiempo de operación sea ligeramente mayor de lo esperado.

Se observa una tendencia hacia un factor de 1/2 en el tiempo necesario para completar los test de AVX respecto a la implementación simple.

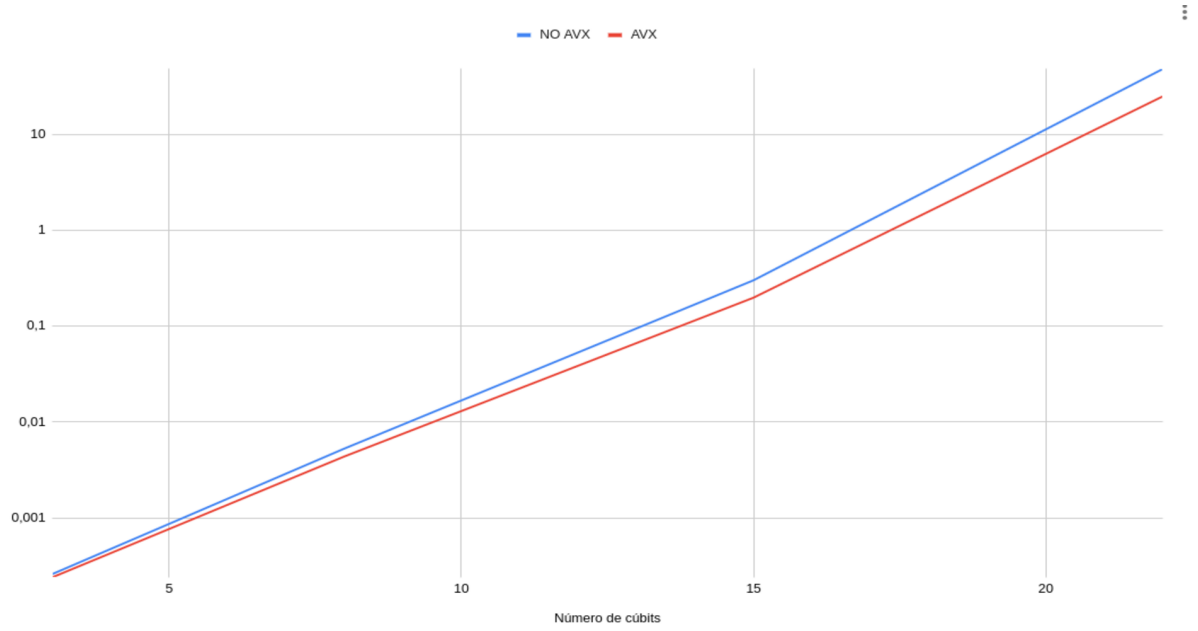

Figura 10: Resultados puerta 1 cúbit. (Escala logarítmica)

En la ejecución de puertas de 2 cúbits también encontraremos una mejora sustancial en el tiempo consumido. Es posible afirmar que las instrucciones vectoriales de AVX han prestado una mejora real en la eficiencia del código, siendo muestra de que el uso de estas técnicas de aceleración hardware son un campo de estudio de interés en el desarrollo de simulaciones cuánticas y tecnologías similares.

5.2.2. AVX vs Qiskit

Adicionalmente es de interés analizar el rendimiento de esta librería frente a soluciones populares para la simulación de circuitos cuánticos.

La mayoría de estas librerías de simulación populares como Qiskit incluyen una gran cantidad de optimizaciones que no se han incluido en la implementación desarrollada en este trabajo, esto significa que el rendimiento será mucho mayor.

Sin embargo, para circuitos con menor número de cúbits, al contar con un overhead mucho menor es posible obtener un rendimiento similar o superior (aproximadamente hasta los 20 cúbits).

Nº Cúbits	Librería AVX	Qiskit
3	0.00004s	0.059256
8	0.00026s	0.059881
15	0.01765s	0.0687
22	3.70672s	0.62939s
28	328.08998s	54.56240s

6

Conclusiones y Líneas Futuras

6.1. Conclusiones

Los resultados obtenidos a lo largo del presente trabajo permiten afirmar que la tecnología AVX, y en general las técnicas de vectorización a nivel de hardware, constituyen un campo de gran relevancia e interés en el contexto de la simulación de computadores cuánticos. En particular, se ha evidenciado que el uso adecuado de instrucciones vectoriales permite aprovechar de forma más eficiente los recursos del procesador, reduciendo significativamente los tiempos de ejecución en operaciones altamente paralelizables, como las aplicadas a vectores de estado cuántico.

