GPU Accelerated Quantum Circuit Simulation using Tensor Contractions

Flemming Morsch

August 2024

1 Introduction

In recent times, machine learning methods such as transformer models gained a lot of media attention and more and more applications of such methods are being used in our every day. A big share of this success can be attributed to the steady development of Graphical Processing Units (GPUs) which allow for fast computation of these models. While, classical high performance computing becomes more and more refined, applications in quantum computing on real quantum computers are still difficult to implement due to accumulation of errors, decoherence as well as scalability issues [1]. Nevertheless, many researchers are working on the development of quantum algorithms which potentially improve the time complexity of corresponding classical algorithms with exponential speedup [2]. Therefore, many academic groups as well as commercial players implement these algorithms on classical computers and simulate the quantum algorithms [3]. One very promising algorithm is Grover's search algorithm for unstructured search problems such as finding the name of a person in a database given only the person's telephone number [4]. The current project investigates strategies to implement Grover's algorithm on GPUs and benchmarks the performance of different GPU-based algorithms against the CPU version.

2 Background

2.1 Quantum circuits

Quantum circuits and algorithms can be written with gates similar to gates used in classical computing albeit with some crucial differences. One very important fact is that due to its quantum mechanic properties, quantum computing has a probabilistic nature and whether a qubit has the value 0 or 1 is only known to the observer upon measurement. Quantum mechanically, we say that the wave function collapses when measuring. Thus, when simulating a quantum circuit, we need to encode this behaviour as well such that we get a vector of probabilities in the end over the possible states of the qubits [2].

2.2 Qubits and quantum gates

Like the name suggests qubits are quantum bits and corresponding to classical bits in classical computing. Qubits are represented in Dirac's bra-ket notation, where bra's represent row-vectors and ket's column-vectors. Therefore, correspondingly to classical bits, we can represent 0 and 1 as a ket or a column vector.

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Usually we work with more than one single qubit. We can write several qubits in the following way:

$$|000\rangle = |0\rangle \otimes |0\rangle \otimes |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

We use the Kronecker product to compute the vector representation of the 3 qubits. One important thing to notice here is that this state vector grows exponentially with the number of qubits.

The operations or gates we apply to the qubits are represented as matrices via matrix multiplication. Some of the most important gates for quantum computation are listed below.

Identity Gate (I):

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Hadamard Gate (H):

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

Pauli-X Gate (X):

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

Pauli-Y Gate (Y):

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

Pauli-Z Gate (Z):

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

As an example, we can apply the X (NOT) gate like this:

$$X |0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle$$

Similarly, we can apply gates to several qubits:

$$X \otimes X \otimes X |000\rangle = \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right) \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = |111\rangle$$

We can see that both the state vector and the operations on qubits grow exponentially with the number of qubits. This is one of the major obstacles when simulating quantum circuits on a classical computer. Thus, we must use methods which tackle this problem and reduce the memory footprint in order the simulate larger circuits.

2.3 Superposition

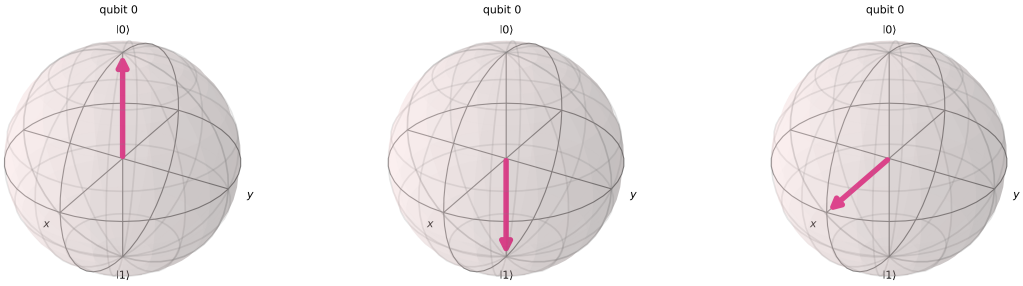
Another important concept in quantum computing which is different from classical computing is superposition. Whilst in classical computing a bit can either be 0 or 1, in quantum computing a qubit can be in a superposition of multiple states. We can write this state as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where $|\alpha|^2 + |\beta|^2 = 1$. This is due to the principles of quantum mechanics. We only know the state of a qubit upon measurement, where we say we collapse the wave function. Therefore, the probability of getting either 0 or 1 during the measurement depends on the factors α and β. Many quantum algorithms (such as Grover's search algorithm as well) make use of this property and start with qubits in superposition. Mathematically, we can apply the H gate to achieve this like in the example below for 3 qubits:

$$H \otimes H \otimes H\,|000\rangle = \left(\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \right) \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$= \frac{1}{\sqrt{8}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle)$$

The result is an equal superposition of all 8 possible states of these qubits. Comparing this to classical computing, we are representing the numbers from 0 to 7 and upon measurement each of these states has the same probability of $\frac{1}{8}$. As mentioned, this property of quantum mechanics is exploited in many algorithms to achieve better complexities compared to existing classical algorithms. To get a better intuition of this behaviour of qubits, one can visualise the qubits on the Bloch sphere.

In figure 1 we can see the a qubit in the states of $|0\rangle$, $|1\rangle$ and in equal superposition between the two. This state is written as $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and can be achieved by applying the H gate on a $|0\rangle$ qubit.



(a) (b) (c)

Figure 1: Qubit in a) $|0\rangle$, b) $|1\rangle$, and c) $|+\rangle$ state.

3 Grover's algorithm

Grover's algorithm is a quantum algorithm designed for searching an unsorted database or solving a search problem with N entries in $O(\sqrt{N})$ time, which is significantly faster than the best possible classical algorithm that requires $O(N)$ time. In the following the different steps of the algorithm are described, the quantum circuit is explained as well as methods for reformulating the algorithm using tensor contractions to simulate Grover's search on GPUs [4].

3.1 Steps of Grover's Algorithm

Grover's algorithm is used to find a specific item in an unsorted database. Suppose we have a function $f(x)$ which outputs 1 if x is the target item and 0 otherwise. The goal is to find the x for which $f(x) = 1$. In a quantum context, this can be achieved by starting at an equal superposition over all possible states (solutions) and then using Grover's algorithm amplify the amplitude of the solution state. This can be achieved with the quantum circuit in figure 2.

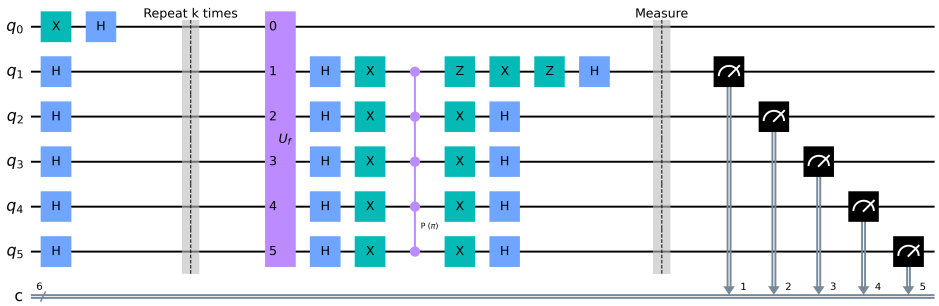


Figure 2: Quantum circuit of Grover's algorithm.

As can be seen, we start in an equal superposition where all qubits are in the $|+\rangle$ state. The 0th qubit is just called the answer qubit and is used with the oracle operator U_f but is not part of the superposition of possible solutions. Once we have applied the H gate to all n qubits, we alternate between the oracle which

knows the solution state and the diffusion operator which is shown as a full circuit. It encompasses a series of H and X gates followed by a multi-controlled Z gate (MCZ gate) where all but the last qubit are the control and the last qubit is the target qubit. Hence this gate applies a Z gate only to the state which contains only 1's and flips its sign like the single qubit Z gate: $Z|1\rangle = -|1\rangle$. Finally, some more single qubit gates are applied and this procedure is repeated k times and finally the qubits are all measured to find the amplified state representing the solution to the search problem.

The number of iterations k in Grover's algorithm where N is the number of states 2^n for n qubits and P the number of processors is given by:

$$k = \left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{P}} \right\rfloor \tag{1}$$

Now, let's describe the different steps in more detail with example calculations for $n = 3$ qubits:

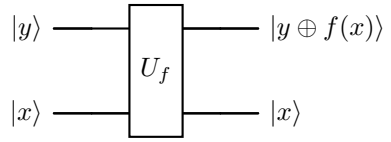
Initiate an equal superposition of n qubits: As described before we can apply the H gate to qubits in the $|0\rangle$ state to achieve this.

$$|\psi\rangle = H \otimes H \otimes H\,|000\rangle = \begin{pmatrix} 0.353553 \\ 0.353553 \\ 0.353553 \\ 0.353553 \\ 0.353553 \\ 0.353553 \\ 0.353553 \\ 0.353553 \end{pmatrix}$$

At this stage, each of the 8 solutions has an equal probability of being measured. By applying the oracle and the diffusion operator several times we would like to amplify the amplitudes of the marked solutions at the cost of the amplitudes of the other states.

Apply the phase oracle U_f:

The phase oracle U_f shown below knows the solution to the search query. It uses the XOR operator to flip the sign of the solution state.



Let's say our desired solution is the state 3 or in binary $|011\rangle$. Then after applying the oracle, we get the following:

$$U_f\,|\psi\rangle = \begin{pmatrix} 0.353553 \\ 0.353553 \\ 0.353553 \\ -0.353553 \\ 0.353553 \\ 0.353553 \\ 0.353553 \\ 0.353553 \end{pmatrix} = |\psi'\rangle$$

We can say that we marked the solution state and in the following we use the diffusion operator to amplify its amplitude.

Amplitude amplification: For convenience we just call the diffusion operator in figure 2 R_S which represents all gates which come after the oracle U_f. Then we get:

$$R_S \left|\psi'\right\rangle = \begin{pmatrix} 0.176777 \\ 0.176777 \\ 0.176777 \\ 0.883883 \\ 0.176777 \\ 0.176777 \\ 0.176777 \\ 0.176777 \end{pmatrix} = \left|\psi''\right\rangle$$

Then, if we compute k for 3 qubits we get $k = \left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{P}} \right\rfloor = 2$ assuming that we use $P = 1$. After 2 iterations, we then get the final state vector:

$$\left|\psi''_k\right\rangle = \begin{pmatrix} -0.088388 \\ -0.088388 \\ -0.088388 \\ 0.972272 \\ -0.088388 \\ -0.088388 \\ -0.088388 \\ -0.088388 \end{pmatrix}$$

We can see that the state of the solution is greatly amplified and we found the correct answer to our search query. Thus, on a quantum computer we got a high probability of measuring the correct state and one measurement would most likely suffice.

4 Methods

4.1 Implementing the quantum circuit

The previous section described the unstructured search problem and how to solve it using Grover's algorithm. We have also seen the biggest issue when trying to simulate quantum circuits: Scalability. With each qubit we add to the problem, we double the state vector and operations through gates get even more expensive as they grow in 2 dimensions. The memory footprint of this quantum circuit when diagonalising the gate operations using the tensor product will therefore increase exponentially in 2 dimensions which is not feasible. On the other hand, we need the exponentially increasing state vector to encode for the probabilities of the different states as well as entanglement between the qubits. One way of representing qubits on classical computers, while controlling the exponential increase in size, is to use matrix product states (MPS). However, as the focus of this project lies on the CUDA implementation and MPS require extensive development work, we keep the complete state vector in the current work. The gates however are kept as 2 by 2 matrices and do therefore not increase exponentially.

4.2 Tensor contractions

Tensor contraction is a generalisation of matrix multiplication in 3 or more dimensions. In our case we would like to contract the state vector of size 2^n with the different gates which are 2×2 matrices. The state vector thus needs to be seen as a tensor of size $2_0, 2_1, ...2_{n-1}$. Therefore, to apply a gate to each qubit we need to contract the gate and the state over each of the n dimensions of the state. Let's take an example as shown in figure 3 where we apply the Hadamard gate to 3 qubits. In many quantum algorithms, we would like to start out with an equal superposition of all $N = 2^n$ states given n qubits. In the example below we thus have 8 possible states and create the superposition by applying a Hadamard gate to each of the qubits as depicted.

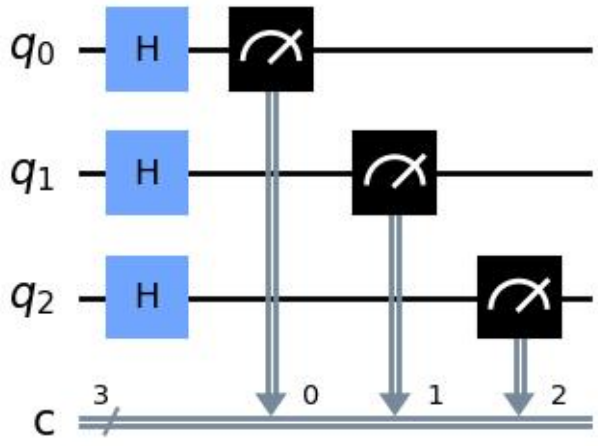


Figure 3: Quantum circuit of superposition of 3 qubits.

We can do the calculations for each of the 3 qubits as follows.
We first define the 2 tensors H and ψ:

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

$$\psi = \left[\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \right]$$

Next, we contract the 2 tensors over the 1st axis of H (indices: i,j) and the 0th axis of ψ (indices: k, l, m):

$$\psi'_{i,l,m} = \sum_{j,k} H_{i,j} \psi_{k,l,m} = \left[\begin{pmatrix} \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 \end{pmatrix} \right]$$

Continuing this procedure for the 1st and 2nd axes of ψ, we get the 2 remaining contractions:

$$\psi''_{i,k,m} = \sum_{j,l} H_{i,j} \psi'_{k,l,m} = \left[\begin{pmatrix} \frac{1}{\sqrt{4}} & 0 \\ \frac{1}{\sqrt{4}} & 0 \end{pmatrix}, \begin{pmatrix} \frac{1}{\sqrt{4}} & 0 \\ \frac{1}{\sqrt{4}} & 0 \end{pmatrix} \right]$$

$$\psi'''_{i,k,l} = \sum_{j,m} H_{i,j} \psi''_{k,l,m} = \left[\begin{pmatrix} \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} \\ \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} \end{pmatrix}, \begin{pmatrix} \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} \\ \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} \end{pmatrix} \right]$$

As we can see we need to do this procedure n times to apply a gate to all qubits and this operation is the core ingredient of our algorithm. Another critical observation is that in order to contract the 2 tensors we either need to reorder the axes every time or we need to compute the "jumps" we need to take to get the correct indices depending on the chosen axis. In the following parts, we will discuss how we can implement Grover's algorithm using tensor contractions on a GPU.

4.3 Tensor networks

One way of implementing quantum circuits on a classical computer is through tensor networks. These networks connect nodes representing the gates with edges representing the qubits. The key advantage of such networks is that we avoid the exponential increase in dimensionality of both the gates and the state (the qubits) as well. However, implementing tensor contractions for entangled states such as in Grover's algorithm requires many additional steps in order to keep the dimensionality low. In particular, we needed to employ matrix product states to represent entangled qubits which entails implementing singular value decomposition and then only taking the first k singular values to avoid too many dimensions during the computation [5, 6]. Since this project focuses optimising algorithms on GPUs the focus is set on CUDA programming rather than the complexities of quantum algorithms. We therefore use a hybrid between diagonalised, and exponentially growing computations in both the state and the gates and matrix product states. The algorithms presented in the current work keep the dimensions of the gates at 2 by 2, thus avoiding an exponential increase. However, the full state vector is represented as shown in the previous section and thus grows exponentially. Despite this disadvantage, we can simulate the exact state with this procedure and it is possible to implement a much simpler algorithm and focus more on techniques to optimise the computations on the GPU using CUDA.

4.4 Baseline Algorithms

4.4.1 OpenMP version

Let us now implement the base algorithms for the different operations in Grover's algorithm and optimise them for CUDA devices. The base version of the tensor contraction algorithm to contract a 2 by 2 tensor with a tensor of the size $2_0, 2_1, ...2_{n-1}$ can be implemented as follows:

```
void contract_tensor(const Complex* state,
                     const Complex gate[2][2],
                     int qubit,
                     Complex* new_state,
                     const int* shape, int n) {
    int total_elements = (int)pow(2, n);

    // Zero out new_state
    zeroOutArray(new_state, total_elements);

    // Iterate over all possible indices of the state tensor
    #pragma omp parallel for
    for (int idx = 0; idx < total_elements; ++idx) {
        int new_idx[n];
        int old_idx[n];
        int temp = idx;

        // Compute the multi-dimensional index
        for (int i = n - 1; i >= 0; --i) {
            new_idx[i] = temp % shape[i];
            temp /= shape[i];
        }

        // Perform the tensor contraction for the specified qubit
        for (int j = 0; j < 2; ++j) {
            // Copy new_idx to old_idx
            for (int i = 0; i < n; ++i) {
                old_idx[i] = new_idx[i];
```

```
29              }
30              old_idx[qubit] = j;
31
32              // Compute the linear index for old_idx
33              int old_linear_idx = 0;
34              int factor = 1;
35              for (int i = n - 1; i >= 0; --i) {
36                  old_linear_idx += old_idx[i] * factor;
37                  factor *= shape[i];
38              }
39
40              new_state[idx] += gate[new_idx[qubit]][j] * state[
                    old_linear_idx];
41          }
42      }
43  }
```

Algorithm 1: OpenMP implementation of tensor contraction

Algorithm 1, shows the basic implementation of the discussed tensor contraction. We start off with the state tensor with is implemented as an array. This is done since its dimensions will grow with each additional qubit and the current algorithm keeping track of the indices of the values in this array which need to be multiplied with the values in the gate matrix. Due to this complexion, the main difficulty of this method is to compute the indices needed for matrix multiplication depending on the chosen axis for the state vector without adding more and more for loops with many new indices. Thus, we need to somehow keep track of the chosen indices in a data structure using the information about the shape of the state tensor without constantly reshaping the state tensor which would be difficult and time consuming.

The **new_idx** array keeps track of the new multidimensional indices of the tensors. This array is later also used to choose the positions in the gate matrix.

The **old_idx** array on the other hand is a copy of the **new_idx** array with the difference that we then choose the axis **qubit** for the tensor contraction. Then based on this array, we compute the **old_linear_idx** for the linear state tensor and perform matrix multiplication. Let's compute this for the 0th index for our 3 qubit example and apply the H gate to the 0th qubit (or axis):

We have $idx = 0$ and compute the **new_idx** array:

$$[0, 0, 0]$$

Next, for each $j = 0, 1$ we compute the **old_idx** array and the **old_linear_idx** which defines the position in the current state we need to access for the matrix multiplication.

For $j = 0$ we get:

$$\textbf{old\_idx} = [0, 0, 0] \quad \textbf{old\_linear\_idx} = 0 \quad value = \frac{1}{\sqrt{2}}$$

For $j = 1$ we get:

$$\textbf{old\_idx} = [1, 0, 0] \quad \textbf{old\_linear\_idx} = 4 \quad value = 0$$

Summing the 2 values indeed gives the correct result of $\frac{1}{\sqrt{2}}$ at that position in the new state as shown in the previous section.

Doing this procedure for all N indices will compute the complete new state and successfully apply the gate to the qubit represented in the state vector.

Now, since we want to optimise this CPU version, we can use OpenMP to parallelise the for loop and compute the N values in parallel.

We also need to implement the oracle and multi-controlled gates. This is actually rather straight forward when dealing with the full state vector since we have all possible states available and both the oracle and the multi-controlled Z gate (MCZ gate) are essentially phase flips which means we can simply multiply with -1 at the desired index. The oracle thus flips the sign at the marked position of the solution and the MCZ gate flips the sign of the state with only 1s (the last entry in the state vector array).

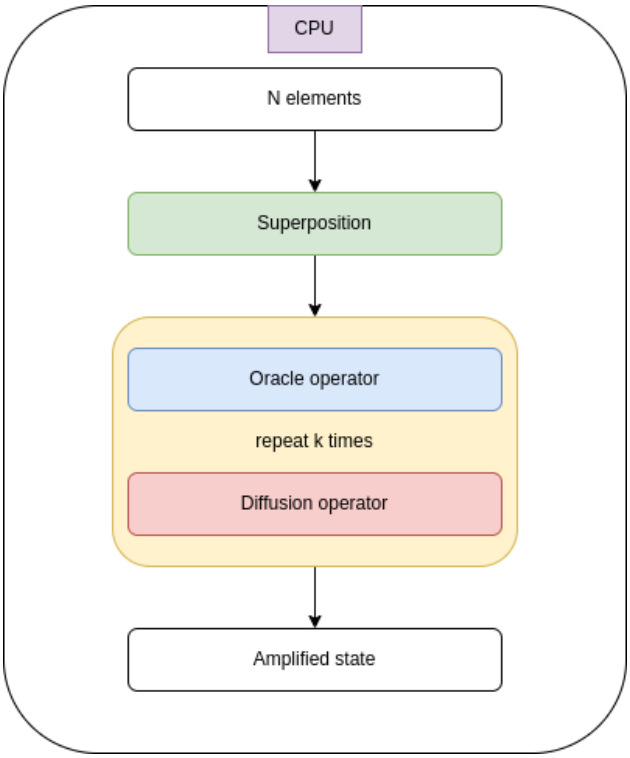


Figure 4: Overview of Grover's algorithm implemented on the CPU.

Figure 4 gives an overview of the implementation of Grover's algorithm on the CPU using OpenMP. As also shown in the code of the contraction algorithm, which is the main sub-routine in Grover's algorithm, this version simply parallelises the computation over the N elements in the state vector and uses the updated state vector in the next computation step. This higher level overview of the full algorithm (see figure 2) highlights that we can only compute 1 element per core and the remaining elements need to wait until a new thread is available for computation.

4.4.2 CUDA Baseline

To get a baseline for a GPU version, we essentially also parallelise of the N computations like in the OMP version. The main difference here is that we need to initiate array for the indices and use an offset to compute them for each qubit and each $idx \in N$. Since we have many threads available on the GPU, one can expect a substantial speedup compared to the CPU version.

```
__global__ void contract_tensor_baseline(
        const Complex* state,
        const Complex* gate,
        int qubit,
        Complex* new_state,
        const int* shape,
        int* new_idx,
        int* old_idx,
        const int n,
        const long long int N
    ) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int offset = idx * n;
    if (idx < N) {

        int temp = idx;

        // Compute the multi-dimensional index
        for (int i = n - 1; i >= 0; --i) {
            new_idx[offset+i] = temp % shape[i];
            temp /= shape[i];
        }

        // Perform the tensor contraction for the specified qubit
        for (int j = 0; j < 2; ++j) {
            // Copy new_idx to old_idx
            for (int i = 0; i < n; ++i) {
                old_idx[offset+i] = new_idx[offset+i];
            }
            old_idx[offset+qubit] = j;

            // Compute the linear index for old_idx
            int old_linear_idx = 0;
            int factor = 1;
            for (int i = n - 1; i >= 0; --i) {
                old_linear_idx += old_idx[offset+i] * factor;
                factor *= shape[i];
            }
            AddComplex(&new_state[idx], cuCmul(gate[new_idx[offset+qubit]
               * 2 + j], state[old_linear_idx]));
            }
        }
    }
```

Algorithm 2: CUDA baseline algorithm

Similarly, figure 5 shows the overview of the baseline algorithm implemented on the GPU. As described with the code, this version can be seen as analogous to the CPU version where each thread is doing the computation of 1 element and we use the **atomicAdd** operation in CUDA to sum elements to get the updated state vector.

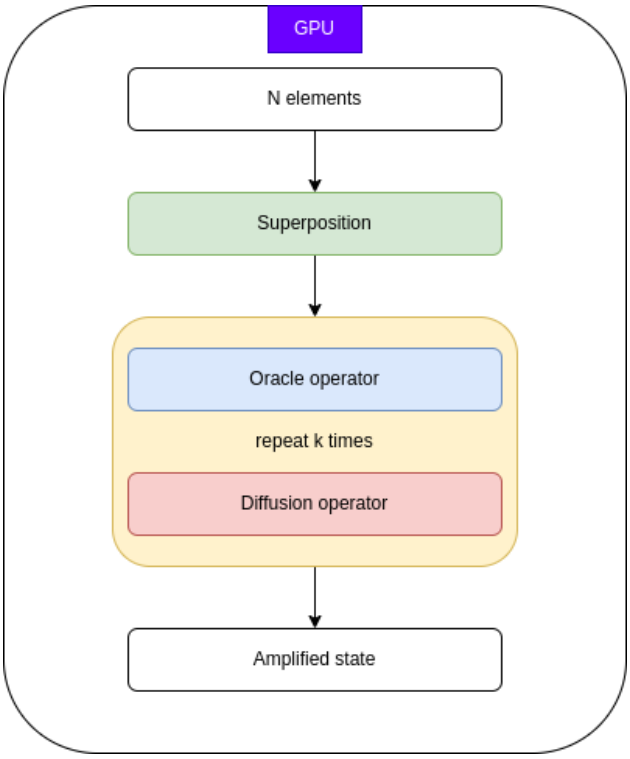


Figure 5: Overview of Grover's algorithm implemented on the GPU as a baseline version.

4.5 Optimised Algorithms

4.5.1 CUDA version 1

One major drawback of the baseline algorithms shown so far is the large state vector which needs to be duplicated for the current and the new state. Moreover, these arrays need to be allocated and updated after each tensor contraction which is not optimal. We could of course slightly improve these baseline algorithms by swapping the pointers between the state and new state arrays but there is also a better method available on the GPU. We can utilise shared memory. Shared memory is a type of memory that is accessible by all threads within the same block and is located on the chip within the streaming multiprocessor (SM). This leads to much faster memory access and better performance compared to accessing global memory. Additionally, the size of the state vector is exponential in qubits, meaning we have an array of size 2^n elements for n qubits. If we therefore control the number of elements within a block (max. 1024 threads per block), we can align the number of threads per block and the number of blocks nicely with the length of the state vector. However, the shared memory is limited and its size varies per GPU (see link). The maximum value for CUDA compute capability 9.0 devices is 227 KB per thread block. We must therefore write the algorithm in a way that not too many values need to be stored in the shared memory in order to make use of shared memory to do computations over the threads of each block. At this stage we can make use of the architecture of Grover's algorithm. Using equation 1 we know that we can divide the space of N elements among several processors and let each of them compute one of the chunks. The only thing we need to adjust is that we need to tell the oracle operator the chunk it needs to operate on. This means that if we take the space of $n = 3 \quad N = 8$ as an example we could divide this space among 2 processors and search in parallel by keeping track of the chunk which contains the solution state:

$$Chunk_0 = |000\rangle + |001\rangle + |010\rangle + |011\rangle \quad Chunk_1 = |100\rangle + |101\rangle + |110\rangle + |111\rangle$$

And since we can normalise each chunk, and would like to state in the $|+\rangle$ state, we can simply initiate 2 arrays with each $\frac{N}{2}$ elements which is the same as using 2 times $n = 2$ qubits in superposition:

$$Chunk_0 = Chunk_1 = H \otimes H \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Then, during computation of Grover's algorithm only the chunk in which the oracle marks the solution will be amplified and all other chunks remain in equal superposition. This basically means that we need to initiate 1 chunk of N/P per block so we ensure we have enough threads and shared memory available so we can store intermediate results in it. This parallel search procedure is also shown in the code 3 as well as in the overview of the algorithm 6. Using this parallel search we can run this algorithm over the full space of N elements but we compute the indices based on the number of qubits per chunk which define the chunk size. Then during the computation of the indices, we compute the offsets for the elements involved in the tensor contraction in only within the range of the chunk size and hence compute separate chunks in parallel on the same GPU. This can be seen in the code in lines 13 and 18, where we compute the state per chunk by using the modulus operator. It is also worth mentioning that the elements are written and read in a coalesced fashion. Nevertheless, due to the fact that we need to use an offset to use the correct indices, we sometimes need to introduce a "jump" which may impact performance. This will be analysed and discussed further in the results.

```
__global__ void contract_tensor(
        Complex* state,
        const Complex* gate,
        int qubit,
        int* new_idx,
        int* old_idx,
        const int n,
        const long long int N,
) {
    extern __shared__ Complex shared_mem[]; // Use shared memory
    int idx = blockDim.x * blockIdx.x + threadIdx.x;

    int chunk_size = pow(2, n);

    if (idx < N)) {

        int offset = idx * n;
        int temp = idx % chunk_size;

        // Compute the multi-dimensional index
        for (int i = n - 1; i >= 0; --i) {
            new_idx[offset + i] = temp % 2;
            temp /= 2;
        }

        // Copy new_idx to old_idx
        for (int i = 0; i < n; ++i) {
            old_idx[offset + i] = new_idx[offset + i];
        }

        // Compute the two values for j = 0 and j = 1 and store in shared
           memory
        for (int j = 0; j < 2; ++j) {
            old_idx[offset + qubit] = j;

            // Compute the linear index for old_idx
            int old_linear_idx = 0;
            int factor = 1;
            for (int i = n - 1; i >= 0; --i) {
                old_linear_idx += old_idx[offset + i] * factor;
                factor *= 2;
            }

            // needed to translate back to the full state array!!!
            if (idx >= chunk_size) {
                old_linear_idx += (idx / chunk_size) * chunk_size;
            }

            // Store the result in shared memory
            shared_mem[2*(idx % chunk_size) + j] = cuCmul(gate[new_idx[
               offset + qubit] * 2 + j], state[old_linear_idx]);
```

```
50          }
51          __syncthreads();
52          state[idx] = cuCadd(shared_mem[2*(idx % chunk_size) + 1],
                shared_mem[2*(idx % chunk_size)]);
53      }
54  }
```

Algorithm 3: CUDA version 1 algorithm

A big concern of this simple hybrid implementation of a quantum algorithm is the large memory footprint. Here, the parallel search approach allows for further optimisation. We can run several parallel searches sequentially. This let's us choose the number of chunks computed per group. Additionally, we can initialise the superposition directly on the GPU since the important step is the interaction of the oracle operator which can be seen as a function knowing the solution of the database query. Then, we only need to keep track of the amplified chunk and only copy this array from the GPU back to the host reducing the memory footprint. Computationally, this is also very interesting as it allows us to fine-tune the block size, and the size of the group of chunks in each parallel search as this affects the performance through among others general memory footprint, shared memory utilisation and memory access.

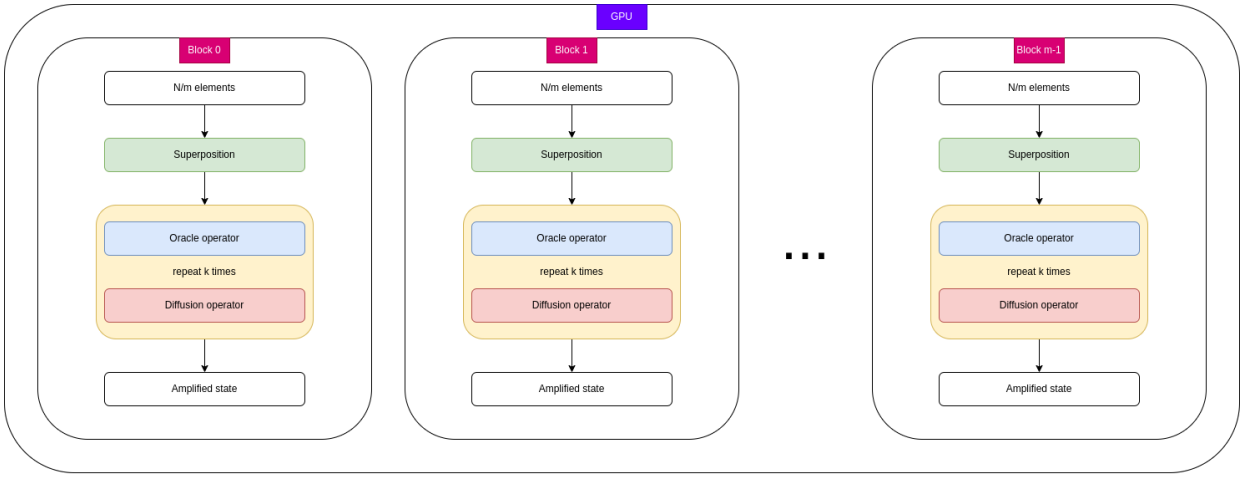


Figure 6: Overview of Grover's algorithm implemented on the GPU using parallel search.

4.5.2 CUDA version 2

The second optimisation of the code improves the computation of the indices. Each time the tensor contraction algorithm v1 is run, it computes and overwrites the old and new indices again which leads to some overhead. We can therefore take this part of the code and run it as a separate kernel and only once in the beginning. Additionally, we only need to compute the indices for the set chunk size, as we are still running a parallel search version of Grover's algorithm, reducing the needed memory for the indices (algorithm 4).

```
__global__ void compute_idx(
        int qubit,
        int* new_idx,
        int* old_idx,
        const int n,
        const long long int N,
        int* old_linear_idxs
    ) {
    extern __shared__ int shared_memory[]; // Use shared memory

    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int offset = idx * n;
    int offset2 = qubit*2*N;

    if (idx < N) {
        int temp = idx;

        // Compute the multi-dimensional index
        for (int i = n - 1; i >= 0; --i) {
            new_idx[offset + i] = temp % 2;
            temp /= 2;
        }

        // Copy new_idx to old_idx
        for (int i = 0; i < n; ++i) {
            old_idx[offset + i] = new_idx[offset + i];
        }

        // Compute the two values for j = 0 and j = 1 and store in shared
           memory
        for (int j = 0; j < 2; ++j) {
            old_idx[offset + qubit] = j;

            // Compute the linear index for old_idx
            int old_linear_idx = 0;
            int factor = 1;
            for (int i = n - 1; i >= 0; --i) {
                old_linear_idx += old_idx[offset + i] * factor;
                factor *= 2;
            }
            shared_memory[2*idx + j] = old_linear_idx;
        }
        old_linear_idxs[2*idx + offset2] = shared_memory[2*idx];
        old_linear_idxs[2*idx + 1 + offset2] = shared_memory[2*idx+1];
    }
}

__global__ void contract_tensor(
        Complex* state,
        const Complex* gate,
        int qubit,
```

```
51          int* new_idx,
52          const int n,
53          const long long int N,
54          int* old_linear_idxs
55
56 ) {
57     extern __shared__ Complex shared_mem[]; // Use shared memory
58     int idx = blockDim.x * blockIdx.x + threadIdx.x;
59
60
61     int chunk_size = pow(2, n);
62
63     if (idx < N) {
64
65         int offset = (idx % chunk_size) * n;
66
67         // Compute the two values for j = 0 and j = 1 and store in shared
               memory
68         for (int j = 0; j < 2; ++j) {
69             // needed to translate back to the full state array!!!
70             int old_linear_idx = old_linear_idxs[2*(idx % chunk_size) + j
                   + qubit*2*chunk_size];
71             old_linear_idx += (idx / chunk_size) * chunk_size;
72
73             // Store the result in shared memory
74             if (j == 0) {
75                 Complex val = cuCmul(gate[new_idx[offset + qubit] * 2 + j
                       ], state[old_linear_idx]);
76                 shared_mem[idx % chunk_size] = val;
77
78             } else {
79                 Complex val = cuCmul(gate[new_idx[offset + qubit] * 2 + j
                       ], state[old_linear_idx]);
80                 shared_mem[idx % chunk_size] = cuCadd(shared_mem[idx %
                       chunk_size], val);
81             }
82
83         }
84         state[idx] = shared_mem[idx % chunk_size];
85     }
86 }
```

Algorithm 4: CUDA version 2 algorithm with pre-computed indices

4.5.3 CUDA version 3

Version 3 of the CUDA implementation is similar to version 2 with the distinction that 2 shared memory buffers are used to have one buffer for each of the 2 j's (algorithm 5). The kernel to compute the indices for the tensor contractions is the same as in version 2. Another important distinction of this version is the MCZ gate kernel which is now applied at once on all chunks of N in parallel, whilst before the flip was applied to each chunk sequentially.

```
__global__ void compute_idx(
        int qubit,
        int* new_idx,
        int* old_idx,
        const int n,
        const long long int N,
        int* old_linear_idxs
    ) {
    extern __shared__ int shared_memory[]; // Use shared memory

    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int offset = idx * n;
    int offset2 = qubit*2*N;

    if (idx < N) {
        int temp = idx;

        // Compute the multi-dimensional index
        for (int i = n - 1; i >= 0; --i) {
            new_idx[offset + i] = temp % 2;
            temp /= 2;
        }

        // Copy new_idx to old_idx
        for (int i = 0; i < n; ++i) {
            old_idx[offset + i] = new_idx[offset + i];
        }

        // Compute the two values for j = 0 and j = 1 and store in shared
           memory
        for (int j = 0; j < 2; ++j) {
            old_idx[offset + qubit] = j;

            // Compute the linear index for old_idx
            int old_linear_idx = 0;
            int factor = 1;
            for (int i = n - 1; i >= 0; --i) {
                old_linear_idx += old_idx[offset + i] * factor;
                factor *= 2;
            }
            shared_memory[2*idx + j] = old_linear_idx;
        }
        old_linear_idxs[2*idx + offset2] = shared_memory[2*idx];
        old_linear_idxs[2*idx + 1 + offset2] = shared_memory[2*idx+1];
    }
}

__global__ void contract_tensor(
        Complex* state,
        const Complex* gate,
        int qubit,
```

```
51          int* new_idx,
52          const int n,
53          const long long int N,
54          int* old_linear_idxs,
55          const int chunk_size
56
57  ) {
58      extern __shared__ Complex shared_mem[];
59      int idx = blockDim.x * blockIdx.x + threadIdx.x;
60      if (idx < N) {
61
62          int offset = threadIdx.x * n;
63          int base_idx = (idx / chunk_size) * chunk_size;
64
65          Complex* buffer1 = shared_mem;
66          Complex* buffer2 = shared_mem + chunk_size;
67
68          // Compute the two values for j = 0 and j = 1 and store in shared
               memory
69          for (int j = 0; j < 2; ++j) {
70
71              // needed to translate back to the full state array
72              int old_linear_idx = old_linear_idxs[2*threadIdx.x + j + qubit
                    *2*chunk_size];
73              old_linear_idx += base_idx;
74
75              if(j==0){
76                  buffer1[threadIdx.x] = cuCmul(gate[new_idx[offset + qubit]
                        * 2 + j], state[old_linear_idx]);
77
78              } else {
79                  buffer2[threadIdx.x] = cuCmul(gate[new_idx[offset + qubit]
                        * 2 + j], state[old_linear_idx]);
80              }
81          }
82          state[idx] = cuCadd(buffer1[threadIdx.x], buffer2[threadIdx.x]);
83      }
84  }
85
86  __global__ void applyPhaseFlipParallel(Complex* state, const long long int
        N, const int N_chunk) {
87      int idx = blockDim.x * blockIdx.x + threadIdx.x;
88      if (idx < N) {
89          if ((idx % N_chunk) == N_chunk - 1) {
90              state[idx] = cuCmul(state[idx], make_cuDoubleComplex(-1.0,
                    0.0));
91          }
92      }
93  }
```

Algorithm 5: CUDA version 3 algorithm with pre-computed indices

4.5.4 CUDA versions with 2 GPUs

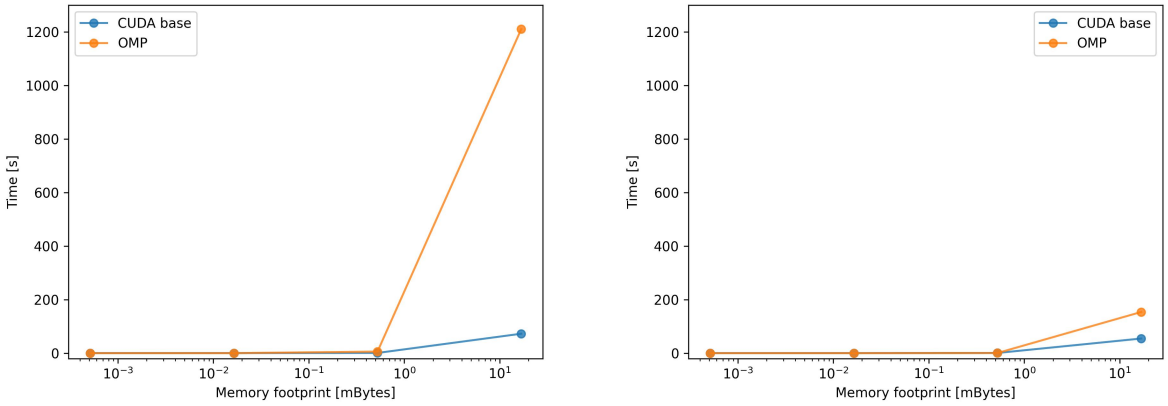
The CUDA versions 1, 2 and 3 have also be implemented on 2 GPUs which allows for dividing the different chunks among both GPUs for faster computation. The kernels are the same as in the corresponding single GPU versions.

In the following sections we investigate how Grover's algorithm using tensor contractions performs on the CPU, 1 and 2 GPUs and how we can further optimise this quantum algorithm for unstructured search.

5 Results and Discussion

5.1 Initial experiments with base versions

Initially, a baseline needed to be established for the OMP and baseline CUDA versions of Grover's algortihm. Both algorithms were tested on 1 H100 GPU and the Grace Hopper GPU (gracy). Figure 7a shows the performance on the H100 GPU for n=5, 10, 15, 20 qubits resulting in a memory footprint of $2^n \cdot 16$ bytes since we are dealing with complex double data types. As expected the CPU version which is using all 64 threads shows much a higher runtime of around 1200 seconds for 20 qubits compared to only 72 seconds for the CUDA baseline. The Gracy chip on the other hand shows results which are much closer to each other with values of 154 and 54 seconds for the OMP and CUDA versions respectively. Also here the CPU was using all 72 threads available on Gracy. An explanation for this result is that the problem is memory bound and that the Gracy chip as a higher bandwidth of 4TB/s compared to 3.35TB/s for the H100.



(a) (b)

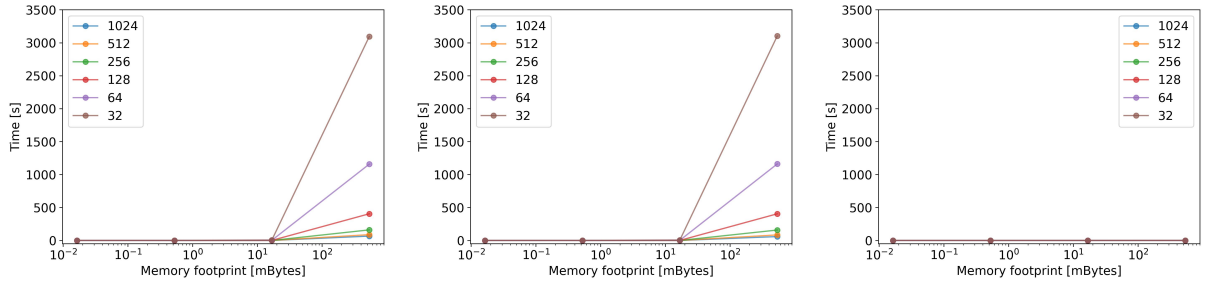
Figure 7: Speedup of OMP and CUDA base versions. a) on 1 H100 GPU. b) On Grace Hopper architecture (Gracy).

5.2 CUDA optimisations

After setting the baseline for the GPU, the goal is to improve the algorithm by systematically analysing and optimising the different kernels introduced in the previous sections. The 3 optimised algorithms all use the parallel search approach in order to make use of shared memory for intermediate results. In fact we need to use parallel search and chunk the N values into smaller chunks to make 1 chunk fit into a block. This is because we need to be able to do tensor contractions on the full state which increases exponentially as a power of 2. We therefore need to cut N into enough chunks such that each chunk is maximally 1024 values and fits in a block. Since we are dealing with powers of 2, the values fit nicely and we increase the grid size

accordingly. Moreover, since the state increases exponentially, we can also compute a part of the state as a second parallel group. For example a state of size $N = 2^{25}$ can be computed by running 2 parallel searches of $N_{group0} = N_{group1} = 2^{24}$. Then within each group we can determine the number of chunks we want which must fit into 1 block of 1024 threads.

Figures 8a, 8b and 8c show the speedup plots of the 3 different optimisations of the algorithm for different block sizes. The problem sizes of 5, 10, 15, 20 and 25 qubits were selected. All were computed using 1 group with the full state vector to allow for a fair comparison since the number of groups also influences the number of rounds k (see equation 1). For all 3 versions higher block sizes also increase the performance of the algorithms. Algorithm v1 shows as expected the poorest results while version 2 is only slightly better, even though v2 pre-computes the indices needed for the tensor contraction while v1 computes them within the kernel. For the last data point, equivalent to a problem size of $16 \cdot 2^{25}$ mBytes, version 1 runs in 66 sec and v2 in 59 sec. Version 3 on the other hand obtains runtimes of below 1 sec for all shown problem sizes.



(a) (b) (c)

Figure 8: Speedup of optimised CUDA versions. Versions v1, v2, v3 versions were tested. a) shows the speedup for CUDA version 1. b) shows the speedup for CUDA version 2. c) shows the speedup for CUDA version 3.

5.3 Differences between the three optimisations

Before analysing the different algorithms further it is worth mentioning the key differences of the 3 algorithms. Version 1 introduces shared memory which makes the extra array for the updated state from the base version obsolete. Shared memory is also very fast, as it is located on the chip, has low latency and higher bandwidth compared to global memory. The downside is its size. However since the algorithm chunks up the large state into chunks of size 1024, 512, 256, 128, 64 or 32 and we only need 2 times this in shared memory per block, it is within the limits. Loading data into shared memory and accessing it multiple times also reduces redundant memory access which is expected to be quite high as we are starting from a superposition of equal values for each value in the initial state array. Additionally, the state vector is directly initialised on the GPU since the algorithms initialises a superposition of states and no copying from the host is needed then. Version 2 introduces the major difference that it splits the tensor contraction kernel and separates the computation of the indices for the tensor contraction from the computation of the values in the state array. Version 3 is further optimising the algorithm by parallelising the the phase flip kernel needed to apply a multi-controlled Z gate to each of the chunks computed in parallel. The first 2 versions were running this kernel separately for each chunk. To investigate this behaviour Nvidia Nsight System is used to compare the performance of versions 2 and 3 in figure 9. Looking at the percentages of the executed kernels, we can see that version 2 mostly calls the phase flip kernel, while version 3 mostly calls the tensor contraction kernel. Hence even though the runtime of the single phase flip kernel is very short (ca. 144 µs), it creates an overhead, especially when many small chunks are computed in parallel since the phase flip kernel must be applied to each of them. The parallel version of it simply applies the MCZ gate to each chunk in the array in one kernel execution and is therefor much more efficient.

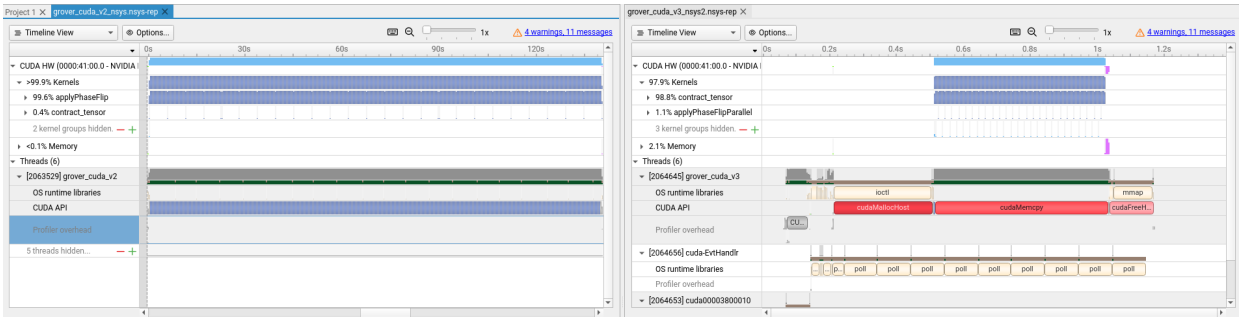


Figure 9: Nvidia Nsight Systems analysis of version 2 (left) and version 3 (right), The problem size was 25 qubits and a block size of 1024 was used.

5.4 Grid search to find optimal parameters

As described before, there are several parameters to be tuned for each of the 3 versions. Each of these algorithms takes the following arguments as input:

n:
The number of qubits. This problem size translates to a state vector of size $N = 2^n$.

markedState:
The marked state is the position of the solution state we give to the oracle operator. It must be between 0 and N-1.

Number of chunks per group:
As described in figure 6, Grover's algorithm can run several chunks of the whole state vector in parallel. And this needs to be done to make sure no chunk is larger than 1024 to fit into 1 block since we make use of the shared memory.

Number of qubits per group:
This defines the size of the state vector per group with $N_{group} = 2^{n\_qubits\_per\_group}$ and can be used to optimise for memory management as a smaller number of qubits per group allows for a smaller array to be copied to the host holding the solution (see algorithm implementation).

Therefore, depending on the number of qubits per group and n, we need to choose the number of chunks per group such that we get a chunk size of maximally 1024.

In the computations shown, we choose values which are a multiple of 32 for N_{chunk} since this also defines the block size. The grid size on the other hand is defined by the number of chunks per group. For 2 GPU versions, the number of qubits per group must always be at least 1 lower than n since the work is divided among 2 devices (see appendix for examples).

So far, the number of groups was always 1 and the problem size was also small enough to fit into the GPU memory as 1 group. However, to compute larger problems, running the search in several groups of chunks is needed. The figures 10 and 11 show experiments to find optimal parameter settings for the 3 optimised versions. In both figures, the left column represents version 1, the middle column version 2 and the right column version 3. All 3 algorithms were executed for a problem size of 25 qubits. The number of chunks per group was adjusted to match a block size of 1024 - 32 respectively. Additionally, the number of qubits per group was also changed ranging from 25 down to 10. This equals 96 combinations of parameters per algorithm which were tested. The figures 10 a) - f) show the number of chunks per group versus the number of qubits per group. The 3 upper plots show all data points, while the lower 3 only show data points with runtimes of maximally 10 seconds. As expected, the runtime improves from v1 to v3 and many different

parameter combinations give good results, and more groups seem to improve performance. The figures 10 g) - i) and 11 a) - c) show the runtime as a function of the number of qubits per group. The figures 10 g) - i) again show all data points while the latter 3 show data points with runtimes faster than 10 sec. In v1 block sizes of 128 and 256 give the best results in combination with 15 - 18 qubits per group. In versions v2 and v3 a block size of 1024 dominates. However, also for these versions the performance peaks with the lowest runtime at 18 qubits per group. Small block sizes of 64 and 32 on the other hand give worse performance indicated by longer runtimes. Similar results are observed in 11 d) - i) where the first 3 plots again show all data points and the lower 3 plots show data points with runtimes faster than 10 sec. Here, the runtime is plotted against the number of chunks per group. The optimal value for all 3 versions lies around $2^8 = 256$ chunks per group indicating that a block size of 1024 combined with 256 chunks per group give faster runtimes.

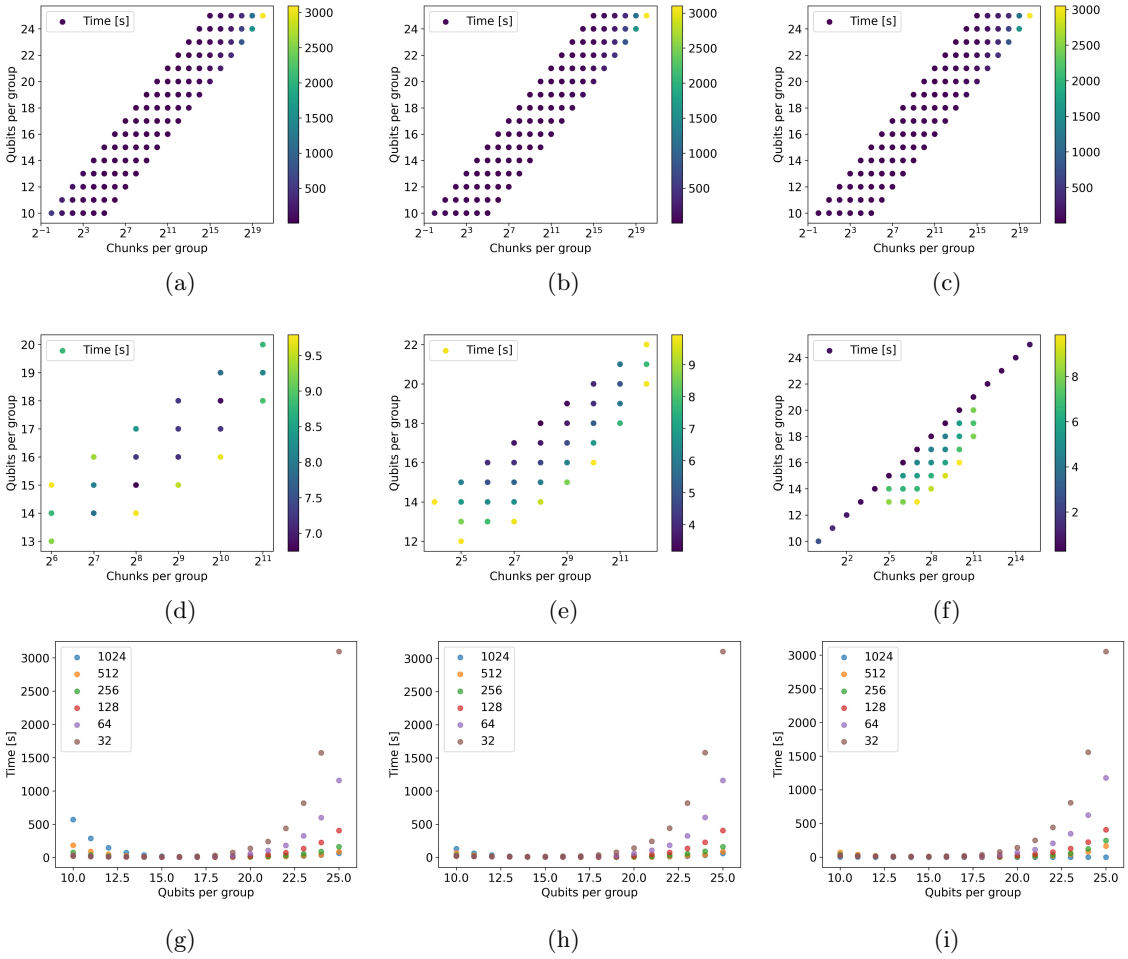


Figure 10: Grid search results of CUDA versions v1 (left column), v2 (middle column) and v3 (right column). a) - f) show the number of chunks per group versus the number of qubits per group. The 3 upper plots show all data points, while the lower 3 only show data points with runtimes of maximally 10 seconds. g) - i) show the runtime as a function of the number of qubits per group for all data points. Different block sizes were chosen from 1024 - 32. All results were obtained on the H100 GPU.

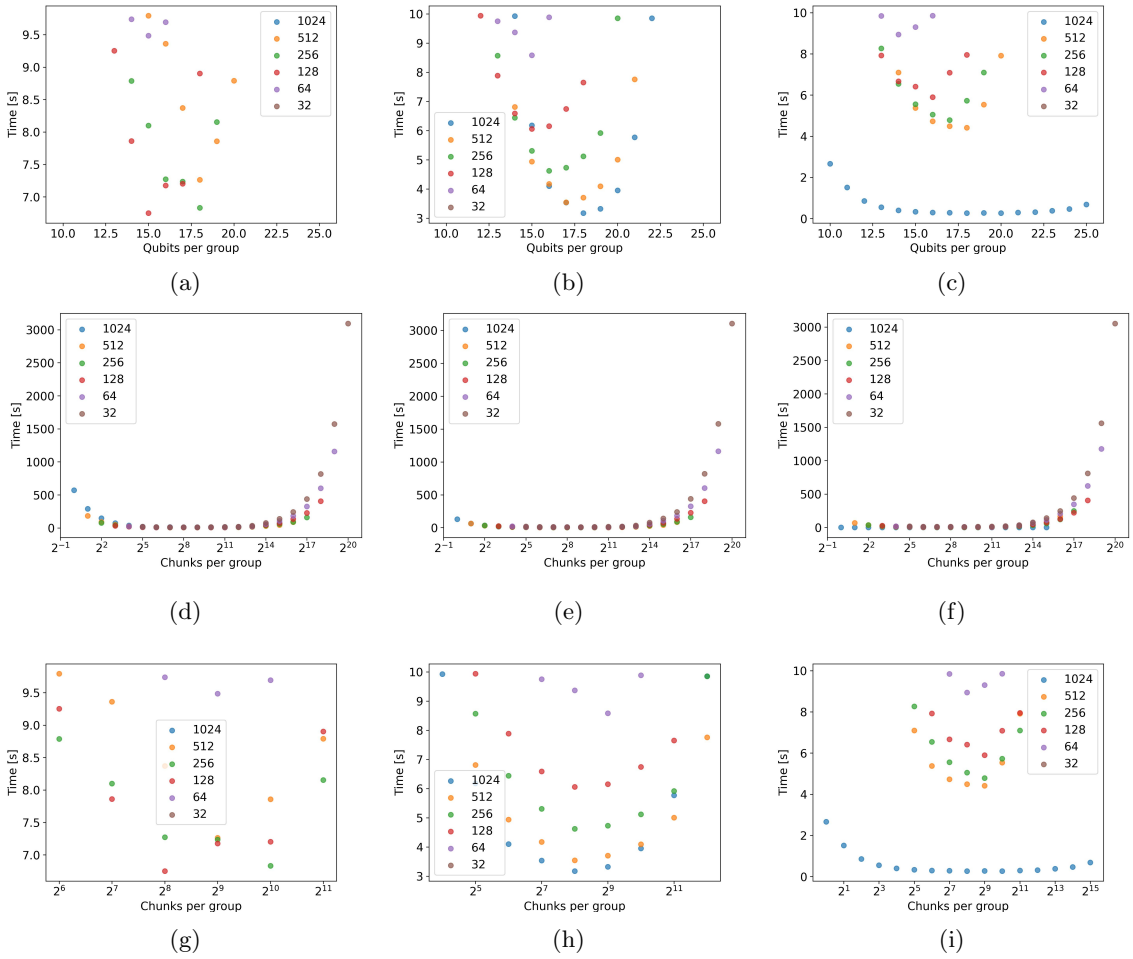


Figure 11: Grid search results of CUDA versions v1 (left column), v2 (middle column) and v3 (right column). a) - c) show the runtime as a function of the number of qubits per group for data points with runtimes faster than 10 sec. d) - i) show the number of chunks per group versus the runtime for all 3 algorithms. d) - f) show all data points, while g) - i) only show data points with runtimes faster than 10 sec. Different block sizes were chosen from 1024 - 32. All results were obtained on the H100 GPU.

5.5 Runtime analysis of CUDA version 3

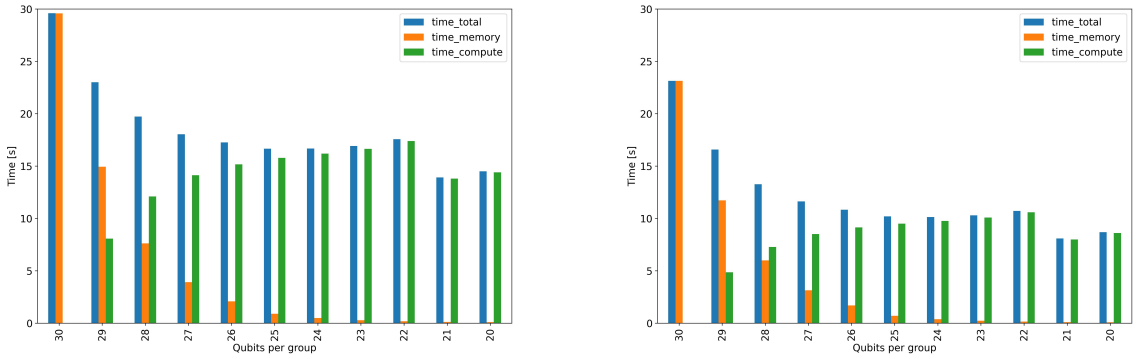
Version 3 is the most optimised and also fastest algorithm. It is therefore interesting to compute larger problem sizes and also profile the application and compare it to the other versions. As discussed in the previous section, it can be beneficial, and for larger problem sizes necessary, to run the algorithm in several groups of parallel Grover searches. Therefore, we now analyse the runtime of version 3 for a problem size of $N = 2^{30}$, thus 30 qubits. This equals approximately 17 gigaBytes and could be computed in 1 group or several. Figure 12 shows the runtime analysis of version 3. The x axis shows the number of qubits per group and the y axis the runtimes. A distinction is made between time needed for memory transfers and allocation and computation times. The different block sizes ranging from 1024-32 are shown in plots a)-f) respectively. As expected, the memory times are larger with more qubits per group since then the problem size per group is also large and it decreases for each block size with decreasing number of qubits per group. Thus, for smaller problem sizes in each group, the compute time dominates over the time needed for memory

operations and the total runtime decreases as well. The best total runtimes are found for block sizes of 64 and 32 as well as 20-25 qubits per group. In this way, we can optimise the algorithm to also search larger state vectors with Grover's algorithm. Another important factor is the number of rounds the algorithm needs to run in order to amplify the solution state. In equation 1 k is dependent on the problem size N, which is the chunk's problem size, as well as P the number of processors or total number of chunks in all groups. Therefore, having many smaller groups can also be used to reduce k and may reduce the runtime. The best overall parameter settings for 30 qubits were a block or chunk size of 64 and 21 qubits per group which gives a small k value of 6 rounds of Grover's algorithm. These settings are used later on and marked as **v3 best**.

5.6 Profiling of tensor contraction kernels

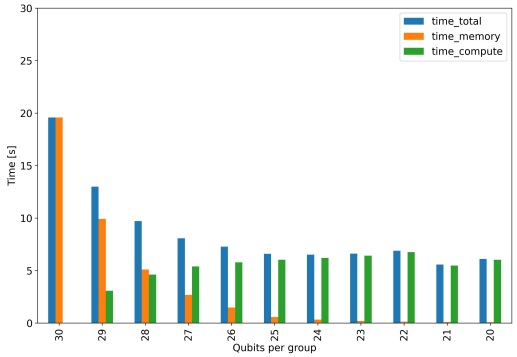
To further investigate the effect of different parameter settings on the performance of version 3, the algorithm was profiled with a focus on the tensor contraction kernel. A compute and memory analysis was conducted using Nsight Compute. In the current implementation of Grover's algorithm the state vector grows exponentially with the number of qubits. Additionally, the different kernels in the algorithm run many smaller computations and need to access shared and global memory often. It was therefore expected that this algorithm is memory bound. Figure 13 shows the roofline analysis of the **contract_tensor** kernel v3. The plot shows the arithmetic intensity vs the performance of the kernel. The 2 squared points and their horizontal lines show the double precision and single precision rooflines with maximal performance. The diagonal line shows the roofline of the memory bandwidth limit. If a kernel lies above this diagonal it is compute bound, while below means memory bound. The green point represents the performance of the analysed kernel. This shows that given the arithmetic complexity of the kernel, it almost reached its maximum performance since it is memory bound. Here the ceiling for compute bound problems would be around 16 teraFlops/s. The analysed kernel is significantly slower and is limited by the memory bandwidth rather than the compute power of the device.

The profiling results of the tensor contraction algorithm of v3 are further analysed in detail. In figure 15a, the compute and memory throughput of the kernel is shown for different block sizes on the x axis. Besides for a block size of 32, all other block sizes achieved a memory throughput of more than 80%, while the compute throughput remained around 30%. This highlights again that the kernel is indeed memory bound by the performance relative to diagonal roofline (see figure 13). The plot b) show the runtime of the kernel given different block sizes and here 32 gives a much larger runtime of about 1 ms, while the other block sizes give runtimes around 0.6 ms. This can be set in relation to the executed warp cycles per instruction (plot e)), where the opposite can be seen. The slower kernel needs only around 18 warp cycles per instruction, while the other kernels need 40 and more. The number of warp cycles per instruction should ideally be 1. High values can indicate memory stalls and cause latency. Since the kernel is rather small and runs very quickly, this is expected behaviour and can be further optimised by improving memory access patters for example. In plot c) the L1 and L2 cache hits are shown per block block size. Each kernel shows an L1 hit rate of about 80 % and an L2 hit rate of about 50 %. These results indicate that many values are cached which reduces memory access latency from global memory. However, we know that not all accesses happen in a coalesced fashion since we introduce an offset which adds a stride for each qubit (see algorithm 5). Therefore, to reduce the amount of uncoaslesced accesses to the global memory, we had to compute the indices again with each kernel execution (like in v1) which would increase the runtime. The absolute compute performance of each kernel is shown in plot d) reaching around 800 gigaFlops/s which is much lower than the maximally possible performance for compute bound problems as shown in plot a).

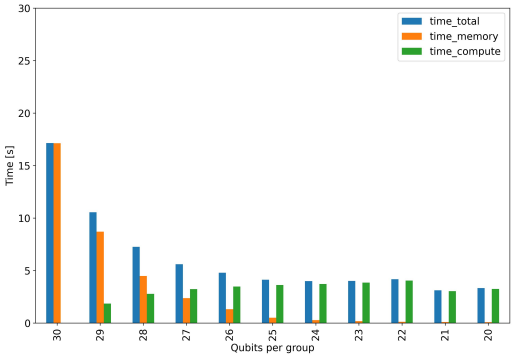


(a)

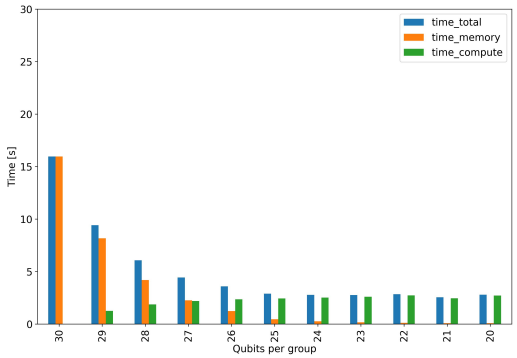
(b)



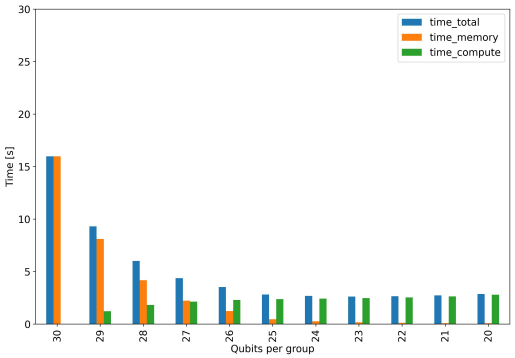
(c)



(d)



(e)



(f)

Figure 12: Runtime analysis of CUDA version 3 for 30 qubits. Different block sizes and qubits per group. $N_{group} = 2^{n_{group}}$ is used and computed in several sequential rounds of parallel search (see figure 6). The x axis shows the number of qubits per group. The y axis shows the time in seconds for the programme to run. Total time, memory time and compute time are shown. The block sizes a): 1024, b): 512, c): 256, d): 128, e): 64, f): 32 are shown. Experiments were conducted on the Nvidia H100 GPU.

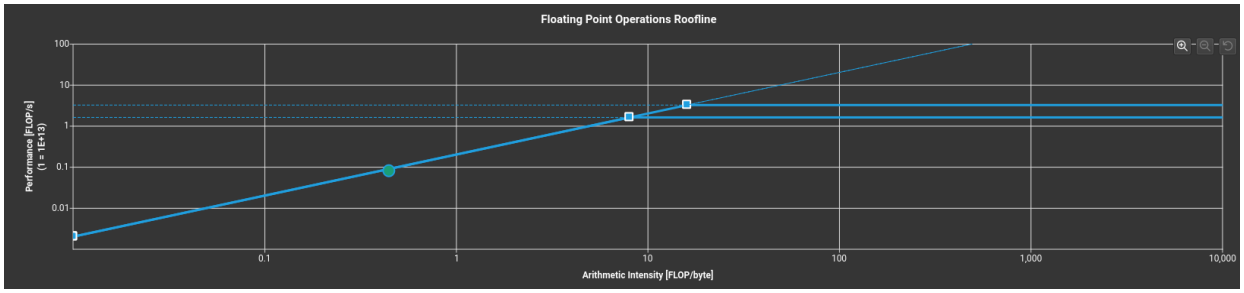


Figure 13: Nvidia Nsight Compute results of the roofline analysis of algorithm v3 with a problems size of 25 qubits and a block size of 1024.

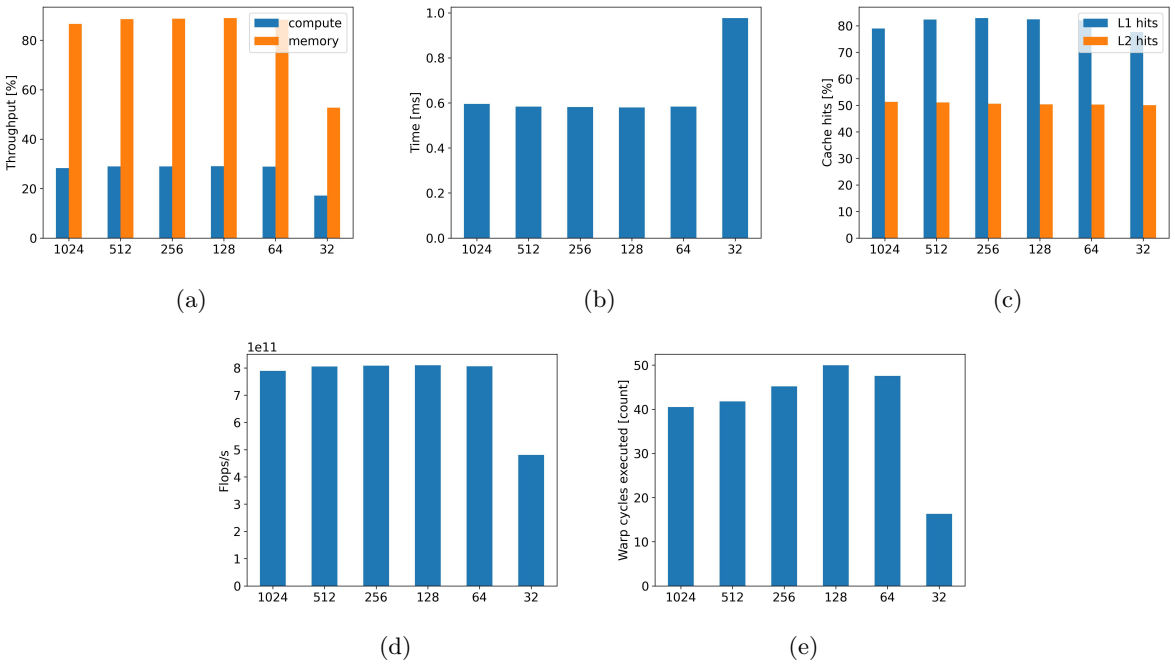


Figure 14: Profiling of CUDA version 3 (v3). The **contract_tensor** kernel was profiled and compared for n=25 qubits, 1 group and 1024, 512, 256, 128, 64, 32 as block size. Profiling was conducted using Nvidia Nsight Compute. a) shows the compute and memory throughput. b) shows the runtime of the kernel in ms. c) shows the percentage of cache hits for L1 and L2 caches. d) shows the absolute flops/s. e) shows the number of executed warp cycles per instruction. Experiments were conducted on the Nvidia H100 GPU.

5.7 Comparison of tensor contraction performance

After analysing the tensor contraction kernel of the best version, let's compare v3 with v1, v2 and the base version with respect to the different profiling statistics. Figure 15 shows the results from the profiling of the tensor contraction kernels of these algorithms. In plot a) the compute and memory throughput is shown. For all 4 versions the memory throughput is higher than 70% indicating again that all kernel versions are memory bound. In b) the runtimes are compared with a base time of more than 60 ms, v1 still runs at more than 10 ms while v2 and v3 achieve 0.7 and 0.6 ms respectively. Plot c) shows the needed warp cycles per

instruction. Here the base and v1 versions have very high values of more than 200 and 150 respectively. This stalls the kernel waiting for dependencies such as memory accesses and causes latency. Versions 2 and 3 however show better warp cycle times of around 20 and 40 respectively and hence cause less latency. The absolute performance of all 4 kernels is generally low with v2 having the best score of around 6 teraFlops/s. In theory the H100 GPU can achieve a performance of 30 teraFlops/s for double floats (link). With regards to the cache hits in plot e), the base and v1 versions show values for both L1 and L2 caches of around 80% and more, allowing for efficient caching, while v2 and v3 only show around 50 % for the L2 cache. However, their tensor contraction kernels run much slower and have more latency (plot b)). This can be explained with the fact that the base and v1 kernels compute the indices for the tensor contraction operation within the kernel and can efficiently access the global memory at the cost of slower runtime. In versions 2 and 3 the indices are only computed once in the beginning, reducing this redundancy. The lower cache performance can be explained with the index offset in the **old_linear_idx** and **new_idx** arrays where the kernel needs to access a different range of values based on the qubit where the gate is applied (see algorithm 5). We therefore trade speed for fewer cache hits. In general, introducing shared memory and computing the indices up front greatly improves the runtimes and reduces the latency of the tensor contraction kernel. Between v2 and v3 there is little difference however for the tensor contraction kernel.

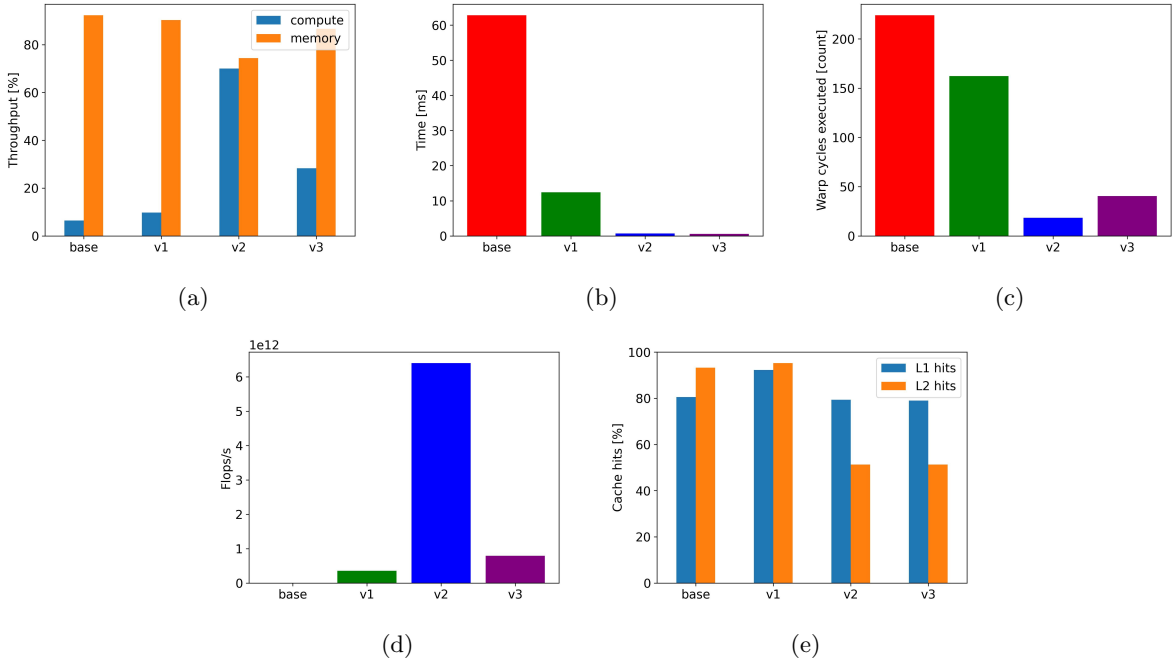


Figure 15: Profiling of tensor contraction kernel of different versions using Nvidia Nsight Compute. All 4 programmes (base, v1, v2, v3) were executed with a block size of 1024, n=25 qubits and only 1 group with all qubits was run. Compute and memory analysis was conducted. Red shows the base version, green v1, blue v2 and purple v3 of the CUDA implementations. a) shows compute and memory throughput b) shows the runtimes for the tensor contraction kernel of each implementation. d) shows cache hits for L1 and L2 caches. All results were obtained on the H100 GPU.

5.8 Speedup Comparison: H100 vs Grace Hopper

After discussing the different optimisations and their profile, the algorithms v1, v2 and v3 can also run nicely on 2 H100 GPUs in parallel by splitting the state vector in 2 parts. This should theoretically by twice as

fast. Additionally, the algorithms were also executed on the Hopper Grace GPU which has a higher memory bandwidth of up to 4TB/s compared to 3.35TB/s (see links: link1, link2). Since the given problem is memory bound, faster memory transfer is expected to improve the runtime of Grover's algorithm comparing an H100 GPU and Gracy. The results are shown in figure 16a. The plot a) shows the data for the H100 GPU(s) and b) shows the performance on Gracy. For version 3, also the best combination of parameters is shown, where the number of qubits per groups and chunks per group were finetuned using the data from figure 12. This also allowed for the computation of 35 qubits by running the algorithm with several groups. The 2 GPU versions of each algorithm are faster than their single GPU counterparts albeit not twice as fast. This can be explained with overheads due to synchronisation of streams and the devices. For the best chosen settings of v3 however, the 2 GPU version achieves double the speed for the last (35 qubits) datapoint with 78 sec compared to 39 sec only. Interestingly version v3 on a single Gracy GPU is even faster, with only 24 sec, than the 2 GPU version on the H100. This highlights that for a memory bound problem, even adding additional GPUs might not get the same results as higher bandwidth. The shown results also show that optimising v3 for a small k value and the group size is highly beneficial to reduce the runtimes.

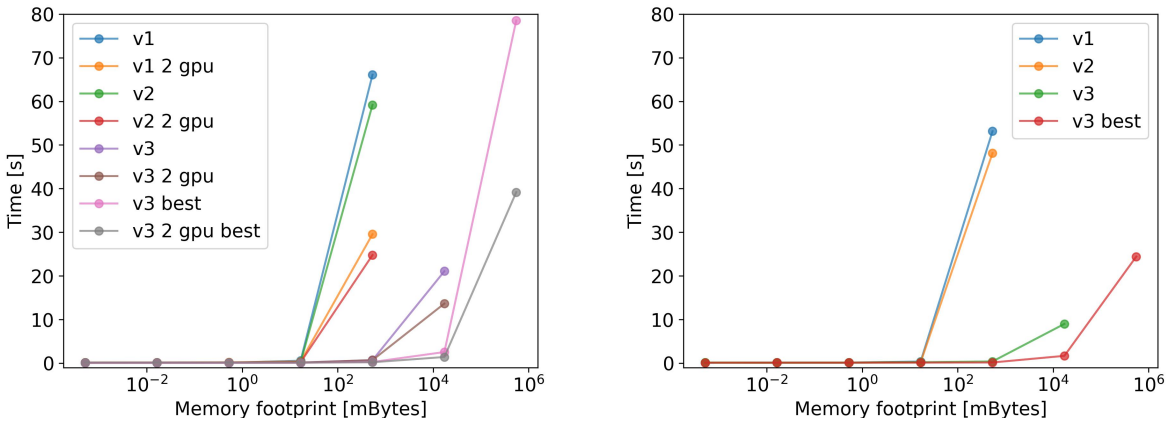
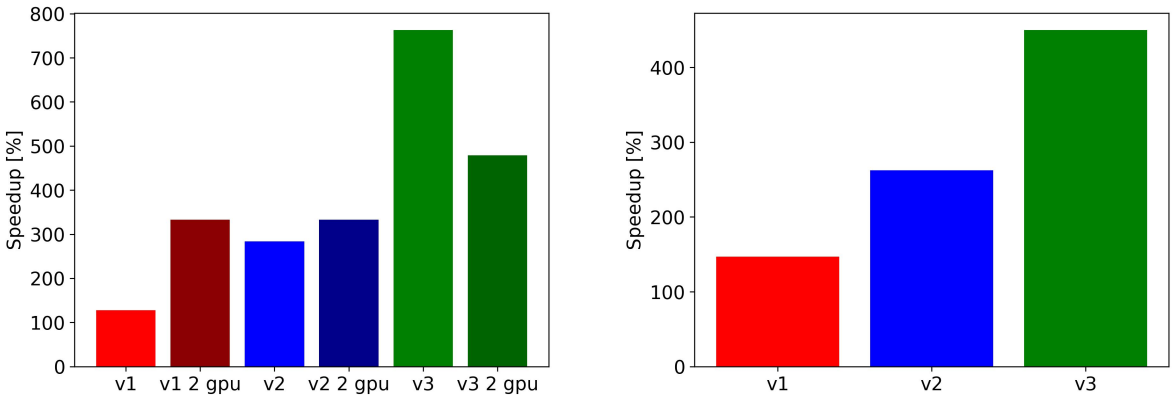


Figure 16: Speedup of optimised CUDA versions and their 2 GPU counterparts. Versions v1, v2, v3 and their 2 GPU versions were tested. For version 3, the best settings for the group size $N_{group} = 2^{21}$ and block size (64) were used to get optimal results. a) Shows the speedup on 1 or 2 H100 GPUs. b) shows the speedup of the single GPU versions on the Grace Hopper GPU. For all versions besides the best ones, a block size of 1024 and 1 group were chosen as settings.

Finally, the speedup in comparison of the different versions to the baseline CUDA implementation is shown in figure 17 at a problem size of 20 qubits or $N = 2^{20}$ since the compute times of the base version for 25 and more qubits are extremely long. We can see the expected speedup of up to 750% for version v3. The 2 GPU versions are lower at this relatively small problem size probably due to synchronisation overheads of the devices. On Gracy, the improvement is also large with the overall winner being version 3. Comparisons at larger problem sizes would be interesting as well but are not feasible when running the slow base version. In general, the large improvement of version 3 compared to v2 is mostly due to the discussed parallel implementation of the phase flip in the MCZ gate, whereas the other versions run this phase flip sequentially for each chunk leading to a large overhead especially for smaller chunk sizes such as 32 or 64. The parallel implementation of the phase flip removes this overhead and allows for the computation of many smaller chunks in parallel which in turn reduces the number of rounds k needed to amplify the solution state. As a comparison, running v3 for 30 qubits with 1 group and a block size (or chunk size N_{chunk}) of 1024 gives $k = 25$ and runs in ca. 30 seconds on the H100 GPU, while using a smaller chunk size of 32 gives us $k = 4$ and runs in 16 seconds. Therefore, more and smaller chunks reduce k, while mostly maintaining the speed of

the tensor contraction kernel (see figure 14).



(a) (b)

Figure 17: Speedup of v1, v2, v3 and their 2 GPU counterparts as a percentage of the baseline CUDA version. The values using 20 qubits, 1 group and 1024 as block size were used. a) Nvidia H100 GPUs were used. b) Nvidia Grace Hopper GPU was used.

6 Conclusion

The implementation of Grover's algorithm on CUDA GPUs demonstrates significant performance improvements over classical CPU-based approaches. By leveraging the parallel processing capabilities of GPUs, the optimised CUDA versions of the algorithm achieve substantial speedups, particularly for larger problem sizes where classical methods become infeasible. The use of shared memory and parallel search strategies in these CUDA implementations effectively reduces the computational overhead associated with the exponentially growing state vectors inherent in quantum computing.

Despite these advancements, the exponential growth of state vectors remains a critical challenge when using such state vector methods. To mitigate this matrix product states (MPS) are a promising approach to be implemented in future versions. MPS can efficiently represent entangled quantum states by decomposing the state vector into a series of smaller matrices, each corresponding to a subset of qubits. This method significantly reduces the memory requirements associated with handling large quantum states but requires a more complex implementation.

The results from the performance plots provide valuable insights into the efficacy of different CUDA optimisations. The baseline CUDA implementation showed considerable improvements over the CPU version, but the real gains were observed with the optimised CUDA versions. Version 3, which utilises shared memory and parallelised the multi-controlled Z gate application, demonstrated the highest speedup, achieving up to 750% improvement over the baseline CUDA implementation for 20 qubits. Profiling data indicated that this version is memory-bound, with a memory throughput exceeding 80% and compute throughput around 30%. Additionally, the algorithms are easy to scale on 2 or more GPUs and achieve up to 2 times faster runtimes.

