Optimizaciones de alto rendimiento en la construcción de simuladores de circuitos cuánticos: el caso de *QSim*

M.P Cuéllar[1] [0000−0002−9736−1608]

Dpto. Ciencias de la Computación e I.A., ETS. Ingenierías Informática y de Telecomunicación, C/. Pdta. Daniel Saucedo Aranda .n., 18015, Granada, España
manupc@decsai.ugr.es

Abstract. Este artículo presenta una revisión de diversas técnicas de optimización de código para cómputo de alto rendimiento y su aplicación en la construcción de simuladores de circuitos cuánticos. Asimismo, se demuestra la eficacia de estas técnicas mediante la implementación de *QSim*, un nuevo simulador desarrollado íntegramente en C++20. La sección experimental incluye pruebas de simulación sobre problemas de búsqueda en espacios de estados, propios de la Inteligencia Artificial clásica. Los resultados revelan una mejora significativa en el tiempo de ejecución secuencial al compararlo con la simulación de circuitos en *Qiskit*.

Keywords: circuitos cuánticos · simulación · sistemas de alto rendimiento.

1 Introducción

La simulación de la evolución de estados cuánticos constituye una herramienta fundamental para el diseño y desarrollo de algoritmos de computación cuántica, debido al limitado acceso a hardware real y los problemas asociados al mismo (tiempos de coherencia limitados, los errores en la aplicación de puertas, imprecisiones en los mecanismos de medición, etc.).

Centrándonos en el modelo de evolución de estados basado en puertas, existe una amplia variedad de herramientas de simulación, tales como *QASM* y *Aer* para *Qiskit* (IBM), *Cirq* (Google) o *Pennylane*, por citar algunas de las más conocidas. La mayoría de estos sistemas ofrecen una excelente integración con arquitecturas de cómputo de altas prestaciones, principalmente basadas en GPU (como *Aer-GPU* para *Qiskit*, *TensorFlow Quantum* para *Cirq* o *Lightning* para *Pennylane*), apoyándose en bibliotecas de cálculo altamente optimizadas como *Basic Linear Algebra Subsystem (BLAS)*, *cuQuantum* o *TensorFlow*. Pese a ello, la simulación en CPU sigue siendo indispensable, especialmente cuando el tiempo de transferencia del estado cuántico entre el dispositivo de cómputo y el programa anfitrión supera el tiempo que requiere la CPU para efectuar los cálculos localmente. Este escenario es frecuente en las simulaciones habituales, donde los estados cuánticos pueden alcanzar tamaños del orden de gigabytes al

representar 30 qubits o más, o también en el caso de circuitos pequeños donde los tiempos de transferencia de datos entre CPU y GPU superan el tiempo de cómputo local total en CPU. El trabajo en [1] presenta un estudio comparativo detallado sobre el rendimiento de distintos simuladores.

Aunque existen modelos de representación que optimizan el uso de la memoria (como las redes de tensores [4] o los diagramas de decisión cuánticos [3]), el modelo de vector de estado sigue siendo el enfoque predominante para codificar estados cuánticos. En estos casos, la evolución formal se basa en la aplicación de operadores unitarios mediante operaciones matriciales, un área donde las bibliotecas mencionadas en el párrafo anterior destacan por su nivel de optimización. No obstante, el tamaño de dichos operadores crece exponencialmente con respecto al número de qubits, lo que exige incorporar optimizaciones de bajo nivel para mejorar drásticamente los tiempos de ejecución, e incluso hacer viable una simulación cuyo coste en memoria crece exponencialmente con el número de qubits. Algunos estudios previos han analizado diversos tipos de optimizaciones, tanto de propósito general [6] como específicas para tareas de Aprendizaje Automático Cuántico [2]. En concreto, el primer trabajo aborda la reducción del tiempo en arquitecturas de acceso no uniforme a memoria (NUMA), logrando una mejor planificación de los accesos y disminuyendo considerablemente el tiempo global de la simulación.

Los siguientes apartados presentan el simulador *QSim*, desarrollado en C++ íntegramente con la biblioteca estándar C++20, y detallan las optimizaciones de bajo nivel implementadas para acelerar la evolución de vectores de estado. Este simulador permite ajustar el grado de precisión en coma flotante, optimizando así tanto el consumo de memoria como el rendimiento en el acceso a esta. Además, ofrece distintas modalidades de simulación con límites configurables de 16, 32 o 64 qubits. Esta flexibilidad paramétrica permite acelerar significativamente el cómputo cuando los circuitos a simular tienen un tamaño reducido o medio. El conjunto de operaciones cuánticas soportadas abarca todas las puertas estándar de un qubit (X, Y, Z, I, H, T, S, $\sqrt{X}$, $P(\theta)$, $R_z(\theta)$, $R_x(\phi)$, $R_y(\phi)$ y sus correspondientes inversas), las puertas de dos qubits $SWAP$, $R_{xx}(\phi)$, $R_{yy}(\phi)$ y $R_{zz}(\phi)$, así como versiones controladas de todas ellas con múltiples qubits de control, múltiples objetivos (o múltiples pares en el caso de dos qubits) y estados de control configurables. En consecuencia, *QSim* facilita la construcción y simulación general de cualquier circuito cuántico, expandiendo de forma nativa las capacidades de los simuladores actuales a puertas multi-controladas multi-objetivo e incorporando mecanismos avanzados para agilizar los cálculos inherentes a la evolución dictada por el circuito.

2 *QSim* "a vista de pájaro"

El simulador *QSim* ha sido implementado en su totalidad utilizando la biblioteca estándar de C++20, con el objetivo de garantizar la máxima flexibilidad para su compilación e instalación en un PC. Su arquitectura se divide en dos componentes integradas y bien diferenciadas: la interfaz de alto nivel `QCircuit`,

destinada al diseño de circuitos cuánticos, y el motor interno `ALUKet`, encargado de implementar la unidad aritmético-lógica (ALU) para la evolución de los estados. La primera facilita la interacción con el programador mediante una API intuitiva y altamente eficiente para el almacenamiento y la secuenciación de instrucciones. La segunda se enfoca en proporcionar mecanismos de máximo rendimiento para la evolución de los vectores de estado.

2.1 La interfaz *QCircuit*

La clase principal para la construcción de circuitos es `QCircuit`. Su función es registrar y organizar las instrucciones por niveles. Se complementa, además, con las clases `QCircuitQuantikzView` y `QCircuitStringView`, las cuales permiten exportar el circuito a código LaTeX para su representación gráfica mediante el paquete *Quantikz*, o bien visualizarlo como texto plano. La integración entre los mecanismos de alto nivel y de bajo nivel de ejecución se realizan a través de la clase intermedia `QProgram`, que se encarga de transladar instrucciones descritas con la interfaz de alto nivel en un circuito a instrucciones a nivel de bit. Las instrucciones se añaden empleando métodos estandarizados como `x(targets)`, `rx(angle, targets)`, `cz(controls, targets, states)`, etc. Adicionalmente, dispone de métodos `addModule` que posibilitan la inserción de subcircuitos dentro de programas más amplios, promoviendo de este modo la abstracción y la reutilización de componentes. El diseño de `QCircuit` parte de la premisa de que un circuito puede contener decenas o cientos de miles de instrucciones, por lo que la gestión eficiente de su tamaño en memoria resulta crítica.

El núcleo operativo de `QCircuit` y de `QProgram` es la instrucción, encapsulada en la clase `QInstr`. Cada instrucción almacena información sobre el tipo de puerta cuántica a aplicar, los qubits objetivo, los qubits y estados de control, y el parámetro de la puerta (en caso de ser una operación parametrizada). Esta estructura se ha diseñado bajo estrictos criterios de eficiencia computacional: mínimo consumo de memoria, reducción drástica de los accesos a esta y minimización de las operaciones necesarias para gestionar los qubits implicados. Para lograrlo, los qubits y sus estados se representan mediante máscaras de bits configurables de 16, 32 o 64 bits, las cuales limitan el tamaño máximo del circuito a generar. Por ejemplo, una puerta `X` controlada con objetivos en los qubits 3, 4 y 12, y controles en 10, 0 y 2 (con estados 1, 0 y 1 respectivamente), se codifica con una máscara de objetivos `0x1018`, una de controles `0x0405` y una de estados `0x0404`. Si se configura el simulador para un máximo de 16 qubits, cada instrucción ocupará únicamente 7 bytes (2 bytes por cada máscara más 1 byte para el tipo de puerta), independientemente del número de qubits involucrados en cada instrucción. Esta forma de compactar la información de forma estructural dentro de una instrucción favorece la alineación en la memoria RAM, acelerando su copia, acceso y posterior procesamiento por parte de la CPU.

Para las puertas parametrizadas, se incorpora una componente adicional que almacena el ángulo de rotación. Este puede ser un valor real literal o un parámetro dinámico con un valor por determinar (clase `Parameter`). La pre-

cisión de estos valores también es configurable por el usuario: mínima precisión (4 bytes), estándar (8 bytes) o máxima (16 bytes).