Aunque la librería desarrollada en el marco de este TFG no alcanza aún el rendimiento de otras implementaciones altamente optimizadas disponibles en la literatura o en proyectos de código abierto consolidados, sí presenta un overhead computacional notablemente bajo. Esto la convierte en una base sólida y modular sobre la que se pueden incorporar futuras optimizaciones, por ejemplo, a nivel de software (reestructuración de algoritmos, uso de técnicas de loop unrolling, memory alignment, etc.).

Además, este trabajo ha puesto de manifiesto la importancia de un diseño cuidado de las estructuras de datos y del control explícito sobre el acceso a memoria, elementos clave en la eficiencia de este tipo de simulaciones. Se ha comprobado que incluso pequeñas mejoras en la disposición de los datos o en la forma de recorrerlos pueden tener un impacto considerable cuando se trata de operaciones que escalan exponencialmente con el número de cúbits.

Otro aspecto relevante es la escalabilidad del enfoque propuesto. Aunque los test realizados se han centrado en simulaciones de un número limitado de cúbits debido a restricciones de

memoria y tiempo de cómputo, se ha diseñado la librería con una arquitectura que permite su extensión a sistemas más complejos. Esto abre la puerta a futuras líneas de trabajo, como la incorporación de técnicas de paralelización a nivel de múltiples núcleos (multithreading con OpenMP o TBB).

En resumen, este trabajo no solo demuestra la viabilidad del uso de instrucciones AVX en el contexto de la simulación cuántica, sino que también presenta una librería ligera y eficiente que puede servir como punto de partida para proyectos de investigación y desarrollo más ambiciosos. La integración de optimizaciones a distintos niveles (vectorización, paralelización, gestión eficiente de memoria) representa una vía prometedora para hacer frente al elevado coste computacional asociado a este tipo de simulaciones.

6.2. Posibles mejoras a la librería

6.2.1. Optimizaciones específicas para puertas lógicas

Existen algoritmos optimizados para muchas de las puertas lógicas comunmente usadas, actualmente se ha diseñado con una implementación genérica para poder aplicar cualquier puerta sin tener que hacer modificaciones al código. Sin embargo añadiendo implementaciones de estos algoritmos específicos se podría obtener un mejor rendimiento.

6.3. Líneas Futuras

7

Bibliografía

Referencias

1. IBM. Quantum Computing. https://www.ibm.com/think/topics/quantum-computing
2. Universidad de Málaga. TFG: Memoria y Defensa. https://www.uma.es/etsi-informatica/info/72589/tfg-memoria-y-defensa/
3. Intel. What is Intel AVX-512. https://www.intel.la/content/www/xl/es/products/docs/accelerator-engines/what-is-intel-avx-512.html
4. Wikipedia. C (lenguaje de programación). https://es.wikipedia.org/wiki/C_(lenguaje_de_programación
5. Wikipedia. GitHub. https://es.wikipedia.org/wiki/GitHub
6. Wikipedia. Advanced Vector Extensions. https://en.wikipedia.org/wiki/Advanced_Vector_Extensions
7. Wikipedia. Python. https://es.wikipedia.org/wiki/Python
8. Wikipedia. Cython. https://es.wikipedia.org/wiki/Cython
9. Wikipedia. Qubit. https://en.wikipedia.org/wiki/Qubit
10. Wikipedia. Cúbit. https://es.wikipedia.org/wiki/Cúbit
11. Wikipedia. Espacio de Hilbert. https://es.wikipedia.org/wiki/Espacio_de_Hilbert
12. Wikipedia. Producto tensorial. https://es.wikipedia.org/wiki/Producto_tensorial
13. IBM. Basics of Quantum Information: Multiple Systems. https://learning.quantum.ibm.com/course/basi of-quantum-information/multiple-systems
14. https://makbtech.com/top-10-most-powerful-quantum-computers-in-the-world/

15. https://en.wikipedia.org/wiki/IBM_Condor

16. https://es.newsroom.ibm.com/announcements?item=122800

17. https://blog.google/technology/research/google-willow-quantum-chip/

18. https://ionq.com/

19. https://www.quantinuum.com/press-releases/introducing-quantinuum

20. Xu, X., Benjamin, S., Sun, J., Yuan, X., & Zhang, P. (2021). A Herculean task: Classical simulation of quantum computers. *arXiv:2101.03548*. Recuperado de https://arxiv.org/abs/2101.03548

21. https://es.wikipedia.org/wiki/Qiskit

22. https://quantumai.google/cirq

23. https://qutip.org/

Apéndice A
Manual de Instalación

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA