12-2021

Continuous Variable Quantum Compilation Analysis

Federico Rueda
fxr8250@rit.edu

Continuous Variable Quantum Compilation Analysis

Federico Rueda

Continuous Variable Quantum Compilation Analysis

Federico Rueda

December 2021

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

RIT | Kate Gleason College of Engineering

Department of Computer Engineering

Continuous Variable Quantum Compilation Analysis

Federico Rueda

Committee Approval:

Dr. Sonia López Alarcón *Advisor* Date
Department of Computer Engineering

Dr. Amlan Ganguly Date
Department of Computer Engineering

Dr. Gregory Howland Date
Department of Physics & Astronomy

Abstract

Over the last few years quantum computers have rose to prominence as a solution for increasing computing power and tackling problems intractable by classical computers. Classical computers have struggled to meet the ever-increasing demand for data processing and modeling. Thus, quantum computers would meet the demand for heavy computing tasks that classical computers could never achieve. Today there exist two promising quantum technologies that have the potential to prove quantum supremacy in the near future: super-conducting qubits and Continuous Variable (CV) model. Out of the two, superconducting qubits have been at the forefront of quantum computing research. NISQ (Noise intermediate-scale quantum) are the existing superconducting qubit computers, and are defined by having small number of qubits with high inaccuracies due to quantum noise. The CV approach utilize photons in Gaussian states, qumodes, as the main processing unit. CV-based quantum computers present similar computation benefits to superconducting qubit devices. The ease of manufacturing and operation – due to being able to operate at room temperature – of CV-based quantum computers make it a likely candidate for wide adoption and accessibility compared to superconducting qubit systems. But because of the underlying differences in the two technologies, research and development of their software stacks have differed greatly, with a lack-thereof for CV devices. The goal of this thesis is to explain and analyze the compilers implemented by Strawberry Fields – cross platform python library to simulate and execute programs on photonic hardware – and propose an additional compilation step that will enable future, more flexible hardware implementations.

Contents

Acronyms

CV

Continuous Variable

GBS

Gaussian Boson Sampling

GKP

Gottesman, Kitaev and Preskill

OpenQASM

Open Quantum Assembly Language

SF

Strawberry Fields

Chapter 1

Introduction

1.1 Motivation

In recent years quantum computing has evolved from a theoretical research subject to reality. Companies and scientists around the world are creating physical quantum systems using different quantum technologies to advance the field. Current quantum computing systems are classified as Noisy Intermediate-Scale Quantum (NISQ) Computers, and are mainly implemented using superconducting qubit technologies. These quantum computers utilize a small number of qubits (50-100) to perform tasks which can potentially surpass the capabilities of today's classical computers, although technology is limited in the amount of qubits in a circuit, due to the inherent noise in the quantum gates. Even with these limitations, quantum supremacy over classical computers in specific computing tasks has been demonstrated [1]. Now, this current state is not regarded as the end goal for quantum computers, rather it is a step toward creating more powerful quantum technologies.

There are multiple technologies in development that target different computing problems and utilize different quantum properties. The differences in the underlying technologies causes their implementations to differ greatly. An example gaining traction in recent years are photonic based quantum computers, which utilize the Continuous Variable (CV) model instead of superconducting qubits. Photonic based

quantum devices introduce their own set of benefits, mainly in ease of manufacturing and operation, compared to NISQ. Moreover, the two quantum systems have been developed independently and thus their software stack differ greatly. With the software stack of photonic devices lacking thorough documentation of its software stack and the design decisions behind it.

1.2 Quantum Bits and Modes

Two novel paradigms that have the potential to bring quantum computing to reality are quantum bits (qubits) and quantum modes (qumodes). Both have the ability to solve similar compute-intensive tasks while having significantly different implementations. Qumodes take the approach of creating a quantum Continuous Variable (CV) model. In a CV model, the basic information-processing unit is represented by an infinite-dimensionless bosonic mode [2], its physical implementations requires the use of bosons (i.e. photons). While more prevalent, the qubit model, uses superconducting qubits to represent a discrete two state system. The inherent difference in how data is stored and computed means that quantum circuits performing similar tasks differ in logic and quantum gates. These differences have impacted how their software stacks are designed and implemented. For example, the *Open Quantum Assembly Language (OpenQASM)* was designed for superconducting qubits, while *Blackbird* assembly language was developed for photonic based quantum states [2]. This thesis seeks to compare qubits and qumodes and the software development stack.

1.3 Objectives

The objective of this research is to analyze and improve the software stack of Strawberry Fields (SF). While not an issue currently, as SF programs increase in size and complexity the overhead of simulation and execution will increase. We introduce a

compiler that will merge of Gaussian operations in a SF program with non-Gaussian and Gaussian operations. Thus the resulting program will contain the minimum amount of Gaussian operations required to keep the same functionality.

Chapter 2

Background

2.1 Qubits

In classical computing, a bit is the fundamental concept of computation and information. Qubit quantum computers build upon this concept with a quantum bit. Similarly to a classical bit, two possible states of a qubit are $|0\rangle$ or $|1\rangle$ due to it being represented in the standard or computational basis. The difference being that qubits can be in a state, superposition of $|0\rangle$ and $|1\rangle$. It is possible to form linear combination of states, called superpositions [3]: $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$. Where α and β are complex numbers. This means a qubit can be represented as a vector in a two-dimensional vector space. In the computational basis, a quantum state can be measured with an outcome of $|0\rangle$ or $|1\rangle$, which can then be mapped to classical states through measurement.

Furthermore, measuring qubits is remarkably different from measuring classical bits. In a classical computer, millions of bits are determined to have a value of 0 or 1 per second. But in quantum mechanics, rather a quantum state measurement is regarded as the projection of the state vector (qubit state) onto a vector of the orthonormal measurement basis (classical bit). This projection is inherently probabilistic. Meaning that there is a probability that classical '0' or '1' is measured. To obtain useful information about the quantum state, we must perform this measure-

ment multiple times. Multiple measurements allows us to determine the probabilities of obtaining a given state when measuring a qubit state. For example, a qubit state where both states $|0\rangle$ and $|1\rangle$ have an equal chance of being projected unto the measurement basis is denoted by $|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. When measured multiple times it can be determined that states $|0\rangle$ and $|1\rangle$ have a $(\frac{1}{\sqrt{2^2}})$ equal chance of being measured.

Qubit quantum computers use the superposition of states to store data, while performing operations is achieved with quantum gates. For example, analogous to the classical NOT operation is a NOT quantum gate, which interchanges the states of $|0\rangle$ and $|1\rangle$. Suppose we define a matrix X to represent the quantum NOT gate:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{2.1}$$

The quantum state $\alpha|0\rangle + \beta|1\rangle$ can be represented by a vector as:

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{2.2}$$

Thus, the gate operation can be calculated using a matrix multiplication of the gate matrix and the state vector. The corresponding output of the quantum NOT gate is:

$$X \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix} \tag{2.3}$$

2.2 Continuous Variable Model

The Continuous Variable (CV) model is a quantum computing approach that retains the same computational power of the qubit model. The main information-processing unit differ greatly, the CV model utilizes a infinite-dimensional bosonic

mode (qumode). The underlying quantum operators of this model work in a continuous spectra. The difference between qubit and CV systems is most evident in the basis expansions of quantum states [2]:

$$\textbf{Qubit:} \quad |\phi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2.4}$$

$$\textbf{Qumode:} \quad |\psi\rangle = \int dx\ \psi(x)\,|x\rangle \tag{2.5}$$

The qubits utilize a discrete set of coefficients, while the CV model has a continuum that models a bosonic harmonic oscillator, which is defined by the canonical mode operators $\hat{a}$ (annihilator) and $\hat{a}^\dagger$ (creation), and satisfy the relation $[\hat{a}, \hat{a}^\dagger] = \mathbb{I}$. Additionally, we define the position & momentum quadrature operators – operate on the phase space and have special properties, i.e. $\hat{x}\,|x\rangle = x\,|x\rangle$ – as such [**?**]:

$$\hat{x} := \sqrt{\frac{\hbar}{2}}(\hat{a} + \hat{a}^\dagger), \tag{2.6}$$

$$\hat{p} := -i\sqrt{\frac{\hbar}{2}}(\hat{a} - \hat{a}^\dagger), \tag{2.7}$$

where $[\hat{x}, \hat{p}] = i\hbar\hat{\mathbb{I}}$ (follows Uncertainty Principle). Furthermore, the position state $|x\rangle$ depicted in Equation 2.5 are the eigenstates of the $\hat{x}$ quadrature, $\hat{x}\,|x\rangle = x\,|x\rangle$. The same property applies for the momentum state $|p\rangle$. The quadrature operators – position & momentum – are able to create and annihilate Hamiltonian's called **Gaussian states** (ground or thermal), which are the medium for storing information in CV computation.

Gaussian states are non-classical states with Gaussian Wigner quasi-probabilistic functions. The Wigner function is a representation of the relationship between the position and momentum in the phase space that behaves similarly to a probability

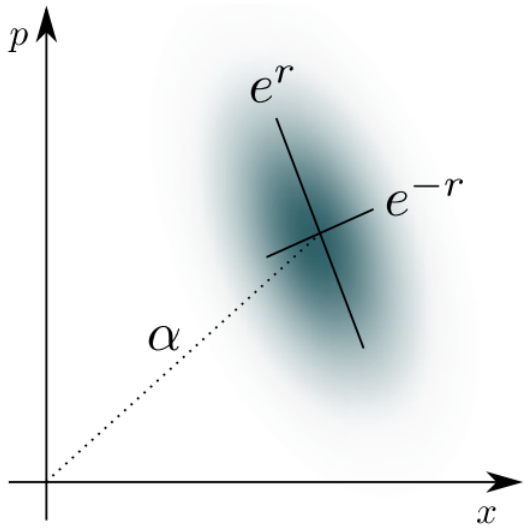


Figure 2.1: Squeezing and Displacement Visual representation of Gaussian state for single qumode. The shape and orientation are defined by the displacement α and squeezing $z = rexp(i\phi)$ parameters [2].

function, with some differences such as that the Wigner function can take negative values. The Gaussian distribution in a pure state can be centered in zero and be equally distributed around, but by interacting with it in the right way, the Gaussian can be shifted, displaced and squeezed [4, 5]. Therefore, these special Gaussian states are parameterized –in a single qumode–, by two continuous complex variables: displacement α and squeezing $z= s \cdot exp(i\phi)$ parameters. The authors of [2] created a visual representation of a Gaussian state for a qumode, which can be seen in Fig. 2.1. While, Table 2.1 shows a summary of pure Gaussian states used in the CV model, with their corresponding parameter values.

To create a comparison with qubit systems, the CV model implements discrete Fock states that are complementary to the continuous Gaussian states. Each of the pure Gaussian states mentioned in Table 2.1 can be expanded to the Fock-basis. For example, coherent states can be expressed in the form:

$$|\alpha\rangle = \exp(-\frac{|\alpha|^2}{2}) \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}\,|n\rangle}, \tag{2.8}$$

Table 2.1: Qumode pure Gaussian states and their relation to the displacement and squeezing parameters [2].

State Family	Displacement	Squeezing
Vacuum State $\|0\rangle$	$\alpha = 0$	$z = 0$
Coherent States $\|\alpha\rangle$	$\alpha \in \mathbb{C}$	$z = 0$
Squeezed States $\|z\rangle$	$\alpha = 0$	$z \in \mathbb{C}$
Displaced Squeezed States $\|\alpha, z\rangle$	$\alpha \in \mathbb{C}$	$z \in \mathbb{C}$
$\hat{x}$ eigenstates $\|x\rangle$	$\alpha \in \mathbb{C}$, $x = 2\sqrt{\frac{\hbar}{2}}Re(\alpha)$	$\phi = 0, r \to \infty$
$\hat{p}$ eigenstates $\|p\rangle$	$\alpha \in \mathbb{C}$, $x = 2\sqrt{\frac{\hbar}{2}}Im(\alpha)$	$\phi = \pi, r \to \infty$

where $|n\rangle$ are the Fock states (number of states) of which n are non-negative integers. Fock based measurements, photon counting, are essential to the CV model as it allows the discretization of Gaussian states, which allow for heterogeneous computing of classical and CV computers.

In the CV model, gates work as unitary operators (i.e $|\psi\rangle = U|0\rangle, U := \exp(-itH)$) that apply a bosonic Hamiltonian (H) over a certain amount of time (t). Unitaries are composed of operations that act on the momentum and position operators ($\hat{x}_i$ and $\hat{p}_i$). Gaussian operations contain at most quadratic operators (i.e. Squeezing; $S_i(z) = \exp(\frac{1}{2}(z^*\hat{a}_i^2 - z\hat{a}_i^{\dagger 2}))$) while Non-Gaussian contain three degrees or more (i.e. Cubic Phase; $V_i(\gamma) = \exp(i\frac{\gamma}{3\hbar}\hat{x}_i^3)$).

However, Hamiltonians in qubit systems use non-Gaussian operators called Pauli operators. These act on the discrete spin (1/2 and -1/2) of the fermions rather than on the position and momentum of the particles, which allows for discrete operations and measurements as seen in the computational basis. Qumodes on the other hand are operated upon and measured using their position and momentum, which are

continuous variables.

Multi-qumode operators can be created by applying a sequence of Gaussian or Non-Gaussian gates, which can act on one or two modes. A CV quantum computer is said to be universal if its able to to implement any unitary which is contains a polynomial in the mode operations. Gaussian and non-Gaussian gates, the elementary CV gates, are presented Table 2.2.

Table 2.2: Some useful CV gates. Beamsplitter is the only two-mode gate and Cubic phase is the only Non-Gaussian gate. $\alpha, \phi, z, \theta, \gamma$ are the parameters that affect the outcome of the operations.

Gate	**Unitary**
Displacement	$D_i(\alpha) = \exp(\alpha \hat{a}_i^\dagger - \alpha^* \hat{a}_i)$
Rotation	$R_i(\phi) = \exp(i\phi \hat{n}_i)$
Squeezing	$S_i(z) = \exp(\frac{1}{2}(z^* \hat{a}_i^2 - z \hat{a}_i^{\dagger 2}))$
Beamsplitter	$BS_{ij}(\theta, \phi) = \exp(\theta(e^{i\phi} \hat{a}_i \hat{a}_j^\dagger - e^{-i\phi} \hat{a}_i^\dagger \hat{a}_j))$
Cubic Phase	$V_i(\gamma) = \exp(i \frac{\gamma}{3\hbar} \hat{x}_i^3)$

There are three measurement types in the CV model. They can be distinguished between Gaussian and non-Gaussian, much like states and gates. There are two Gaussian (continuous) measurement types: homodyne and heterodyne measurements, with the third non-Gaussian measurement being photon counting. **Homodyne** detecting involves projecting a measurement onto the eigenstates of the quadrature operator $\hat{x}$. Whereas **heterodyne** is a simultaneous measurement of both $\hat{x}$ and $\hat{p}$. But because these operators do not commute – *canonical commutation relations* [6]–, there exists some uncertainty when heterodyne measurements are performed. Both measurements are defined as Gaussian due to the fact that their results are inherently continuous and unmeasured qumodes remain Gaussian (in multimode Gaussian states). Lastly, the photon counting measurement makes use of the particle-like nature of qumodes. It achieves this by projecting measurements onto a number of eigenstates $|n\rangle$. A

single mode photon-counting measurement of a multimode Gaussian state will cause the remaining modes to lose their Gaussian state.

Table 2.3: Key measurement types and operators for the CV model.

Measurement	**Measurement Operators**	**Measurement Values**	**Notes**
Homodyne	$\|x_\phi\rangle \langle x_\phi\|$	$x \in \mathbb{R}$	Where the Hermetian operator $\hat{x}_\phi = cos\phi\ \hat{x} + sin\phi\ \hat{p}$. Which performs a rotation of the state clockwise by ϕ before measuring.
Heterodyne	$\frac{1}{\pi} \|\alpha\rangle \langle\alpha\|$	$\alpha \in \mathbb{C}$	Referred as the projection onto the coherent states.
Photon Counting	$\|n\rangle \langle n\|$	$n \in \mathbb{N}$	Particle-like measurement, non-Gaussian.

2.3 Applications: Qubit vs Qumodes

The qubit models has been the main candidate to achieve quantum supremacy. This model was first introduced by Richard P. Feynman in [7], as a theoretical computing model. Since then there has been great deal of research into solving the hurdles of creating a physical qubit computer. Currently, there exists quantum processors that contain a small number of qubits (< 100), that have been developed by IBM and Google. There is still much research to be done as its been challenging to fit more qubits into a processor while retaining high fidelity. But this model (NISQ) has proven quantum supremacy and is the candidate for the first commercial quantum computers. Furthermore, there is plenty of research into creating quantum algorithms; which have significant computational speedups for factorization[8], search [9], or Fourier transform [10] when compared to its classical counterparts.

Meanwhile, the CV model is in its early development stages, with Xanadu being one of the only companies focusing in its development. This model has its merits when compared to qubits. The CV model retains the computational power of the qubit model while adding unique features. A CV quantum computer could simulate bosonic systems (Bose-Einstein condensates, photons, harmonic oscillators, electromagnetic fields) and model systems where continuous operators (position and momentum) are present. Additionally, CV and qubit systems can potentially tackle similar problems, such as graph-based problems and point processes [11]. So while in the premature stages, the CV model has the potential to model systems and accelerate computation tasks similar to that of qubits. Because of this it would be beneficial to explore the similarities and differences in the software implementation of CV and NISQ Quantum computers, while determining the possibility of creating software that can apply to both systems. There has been work on defining a unified formalism to conduct logical qubit operations using continuous variables [12]. But

it mainly focuses on the mathematical implementation rather than on the software stack. To the writers knowledge, there is no research that focuses on comparing the software for these technologies, due to CV computing being a new technology even in the quantum computing research community.

2.3.1 Boson Sampling

Boson sampling is a model for non-universal quantum computing that is relatively simple to implement using current technologies. Its simplicity stems from boson sampling devices being strictly passive, requiring only single-photon sources, linear optics (i.e. beamsplitters and phase-shifters), and photodetection. In a strict sense it lacks many of the requirements other quantum computers (i.e. memory, feedforwarding) at the cost of a limited range of applications.

All boson sampling devices begin by preparing an input state comprising of n single photons in m modes. The concept of modes is a state that simplifies the mathematical model of how light propagates and is able to describe the transport of energy and information using the quantity of photons in a mode [13]. The number of modes is generally scaled quadratically with the number of photons, $m = O(n^2)$. The input state can be mathematically expressed as,

$$|\phi_{in}\rangle = \hat{a}_1^\dagger ... \hat{a}_n^\dagger \, |0_1, ..., 0_m\rangle = |1_1, ..., 1_n, 0_{n+1}, ..., 0_m\rangle \tag{2.9}$$

where $|1_i\rangle$ and $|0_i\rangle$ represent if a photon is in mode i or if this mode is in the vacuum state. and $\hat{a}_i^\dagger$ is the photon creation operator for the ith mode. The input state is evolved through a linear optics network (linear interferometer), which implements a unitary map on the creation and annihilation operators,

$$\hat{U}\hat{a}_i^\dagger\hat{U}^\dagger = \sum_{j=1}^{m} = U_{i,j}\hat{a}_j^\dagger \tag{2.10}$$

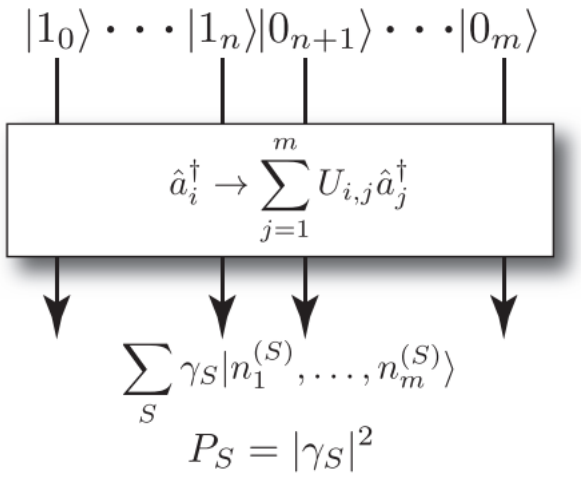


Figure 2.2: Boson-Sampling model, where n single photons are prepared in m modes. The modes are passed through a linear optics network $\hat{U}$. Lastly, the output statistics is sampled using photodetection, which are sampled many times to reconstruct the output distribution P_S [14].

where $\hat{U}$ is a unitary matrix describing the linear transformation performed by the linear optics network. The output state is a superposition of the different configurations – paths photons take to get to an output mode – of how the n photon reached the output nodes,

$$|\phi_{out}\rangle = \sum_S \gamma_S \, |n_1^{(S)}, ..., n_m^{(S)}\rangle \tag{2.11}$$

where S is a configuration, $n_i^{(S)}$ is the number of photons in the ith mode association with the configuration (S), and γ_S is the amplitude associated with the configuration. The probability of measuring the configuration S is given by $P_S = |\gamma_S|^2$. The amplitudes γ_S are related to matrix permanents

$$\gamma_S = \frac{Per(U_S)}{\sqrt{n_1^{(S)}! ... n_m^{(S)}!}} \tag{2.12}$$

where U_S is an $n \times n$ sub-matrix of $\hat{U}$, and $Per(U_S)$ is the permanent of U_S.

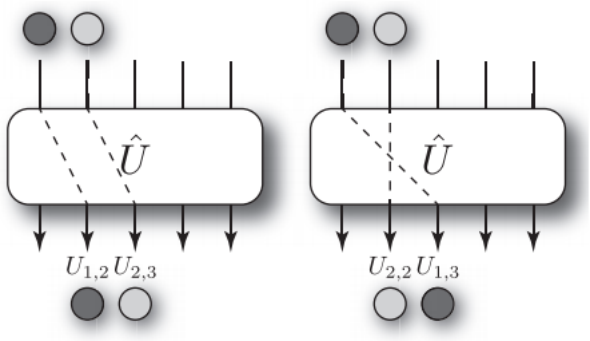


Figure 2.3: Two-photon boson sampling, where figuring out the amplitude ("probability") of measuring a photon at output modes two and three involve both bosons passing straight through, or swapping. [14]

The permanent is the mathematical representation of the paths taken by the photons. Consider a five-mode boson sampling device in which the first two modes have single photons, with the remaining ones in the vacuum state. Lets consider the case where one photon is measured at the output mode 2 and another at output mode 3. There are two ways this could happen. Either the first photon reaches mode 3 and the second, mode 2, or vice versa, i.e. the photons pass through or are swapped, see Figure 2.3. Therefore, there are $2! = 2$ ways in which the photons could reach the outputs. Thus, the amplitude can be written as a 2×2 matrix permanent:

$$\gamma_{2,3} = U_{1,2}U_{2,3} + U_{1,3}U_{2,2} = Per \begin{bmatrix} U_{1,2} & U_{2,2} \\ U_{1,3} & U_{2,3} \end{bmatrix} \tag{2.13}$$

We can expand the paths the photons take as the model is enlarged. Generally, with n photons, there will be $n!$ ways in which photons can reach the output modes. The associated amplitude will relate to a $n \times n$ matrix permanent. Simulating a boson-sampling model would involve calculating the permanent matrix, which is know to be **#P-** complete. Even the best known algorithm – by Ryser [15] – requires $O(2^n n^2)$ runtime. Thus, we can conclude that classically simulating boson-sampling by calculating matrix permanents would require exponential classical resources [14].

Additionally, because the number of photons scales quadratically with the number of photons ($m = O(n^2)$), for large systems it is statistically guaranteed that all photons arrive at different output modes. The number of configurations – paths photons can take to get to output mode – in the output mode scales as,

$$|S| = \binom{n+m-1}{n}, \tag{2.14}$$

which is exponentially driven by n. Thus, with an 'efficient' (i.e. polynomial) number of trials, we are unlikely to sample a given configuration more than once. This implies that we are unable to determine any given P_s with more than binary accuracy. Thus, boson-sampling devices do not let us compute matrix permanents, as doing so would require determining the amplitudes with a high level of precision, which requires an exponential number of measurements. Generally, boson-sampling experiments are run many times, each time performing photodetection at the output modes. For each run we sample from the distribution P_S, this yields the so-called *sampling problem*, where the goal is to sample a statistical distribution using a finite number of measurements. Because boson-sampling is a sampling problem, it can tackle a limited number of applications.

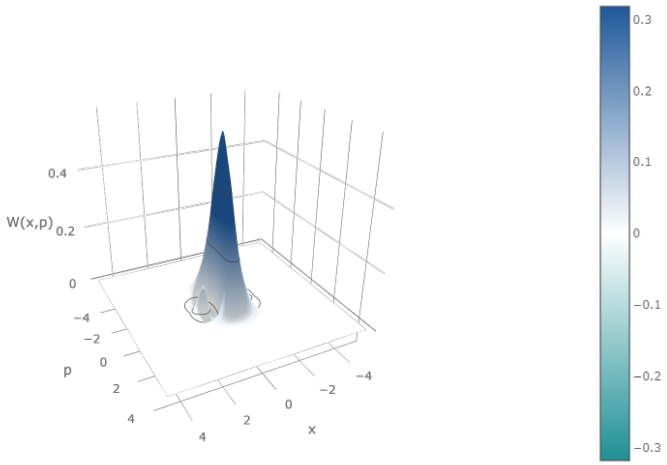


Figure 2.4: Squeezed Gaussian state viewed as a quasi-probability Wigner distribution function

2.3.1.0.1 Gaussian Boson Sampling

In Gaussian Boson Sampling (GBS), a Gaussian state – see Figure 2.4 – is taken as the input and photon number statistics are generated as the output, to efficiently sample distributions that are computational hard to sample in a classical implementations. The reason to use Gaussian states stems from the difficulty of generating accurate single-photon sources. Pure Gaussian states can emulate their behavior, and can be manipulated in the phase space.

Early devices utilizing this technology make use of postselected photon-pair states from probabilistic photon-pair sources (such as two-mode squeezed states) to emulate single photon input states. However these devices ignore the Gaussian nature of Gaussian states, as only a specific number of single photons are retained from the complete distribution and the squeezers are driven in low gain (mean photon number $\langle n \rangle \ll 1$). Ideally lifting this constraint on pure single photons input states and considering squeezed states with higher gain ($\langle n \rangle \approx 1$) will allow for a wider range of applications – such as simulating vibronic spectra – and in some special cases will be able to solve the sampling problem with the use of multi-mode thermal states and nonlinear-continuous variable quantum states [16].

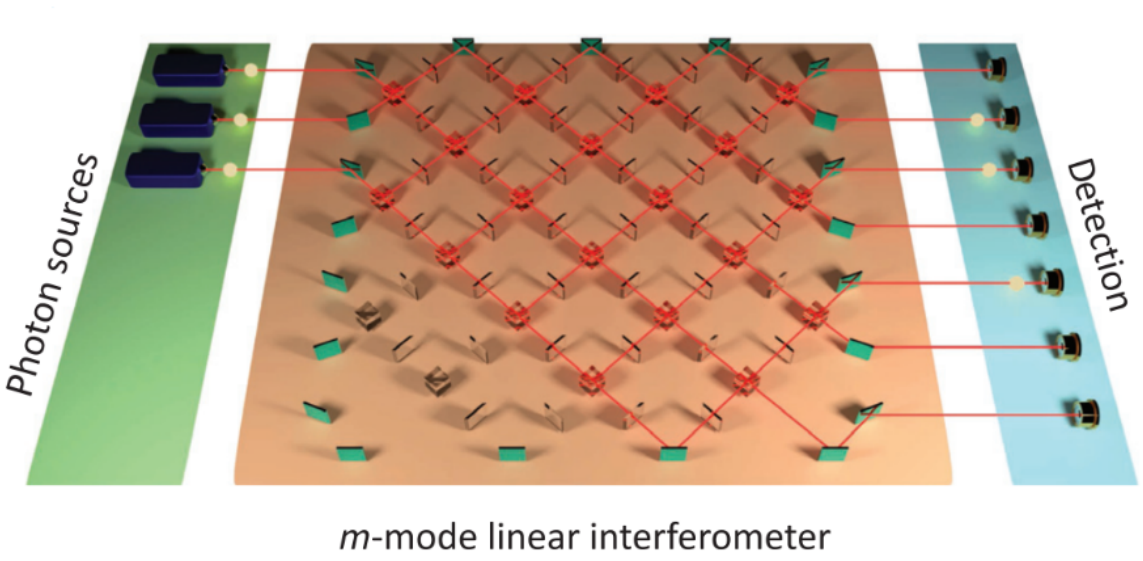


Figure 2.5: Model for Gaussian Boson Sampling device. The three main building blocks are: n Squeezed Gaussian input states, m-mode linear interferometer composed of beam-splitters and phase shifters – which can implement any linear transformation–, and photon count detectors on each of the output modes [17].

Furthermore, encoding information in GBS devices isn't analogous to how information is stored/used in traditional computers. In a traditional computer we store and perform operations on information stored in variables, meanwhile a photonic based quantum computer creates a continuous variable system – models harmonic oscillators – that would be too difficult for a traditional computer to simulate. Both devices would reach the same results but traditional computers have to perform mathematical algorithms that model the quantum systems. These algorithms generally perform and scale poorly.

2.4 Traditional Compilers vs Quantum Compilers

Traditional compilers for classical computers translate source code written in a high-level language into a set of machine-language instructions that can be understood by a digital computers CPU. They contain error-checking and optimization abilities that ensures source code can be executed correctly by the CPU in an efficient manner. A

compiler is essential to abstract out the most basic machine instructions (machine-language) to higher level concepts that can be easily understood and implemented. An example of such compiler is the GNU Compiler Collection (GCC); which supports compilation for multiple programming languages, hardware architectures and operation systems and has been in development for three decades.

Comparatively, compilers targeting quantum systems differ depending on the underlying technology when compared to traditional compilers. For example, IBM qubit systems can only apply operations between qubits that are connected together. So to abstract out this mapping problem from the user writing a quantum program, the compiler will restructure the quantum code so that it can be executed with the qubit configuration targeting. These compilers still fall under what a traditional compilers is defined, as its still handling all the logic needed to transform a high-level program to code that can be understood by the hardware.

On the other hand, – as we will explore later in the thesis – Strawberry Fields compilers do not meet all the definitions of traditional compilers. They do enable compilation of higher-level programs into lower-level understood by hardware and simulators, but to an extend. SF compilers behave more like helper structures that define decompositions and primitives allowed in the compilation process. Some do contain compilation logic but they behave as helper methods that can be called upon by other compilers performing more complex logic. This creates a hierarchical structure not seen in traditional high-level programming language compilers.

Chapter 3

Hardware

3.1 X8 Devices

Xanadu has developed and manufactured a photonic quantum chip using the CV model, called X8. Being such a novel device, it has many restrictions. While great for Gaussian Boson Sampling (GBS) type problems, it is restricted in the operations it can perform, meaning it cannot perform universal computing. Furthermore, the device contains eight qumodes – Xanadu's name for their high-dimensional/continuous qubits – which are divided into two groups, idler and signal modes. Qumodes 0 to 3 are the signal modes and qumodes 4 to 7 are the idler modes. Each of the qumodes are paired with one of the other group through a beam splitter, with the pairs being: (0, 4), (1, 5), (2, 6), (3, 7), as seen in Figure 3.1. Both groups are implemented identically but are categorized to emphasize that the operations that happen on the signal modes must be replicated to the idler modes.