The grid search for optimal parameters further highlighted that running the algorithm with several groups by setting fewer qubits per group as input argument can improved performance, particularly for large problem sizes. These configurations minimised memory overhead and maximised computational efficiency, allowing for the simulation of Grover's algorithm with up to 35 qubits. Additionally, the comparison between single and dual GPU setups revealed that while dual GPU configurations offered speed improvements, the gains were not always proportional to the expected speedup possibly due to synchronisation overheads.

Implementing MPS in future versions of the algorithm could further optimise memory usage and enhance performance, especially for simulations involving a large number of qubits. By integrating singular value decomposition (SVD) and truncating to the most significant singular values, the dimensionality of the problem can be controlled, maintaining computational feasibility without sacrificing too much accuracy of the simulation. This hybrid approach, which let's only the state vector grow exponentially, while the gates are constant in size has the strength that it is easily implemented and scalable to many GPUs. It uses the strengths of simple tensor contractions and demonstrates optimisation techniques for simulating complex quantum algorithms on classical hardware. Overall, this work underscores the importance of continued research and development in hybrid computational methods on GPUs, leveraging both classical and quantum techniques to advance the field of quantum computing.

References

[1] M. Akhtar, F. Bonus, F. R. Lebrun-Gallagher, N. I. Johnson, M. Siegele-Brown, S. Hong, S. J. Hile, S. A. Kulmiya, S. Weidt, and W. K. Hensinger. A high-fidelity quantum matter-link between ion-trap microchip modules. *Nature Communications*, 14(1), 2023.

[2] Thomas G. Wong. *Introduction to Classical and Quantum Computing*. Rooted Grove, Omaha, 2022.

[3] Erik Gustafson, Burt Holzman, James Kowalkowski, Henry Lamm, Andy C. Y. Li, Gabriel Perdue, Sergio Boixo, Sergei Isakov, Orion Martin, Ross Thomson, Catherine Vollgraff Heidweiller, Jackson Beall, Martin Ganahl, Guifre Vidal, and Evan Peters. Large scale multi-node simulations of $\mathbb{Z}_2$ gauge theory quantum circuits using Google Cloud Platform. oct 2021.

[4] Lov K. Grover. A fast quantum mechanical algorithm for database search, 1996.

[5] Andrew Eddins, Mario Motta, Tanvi P. Gujarati, Sergey Bravyi, Antonio Mezzacapo, Charles Hadfield, and Sarah Sheldon. Doubling the size of quantum simulators by entanglement forging. *PRX Quantum*, 3:010309, Jan 2022.

[6] Martin Ganahl, Jackson Beall, Markus Hauru, Adam G.M. Lewis, Tomasz Wojno, Jae Hyeon Yoo, Yijian Zou, and Guifre Vidal. Density Matrix Renormalization Group with Tensor Processing Units. *PRX Quantum*, 4(1), jan 2023.

Appendices

A Compute resources

A.1 H100 setup

A.1.1 CPU

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Address sizes:	52 bits physical, 57 bits virtual
Byte Order:	Little Endian
CPU(s):	64
On-line CPU(s) list:	0-63
Vendor ID:	AuthenticAMD
Model name:	AMD EPYC 9354 32-Core Processor
CPU family:	25
Model:	17
Thread(s) per core:	1
Core(s) per socket:	32
Socket(s):	2
Stepping:	1
Frequency boost:	enabled
CPU(s) scaling MHz:	89%
CPU max MHz:	3799.0720
CPU min MHz:	1500.0000
BogoMIPS:	6490.17
Flags:	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good amd_lbr_v2 nopl nonstop_tsc cpuid extd_apicid aperfmperf rapl pni pclmulqdq monitor ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs skinit wdt tce topoext perfctr_core perfctr_nb bpext perfctr_llc mwaitx cpb cat_l3 cdp_l3 invpcid_single hw_pstate ssbd mba perfmon_v2 ibrs ibpb stibp ibrs_enhanced vmmcall fsgsbase bmi1 avx2 smep bmi2 erms invpcid cqm rdt_a avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local avx512_bf16 clzero irperf xsaveerptr rdpru wbnoinvd amd_ppin cppc arat npt lbrv svm_lock nrip_save tsc_scale vmcb_clean flushbyasid decodeassists pausefilter pfthreshold avic v_vmsave_vmload vgif x2avic v_spec_ctrl avx512vbmi umip pku ospke avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg avx512_vpopcntdq la57 rdpid overflow_recov succor smca fsrm flush_l1d
Virtualization features:	
Virtualization:	AMD-V
Caches (sum of all):	

L1d:	2 MiB (64 instances)
L1i:	2 MiB (64 instances)
L2:	64 MiB (64 instances)
L3:	512 MiB (16 instances)
NUMA:	
NUMA node(s):	2
NUMA node0 CPU(s):	0-31
NUMA node1 CPU(s):	32-63
Vulnerabilities:	
Gather data sampling:	Not affected
Itlb multihit:	Not affected
L1tf:	Not affected
Mds:	Not affected
Meltdown:	Not affected
Mmio stale data:	Not affected
Reg file data sampling:	Not affected
Retbleed:	Not affected
Spec rstack overflow:	Mitigation; safe RET
Spec store bypass:	Mitigation; Speculative Store Bypass disabled via prctl
Spectre v1:	Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Spectre v2:	Mitigation; Enhanced / Automatic IBRS; IBPB conditional; STIBP disabled; RSB filling; PBRSB-eIBRS Not affected; BHI Not affected
Srbds:	Not affected
Tsx async abort:	Not affected

A.1.2 GPU

Technical Specifications	**H100 SXM**
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core*	989 teraFLOPS
BFLOAT16 Tensor Core*	1,979 teraFLOPS
FP16 Tensor Core*	1,979 teraFLOPS
FP8 Tensor Core*	3,958 teraFLOPS
INT8 Tensor Core*	3,958 TOPS
GPU Memory	80GB
GPU Memory Bandwidth	3.35TB/s
Decoders	7 NVDEC, 7 JPEG
Max Thermal Design Power (TDP)	Up to 700W (configurable)
Multi-Instance GPUs	Up to 7 MIGs @ 10GB each
Form Factor	SXM
Interconnect	NVIDIA NVLink™: 900GB/s, PCIe Gen5: 128GB/s
Server Options	NVIDIA HGX H100 Partner and NVIDIA-Certified Systems™ with 4 or 8 GP NVIDIA DGX H100 with 8 GPUs

A.2 Grace Hopper (Gracy) setup

A.2.1 CPU

Architecture:	aarch64
CPU op-mode(s):	64-bit
Byte Order:	Little Endian
CPU(s):	72
On-line CPU(s) list:	0-71
Vendor ID:	ARM
Model name:	Neoverse-V2
Model:	0
Thread(s) per core:	1
Core(s) per socket:	72
Socket(s):	1
Stepping:	r0p0
Frequency boost:	disabled
CPU max MHz:	3465.0000
CPU min MHz:	81.0000
BogoMIPS:	2000.00
Flags:	fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm jscvt fcma lrcpc dcpop sha3 sm3 sm4 asimddp sha512 sve asimdfhm dit uscat ilrcpc flagm ssbs sb paca pacg dcpodp sve2 sveaes svepmull svebitperm svesha3 svesm4 flagm2 frint svei8mm svebf16 i8mm bf16 dgh bti
Caches (sum of all):	
L1d:	4.5 MiB (72 instances)
L1i:	4.5 MiB (72 instances)
L2:	72 MiB (72 instances)
L3:	114 MiB (1 instance)
NUMA:	
NUMA node(s):	9
NUMA node0 CPU(s):	0-71
NUMA node1 CPU(s):	
NUMA node2 CPU(s):	
NUMA node3 CPU(s):	
NUMA node4 CPU(s):	
NUMA node5 CPU(s):	
NUMA node6 CPU(s):	
NUMA node7 CPU(s):	
NUMA node8 CPU(s):	
Vulnerabilities:	
Gather data sampling:	Not affected
Itlb multihit:	Not affected
L1tf:	Not affected
Mds:	Not affected
Meltdown:	Not affected
Mmio stale data:	Not affected
Retbleed:	Not affected
Spec rstack overflow:	Not affected

Spec store bypass:	Mitigation; Speculative Store Bypass disabled via prctl
Spectre v1:	Mitigation; __user pointer sanitization
Spectre v2:	Not affected
Srbds:	Not affected
Tsx async abort:	Not affected

A.2.2 GPU

Feature	**GH200**
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core	989 teraFLOPS* — 494 teraFLOPS
BFLOAT16 Tensor Core	1,979 teraFLOPS* — 990 teraFLOPS
FP16 Tensor Core	1,979 teraFLOPS* — 990 teraFLOPS
FP8 Tensor Core	3,958 teraFLOPS* — 1,979 teraFLOPS
INT8 Tensor Core	3,958 TOPS* — 1,979 TOPS
High-bandwidth memory (HBM) size	96GB HBM3 — 144GB HBM3e
Memory bandwidth	Up to 4TB/s — Up to 4.9TB/s
NVIDIA NVLink-C2C CPU-to-GPU bandwidth	900 GB/s
Power	Configurable 450 to 1000W (Memory + CPU + GPU)

B Running the programmes

Here are some examples on how to run the programmes with different parameters.
Arguments:

n:
The number of qubits giving a state vector of size $N = 2^n$.

markedState:
The marked state. It must be between 0 and N-1.

Number of chunks per group:
This value must split the state vector of size N into chunks of a size which is minimally 2.

Number of qubits per group:
The number of qubits per group defines the group state vector. This is useful for state vectors which cannot fit into the GPU memory completely. They can then be divided in groups of size N_{group}.

B.1 OMP version

The OMP version only takes the first 2 arguments like in this example:

```
// ./grover_omp n markedState
// run for 10 qubits with marked state 1
./grover_omp 10 1
```

B.2 CUDA Base version

Here, we add an argument for the block size:

```
// ./grover_cuda_baseline n markedState block_size
// run for 10 qubits with marked state 1 and block size 1024
./grover_omp 10 1 1024
```

B.3 CUDA v1, 2, 3

As an example we can run the programmes of v1, 2, 3 with the same input arguments:

```
    // ./grover_cuda_v1 n markedState n_chunks_per_group
       n_qubits_per_group

    // run for 10 qubits with 1 chunk per group and 1 group
    // 2^10 fits into 1 block of size 1024
    ./grover_cuda_v1 10 1 1 10

    // run for 10 qubits with 2 chunks per group and 1 group
    // 2^10 fits into 2 blocks of size 512
    ./grover_cuda_v1 10 1 2 10

    // run for 10 qubits with 2 chunks per group and 2 groups
    // 2^9 fits into 2 blocks of size 256
    ./grover_cuda_v1 10 1 2 9

    // run for 11 qubits with 1 chunks per group and 2 groups
    // 2^10 fits into 1 block of size 1024
    ./grover_cuda_v1 11 1 1 10
```

B.4 CUDA versions with 2 GPUs

These versions are the 2 GPU versions of v1, 2, 3 and divide the work between 2 GPUs running in parallel. Thus, we need at least 2 groups. Let's take the examples from the single GPU versions and adapt them for 2 GPUs:

```
    // ./grover_cuda_v1_2_gpu n markedState n_chunks_per_group
       n_qubits_per_group
    // n_qubits_per_group must be 1 lower than n

    // run for 10 qubits with 1 chunk per group and 2 groups
    // 2^9 fits into 1 block of size 512 on each device
    // The 2 groups are divided between the 2 devices and run in parallel
    ./grover_cuda_v1_2_gpu 10 1 1 9

    // run for 10 qubits with 2 chunks per group and 2 groups
    // 2^10 fits into 2 blocks of size 256 on each device
    // The 2 groups are divided between the 2 devices and run in parallel
    ./grover_cuda_v1_2_gpu 10 1 2 9

    // run for 10 qubits with 2 chunks per group and 4 groups
```

```
15      // 2^8 fits into 2 blocks of size 128 on each device
16      // The 4 groups are divided between the 2 devices and 2 groups are run
            in parallel at a time
17      ./grover_cuda_v1_2_gpu 10 1 2 8
18
19      // run for 11 qubits with 1 chunks per group and 2 groups
20      // 2^10 fits into 1 blocks of size 1024 on each device
21      // The 2 groups are divided between the 2 devices and run in parallel
22      ./grover_cuda_v1_2_gpu 11 1 1 10
```