Más allá del ahorro y el alineamiento en memoria, este diseño basado en máscaras de bits permite emplear operaciones directas a nivel de bit para resolver cálculos complejos. Desde la versión C++14, la biblioteca estándar proporciona funciones que se traducen en una única instrucción en lenguaje ensamblador, lo cual resulta extremadamente eficiente. Funciones como `popcount` (para contar los bits activos en una máscara), `countr_zero` (para localizar el primer bit activo) o `bit_width` (para determinar el bit más significativo) permiten obtener en un solo ciclo de reloj el número de qubits implicados, el valor $log_2(2^k)$ o la posición del qubit de mayor peso, por poner algunos ejemplos representativos. Al combinar estas funciones con los operadores lógicos clásicos (*AND, OR, NOT, RSHIFT, LSHIFT*), obtenemos ventajas adicionales como la simplificación de manera notable en la detección de errores en tiempo de ejecución, como la verificación rápida de solapamientos inválidos entre qubits de control y objetivo.

2.2 La interfaz interna *ALUKet*

`ALUKet` es el motor de procesamiento encargado de simular la evolución del estado cuántico. Dicho estado se representa como un vector complejo a través de la clase `Ket`, donde la precisión de cada número complejo es ajustable desde los 8 bytes (precisión simple, utilizando dos reales de 4 bytes) hasta los 16 bytes (precisión media) o 32 bytes (alta precisión). Resulta evidente que, conforme aumenta el número de qubits, mantener múltiples copias de los vectores de estado en memoria se vuelve técnica y físicamente inviable. Por ejemplo, simular un circuito de 30 qubits con baja precisión ya exige una capacidad de RAM en torno a los 8 GB. Esta barrera de hardware impide aplicar de manera explícita el modelo teórico de evolución de estados utilizando matrices unitarias densas, lo que hace imprescindible el uso de técnicas de acceso a memoria propias de la computación de alto rendimiento.

Para mitigar esta limitación funcional en circuitos de tamaño reducido, *QSim* incorpora la unidad auxiliar `ALUUnitary`. Su objetivo es calcular explícitamente la matriz unitaria global del circuito, lo cual resulta de gran utilidad para tareas de depuración y validación comparativa frente al desempeño de `ALUKet`. Ambas unidades aritmético-lógicas comparten las estructuras base `Unitary` (que modela el operador) y `Ket`. A su vez, la clase `Unitary` implementa diversas representaciones matriciales que se adaptan dinámicamente al contexto de ejecución: `FullUnitary` (matrices densas), `SparseUnitary` (dispersas), `DiagUnitary` (diagonales), `PermUnitary` (permutaciones) e `IdentityUnitary` (identidad, altamente optimizada debido a su uso masivo en este tipo de modelado).

En el caso de `ALUKet`, la aplicación de los operadores unitarios se realiza de forma implícita, sin instanciar la matriz correspondiente en memoria. Este enfoque no solo ahorra una cantidad masiva de RAM, sino que evita las costosas multiplicaciones matriciales de orden $\mathcal{O}(n^{2+x})$. Asimismo, dada la magnitud del vector de estado, `ALUKet` opera *in-place*, utilizando la misma estructura `Ket` como origen y destino. Aún así, la principal desventaja de esta decisión recae

sobre aquellos circuitos que presenten un uso extensivo de puertas con matrices densas (como H, $\sqrt{X}$ o las rotaciones $R_x(\phi)$ y $R_y(\phi)$), ya que su simulación implícita conlleva un mayor esfuerzo computacional relativo.

Para evitar recorrer múltiples veces los gigabytes del vector de estado, `ALUKet` agrupa y ejecuta de forma conjunta todas las instrucciones pertenecientes a un mismo nivel del circuito cuántico. Además, en cada nivel, los operadores a aplicar se clasifican bajo los siguientes criterios:

- Agrupar las instrucciones con matrices diagonales. Esto permite aprovechar la propiedad de que el producto tensorial de matrices diagonales genera otra matriz diagonal. Incluye las puertas Z, T, $T^\dagger$, S, $S^\dagger$, $P(\theta)$, $R_z(\theta)$, $R_{zz}(\theta)$ y sus variantes controladas o multi-objetivo.
- Agrupar las instrucciones con matrices de permutación. Incluye las puertas X, $SWAP$ y todas sus derivaciones controladas.
- Agrupar el resto de instrucciones bajo la categoría de matrices unitarias densas. Abarca puertas como H, $\sqrt{X}$, $\sqrt{X}^\dagger$, $R_x(\phi)$, $R_y(\phi)$, $R_{xx}(\phi)$, $R_{yy}(\phi)$ y sus respectivas versiones de control.

Durante la ejecución, `ALUKet` recibe el estado inicial y procesa el circuito iterando nivel por nivel. En cada paso, las operaciones se aíslan según las categorías descritas y se aplican secuencialmente sobre el `Ket` mediante el siguiente flujo de trabajo:

1. Obtener el subconjunto D de instrucciones diagonales.
2. Obtener el subconjunto P de instrucciones de permutación.
3. Obtener el subconjunto G de instrucciones densas.
4. Determinar la permutación global K requerida para realinear el vector de estado respecto a los qubits afectados.
5. Aplicar la permutación K sobre el vector de estado.
6. Ejecutar las operaciones de los subconjuntos D, P y G sobre el vector.
7. Aplicar la permutación inversa K^{-1} para restaurar el estado lógico del sistema.

Esta descomposición es matemáticamente válida gracias a la propiedad distributiva del producto tensor $(A \otimes B) = (A \otimes I)(I \otimes B)$. De esta forma, cada bloque lógico de puertas se puede optimizar por separado sin comprometer la integridad del resultado final. Por su parte, la permutación K reorganiza transitoriamente los datos en memoria para que el modelo formal de cada puerta se pueda aplicar directamente. Esto elimina la necesidad de insertar puertas artificiales ($SWAP$ o X) al operar sobre qubits no consecutivos o al procesar controles de estado invertido. La siguiente sección detalla estas optimizaciones a nivel de simulador software.

3 Optimizaciones de alto rendimiento incluidas en *QSim*

Antes de abordar las técnicas concretas integradas en el simulador, es pertinente repasar la jerarquía de memoria en las arquitecturas de CPU contemporáneas.

El acceso a los datos se distribuye típicamente en tres niveles de memoria caché internos al microprocesador (L1, L2 y L3) y la memoria principal (RAM o disco). Mientras que el acceso a la caché es prácticamente instantáneo (≈1 ns para L1, ≈4 ns para L2, ≈15 ns para L3), su capacidad de almacenamiento es limitada (desde los ≈64 KB en la L1 hasta los ≈8 MB en la L3). Por el contrario, la memoria RAM presenta latencias considerablemente mayores (≈80 ns), pero ofrece el volumen necesario para manejar estados cuánticos (de 16 GB a 64 GB). Cuando la CPU requiere un dato, primero lo busca en su caché local; si este no se encuentra, se produce lo que se denomina *fallo de caché* y la lectura debe redirigirse a la memoria principal. Estas transferencias ocurren siempre en bloques contiguos, estandarizados habitualmente en 64 bytes.

En el paradigma del cómputo de altas prestaciones, minimizar la tasa de fallos de caché resulta un objetivo imperativo. Es crucial estructurar los datos de un programa de manera que queden alineados con los límites de 64 bytes de las líneas de caché, evitando lecturas solapadas e innecesarias. Simultáneamente, el software debe explotar a fondo el paralelismo multinúcleo y las instrucciones vectorizadas del procesador (SIMD: *Single Instruction Multiple Data*).

La simulación de puertas puramente diagonales se beneficia enormemente de estos principios, ya que su modelo solo exige recorrer el vector de estado una única vez. A fin de maximizar este rendimiento, `ALUKet` incorpora lógicas algorítmicas especializadas. Tomemos como ejemplo la ejecución de una puerta `Z` multi-objetivo ($Z^{\otimes n}$). Como ilustra la ecuación 1 para el caso de dos qubits ($Z^{\otimes 2}$), aplicar esta matriz sobre un vector $\boldsymbol{v} = (v_0, ..., v_{2^n-1})$ tiene como único efecto matemático invertir el signo de exactamente la mitad de sus componentes. Específicamente, los cambios se aplican sobre aquellas amplitudes v_i cuyo índice i, expresado en binario, contenga un número impar de bits activos (por ejemplo, para $Z^{\otimes 2}$ los índices serían $i = 1, 2$; para $Z^{\otimes 3}$ serían $i = 1, 2, 4, 7$). Al omitir la manipulación de la otra mitad del vector, el cual puede alcanzar un tamaño de gigabytes, se divide a la mitad el tráfico hacia el bus de la RAM, lo que deriva en una reducción drástica de los tiempos totales de ejecución.

$$Z^{\otimes 2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

Esta heurística es extrapolable a casi la totalidad de las operaciones diagonales, dado que todas ellas basan su acción en la inducción de un cambio de fase focalizado en la entrada inferior derecha de sus respectivas sub-matrices. La única adaptación reside en la fase compleja a aplicar. Sin embargo, para generalizar este modelo de aceleración es necesario considerar la posición (o *significancia*) topológica que ocupa la puerta dentro de la línea de qubits. Para un sistema con $n = n_1 + n_2 + n_3$ qubits, una operación focalizada en la franja central ($Z^{\otimes n_2}$) se comportaría matricialmente como $U_1 \otimes Z^{\otimes n_2} \otimes U_3$. En este escenario, la evaluación del cambio de signo no se procesa componente a componente, sino mediante el desplazamiento en bloques continuos de memoria. Por ejemplo, al

observar el producto tensorial simple $Z \otimes I$ de la ecuación 2, constatamos que el tamaño lógico del bloque correspondiente a los bits menos significativos equivale a $2^{n_3} = 2$. El salto de fase afectará exclusivamente a bloques completos de tamaño 2, cuyo índice de bloque presente un número impar de unos.

$$Z^{\otimes 2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \tag{2}$$

En base a este comportamiento predecible, una de las optimizaciones de diseño de mayor impacto en *QSim* consiste en forzar al iterador del algoritmo a operar saltando exclusivamente sobre aquellos bloques contiguos de memoria que deban asimilar el cambio de fase, evitando de esta forma transferir datos de memoria que no vayan a ser modificados.

Conseguimos realizar el salto a los bloques de memoria sujetos a modificación haciendo uso de la técnica *Parallel Bit Deposit* (PDEP), la cual fue introducida en el repertorio de instrucciones máquina de las CPU de Intel a partir de 2013, y que fue posteriormente adoptada por los microprocesadores AMD. La operación de *Bit Deposit* es altamente eficiente y consume un único ciclo de reloj de la CPU. La técnica PDEP, a nivel de bit, consiste en distribuir un conjunto de bits en un registro de datos según una máscara dada. Por ejemplo, la distribución de los datos binarios (1011) con la máscara (01100110) daría como resultado el dato (0**1**0**00**1**1**0), localizando cada i-ésimo bit de entrada en la i-ésima posición del dato donde la máscara tiene valor 1. Por tanto, se convierte en una herramienta esencial para calcular los bloques del vector de estado que deben modificarse, lo que permite evitar visitar aquellos que no lo requieren. Esta herramienta es especialmente eficiente en el cálculo de bloques en puertas controladas, donde los qubits de control son fijos y no requieren modificación.

Otra optimización posible, principalmente en puertas diagonales, consiste en el precálculo de las fases a aplicar y su implementación en una tabla. Consideremos, por ejemplo, la puerta $S^{\otimes n}$. En el caso de $S^{\otimes 3}$, la matriz unitaria se corresponde con la diagonal $[1, i, i, -1, i, -1, -1, -i]$. Podemos observar que solo existen tres fases posibles a aplicar: $e^{i\frac{\pi}{2}} = i$, $e^{i2\frac{\pi}{2}} = -1$ y $e^{i3\frac{\pi}{2}} = -i$. El exponente nos indica el tamaño de la tabla, la cual se puede precalcular fácilmente con antelación a la aplicación de la operación, para posteriormente utilizar PDEP a fin de distribuir cada fase en la correspondiente posición del vector de estado.

La estrategia PDEP es también fácilmente integrable con las puertas controladas que utilizan diversos estados de control. Si consideramos una operación unitaria U controlada por múltiples qubits, la matriz unitaria resultante de la operación controlada puede interpretarse como una matriz diagonal por bloques del tamaño de U. Por ejemplo, la definición estándar de la matriz de la puerta Toffoli se representa por la diagonal por bloques $[I_2, I_2, I_2, X]$. Si consideramos la puerta Toffoli con posibles estados de control 00, 01 y 10, la matriz resultante es fácilmente calculable localizando el bloque X en la posición dada por los estados de control; de este modo, para el estado 00 (bloque 0) se obtiene la matriz

$[X, I_2, I_2, I_2]$, para 01 (bloque 1) se obtiene $[I_2, X, I_2, I_2]$, y para 10 (bloque 2) se obtiene $[I_2, I_2, X, I_2]$. Únicamente el bloque indicado por el estado de control necesita ser modificado, pudiendo omitir el recorrido por el resto del vector de estado para ahorrar transferencias de datos de memoria y tiempo de ejecución innecesario.

Con respecto a las matrices de permutación, tales como X, $SWAP$ y sus versiones controladas, las operaciones requeridas por la aplicación de estas puertas tienen pocas necesidades de cómputo, aunque sí suponen un incremento sustancial en los tiempos de acceso a memoria requeridos para intercambiar posiciones en el vector de estado. En este aspecto, el cálculo de bloques activos (en caché) y sus correspondientes bloques a permutar resulta clave para minimizar los accesos a memoria. La estrategia utilizada por el simulador *QSim*, en este caso, supone el precálculo de bloques contiguos con un tamaño promedio similar al de la caché L3 y sus correspondientes bloques de permuta, haciendo uso de PDEP para la identificación de los elementos que no se deben permutar (por ejemplo, por ser qubits de control o por tratarse de los extremos de la puerta SWAP) y evitar así cargas de memoria innecesarias.

4 Experimentación

El simulador *QSim* se encuentra disponible para su descarga de libre acceso en el repositorio `https://github.com/manupc/QSim`. En este apartado se llevan a cabo dos pruebas de ejecución con problemas previamente estudiados y publicados en [5]. Estos problemas plantean una búsqueda en espacios de estados clásica de la Inteligencia Artificial, considerando dos casos: un problema de tipo 1, donde el estado objetivo a alcanzar es conocido, y un problema de tipo 2, donde el estado objetivo es desconocido pero se conoce un coste asociado a cada nodo y una cota inferior de dichos costes. El objetivo, en este segundo caso, consiste en obtener un camino desde un estado inicial dado hasta un nodo con el mínimo coste. Ambos tipos de problemas han sido seleccionados por considerarse representativos y por consumir una alta cantidad de recursos de memoria y computacionales, llegando a utilizar hasta un máximo de 30 qubits, agrupar puertas cuánticas y módulos de diverso tipo (QFT, Grover), y requerir una profundidad de circuito elevada (entre 300 y 1200 niveles aproximados, dependiendo de la instancia del problema).

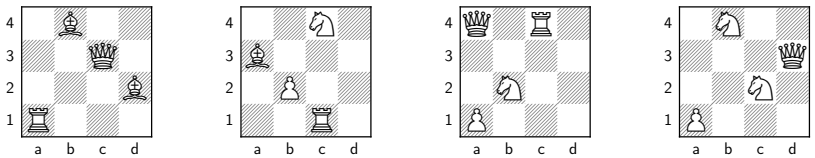


Fig. 1: Instancias de problemas a solucionar

En particular, el problema abordado es un solitario de ajedrez consistente en un conjunto de piezas colocadas en un tablero reducido de tamaño 4×4, donde el objetivo consiste en eliminar todas las piezas salvo una. Un ejemplo de este pasatiempo se ilustra en la Figura 1, donde se muestran cuatro configuraciones o instancias distintas del problema. La solución de la primera instancia pasa por aplicar los movimientos *b4c3*, *d2c3*, *c3a1* o, alternativamente, *d2c3*, *b4c3*, *c3a1*. La solución de la segunda instancia es la secuencia de movimientos *c4b2*, *a3b2*, *b2c1*. Por último, las soluciones de las dos últimas instancias son las secuencias *b2a4*, *c4a4*,*a4a1* y *d3c2*, *b4c2*, *c2a1*, respectivamente. El número de qubits necesario para resolver cada instancia varía dependiendo del número de acciones posibles y del tamaño del espacio de estados resultante. En el caso de la búsqueda del nodo objetivo por coste, definimos el coste $c(s)$ de un estado s del tablero como el número de piezas que sobran en el mismo para alcanzar la solución. Se puede consultar un mayor detalle sobre el diseño del algoritmo de búsqueda en la referencia [5].

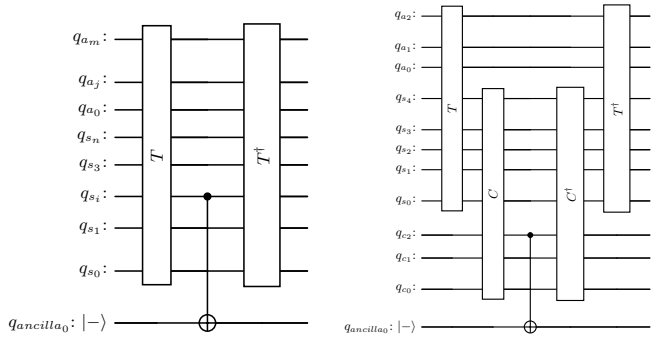


Fig. 2: Oráculos de los casos de estudio 1 (izquierda) y 2 (derecha)

En todos los casos se asume un circuito con matriz unitaria T que implementa la función de transición $T(s, a)$ entre estados, donde s es el estado actual del tablero, a la acción que se aplica sobre s, y $T(s, a)$ el estado resultante. Adicionalmente, utilizamos un qubit auxiliar para generar el oráculo que marca la solución a buscar por el algoritmo de Grover. En el caso de búsqueda con nodo destino conocido, el oráculo a utilizar se corresponde con el circuito de la parte izquierda de la Figura 2. En el caso de búsqueda con nodo destino desconocido pero coste conocido, también resulta necesario un circuito que calcule el coste $c(s)$ de un estado (con matriz unitaria C). El oráculo resultante de esta versión del problema se encuentra en la parte derecha de la Figura 2.

En ambos casos, la solución se ha implementado tanto en Qiskit como en el simulador *QSim*, a fin de poder comparar los tiempos de ejecución de ambos simuladores. En todos los experimentos se tomaron un total de 128 muestras sobre el resultado final del circuito, y ambos tipos de ejecuciones proporcionaron siempre la misma solución final en cada instancia. De este modo, conseguimos

validar el correcto funcionamiento del simulador y de los circuitos generados para resolver el problema.

Table 1: Tiempos de ejecución obtenidos con el simulador *QSim* y con Qiskit en la búsqueda con estado final conocido.

Método	Instancia	Tipo de medición	Tiempo medio ± s.d. (s)
QSim	1	Real	45.259 ± 5.26398
QSim	1	Secuencial	44.9053 ± 5.2711
Qiskit	1	Real	96.2557 ± 2.37393
Qiskit	1	Secuencial	554.384 ± 5.93107
QSim	2	Real	16.963 ± 2.89513
QSim	2	Secuencial	16.8513 ± 2.87892
Qiskit	2	Real	24.059 ± 0.292845
Qiskit	2	Secuencial	135.228 ± 1.1961
QSim	3	Real	178.382 ± 8.14145
QSim	3	Secuencial	177.617 ± 8.11655
Qiskit	3	Real	199.46 ± 0.854373
Qiskit	3	Secuencial	1161.72 ± 4.75556
QSim	4	Real	4.08233 ± 0.701002
QSim	4	Secuencial	4.05933 ± 0.701695
Qiskit	4	Real	6.31833 ± 0.10472
Qiskit	4	Secuencial	31.0767 ± 0.521677

Con respecto al rendimiento, medimos el tiempo de ejecución de 3 ejecuciones distintas de cada algoritmo para cada instancia del problema, utilizando para ello la instrucción `time` de Linux. Esta instrucción resulta de especial utilidad en este contexto, dado que permite devolver tanto el tiempo de ejecución real de cada proceso como el tiempo acumulado secuencial cuando el programa se ha paralelizado en los diferentes núcleos de la CPU. Esto nos permite valorar no solo el tiempo que tarda cada programa en ejecutarse, sino también el tiempo que Qiskit hubiese tardado en resolver el problema en caso de no disponer de mecanismos de paralelización de instrucciones. Este hecho hace que la comparativa sea más justa, dado que *QSim*, en su estado actual, no incorpora mecanismos de paralelización.

Las Tablas 1 y 2 muestran los resultados obtenidos por cada método (*QSim* o *Qiskit*) en cada tipo de problema, para cada instancia dada en la Figura 1. La columna *Tiempo medio ± s.d.* muestra el tiempo promedio, junto con la desviación típica asociada, de 3 ejecuciones diferentes de cada instancia. El tipo de medición indica si el tiempo mostrado se corresponde con el tiempo real de ejecución (caso *Real*) o con el tiempo de CPU acumulado por todos los núcleos en el caso de que el programa se ejecutase secuencialmente sin paralelización.

Podemos observar, para el caso de *QSim*, que ambos tiempos (real y secuencial) son muy similares en cada problema, debido a que el simulador aún no incluye mecanismos de paralelización de cómputo. Sin embargo, es especial-

Table 2: Tiempos de ejecución obtenidos con el simulador *QSim* y con Qiskit en la búsqueda con estado final desconocido.

Método	Instancia	Tipo de medición	Tiempo medio ± s.d. (s)
QSim	1	Real	2477.85 ± 11.947
QSim	1	Secuencial	2474.5 ± 12.0875
Qiskit	1	Real	970.448 ± 4.31024
Qiskit	1	Secuencial	5221.24 ± 42.0861
QSim	2	Real	776.711 ± 2.44992
QSim	2	Secuencial	775.779 ± 2.3618
Qiskit	2	Real	296.352 ± 3.56598
Qiskit	2	Secuencial	1652.34 ± 0.803493
QSim	3	Real	7434 ± 355.952
QSim	3	Secuencial	7429.04 ± 355.616
Qiskit	3	Real	2349.48 ± 215.666
Qiskit	3	Secuencial	12614.8 ± 135.287
QSim	4	Real	159.525 ± 5.85977
QSim	4	Secuencial	159.368 ± 5.86384
Qiskit	4	Real	55.18 ± 0.0589406
Qiskit	4	Secuencial	323.163 ± 0.376232

mente notable el efecto del paralelismo en Qiskit, el cual reduce drásticamente el tiempo desde la versión secuencial a la versión paralela.

Al comparar los tiempos de simulación de *QSim* con los obtenidos por *Qiskit*, también se observa que el tiempo de ejecución secuencial de *QSim* es notablemente inferior al requerido por *Qiskit*. Esto pone de manifiesto un alto grado de eficiencia, alcanzado gracias a las decisiones de diseño y a los mecanismos de optimización de alto rendimiento implementados. Además, el tiempo de ejecución de *QSim* también es inferior al de *Qiskit* en la primera modalidad del problema, incluso cuando este último hace uso de la paralelización.

Los mecanismos de paralelización de *Qiskit* proporcionan a la simulación una notable mejora, sobre todo en el problema de tipo 2, donde los circuitos presentan una anchura y una profundidad mayores. En este caso, la ejecución paralela de *Qiskit* mejora sustancialmente el tiempo de ejecución real de *QSim*. Sin embargo, si consideramos la ejecución secuencial de ambos métodos, el simulador *QSim* obtiene mejores tiempos en todos los escenarios evaluados.

5 Conclusiones

Las optimizaciones de bajo nivel en *QSim* aportan mejoras significativas, logrando tiempos de ejecución muy prometedores frente a *Qiskit* incluso sin mecanismos de paralelización en CPU o GPU. A continuación, se resumen estas optimizaciones:

– Gestión de qubits y estados de control mediante máscaras de bits, lo que reduce el consumo de memoria y agiliza la ordenación y validación de instrucciones.

– Aplicación implícita de instrucciones frente al modelo matricial clásico. Esto minimiza el uso de memoria, evita multiplicaciones matriciales costosas y permite operar sobre qubits no consecutivos sin permutaciones previas.
– Agrupación de puertas con la misma estructura (diagonales o de permutación) para reducir iteraciones sobre el vector de estado, optimizando el ancho de banda y la localidad en caché.
– Uso de operadores a nivel de bits y *Parallel Bit Deposit* (PDEP) para localizar y procesar exclusivamente los bloques afectados por una operación, evitando accesos improductivos a la memoria.
– Precisión de coma flotante y tamaño de máscaras ajustables, optimizando el almacenamiento global y favoreciendo la localidad en la caché de la CPU.
– Alineamiento de memoria en bloques de 64 bytes para maximizar la localidad en caché y evitar transferencias de datos redundantes.
– Sustitución de cálculos aritméticos estándar por operaciones de bajo coste (ej. multiplicar por $\pm i$ intercambiando la parte real e imaginaria, o usar *bit-shifting* para multiplicaciones o divisiones por potencias de 2).

Las principales líneas de trabajo futuro para perfeccionar *QSim* incluyen:

– Implementar mecanismos de paralelización de operaciones tanto en CPU multinúcleo como en arquitecturas GPU.
– Extender el modelo de simulación hacia representaciones más eficientes en memoria, como las redes de tensores o los diagramas de decisión cuánticos.

Disclosure of Interests. El autor declara no tener conflicto de interés alguno con respecto al contenido del artículo.

References

1. Cerrudo Herrera, J., Talaván-Vega, D., Rodríguez-Oliver, P., Ziabat-Ziabat, A., Rico-Gallego, J.: Benchmarking quantum computing statevector simulators on high-performance computing. SIMULATION (04 2026). https://doi.org/10.1177/00375497261438515
2. Kuzmin, V., Kyriacou, B., Protasevich, T., Papierz, M., Kordzanganeh, M., Melnikov, A.: Tqml simulator: optimized simulation of quantum machine learning (2026)
3. Li, S., Kimura, Y., Sato, H., Fujita, M.: Parallelizing Quantum Simulation With Decision Diagrams . IEEE Transactions on Quantum Engineering **5**(01), 1–12 (Jan 2024). https://doi.org/10.1109/TQE.2024.3364546, `https://doi.ieeecomputersociety.org/10.1109/TQE.2024.3364546`
4. Pan, F., Gu, H., Kuang, L., Liu, B., Zhang, P.: Efficient quantum circuit simulation by tensor network methods on modern gpus **5**(4) (2024)
5. Prado, J., Rojas, I., Cuellar, M.P.G.: A grover-based approach to find the optimal solution in the "single piece remaining" puzzle. 2026 IEEE Conference on Artificial Intelligence (CAI) pp. 1829–1833 (2026)
6. Rezaei, A., Jaulmes, L., Bahna, M., Brown, O.T., Barbalace, A.: Low-level and numa-aware optimization for high-performance quantum simulation (2025)