3.1.1 Initialization of States

The X8 hardware initializes the qumode states by strictly enforcing the use of two-mode squeezing gates (**S2gate**). The S2gates serve to create a superposition of squeezed states between qumode pairs. As a comparison, this is similar to how superposition of qubits is achieved using Hadamard gates followed by CNOT gates. In

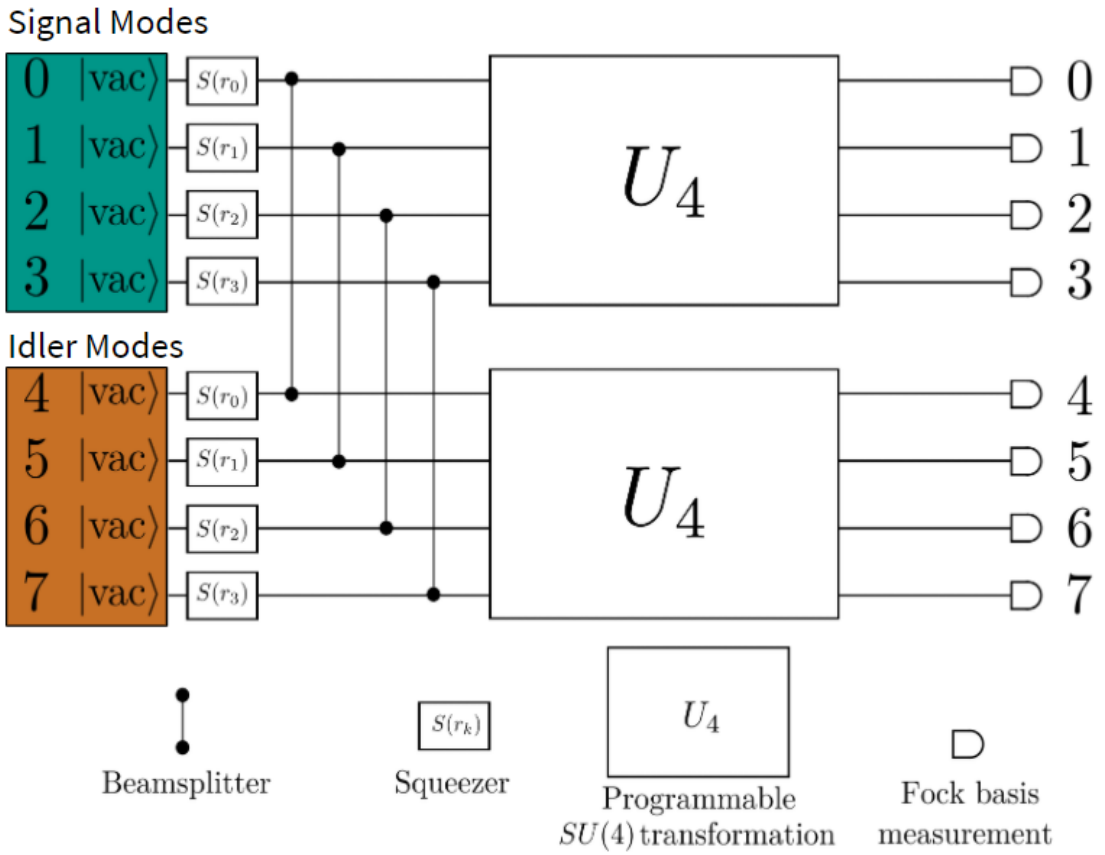


Figure 3.1: X8 Chip Topology. Currently, only the parameters $(r = 1, \phi = 0)$ and $(r = 0, \phi = 0)$ (corresponding to no squeezing) are allowed in the two-mode squeezing gates between signal and idle more pairs. Eventually, a range of squeezing amplitudes r will be supported.

this case the superposition is enforced, meaning a quantum program compiled to run on X8 must initialize with these operations. Though, the parameters of the squeezing gates can be specified by the strawberry fields program. Currently, the only parameters allowed for the S2gate are $(r = 1, \phi = 0)$ and $(r = 0, \phi = 0)$ (no squeezing), although Xanadu mentions that a range of squeezing amplitudes r will be supported in the future.

The S2gate can be decomposed into two opposite local squeezers sandwiched between two 50% beamsplitters. Mathematically it can be defined as:

$$S_2(z) = B^\dagger(\pi/4, 0)\,[S(z) \otimes S(-z)]\,B(\pi/4, 0) \tag{3.1}$$

Where $z = re^{i\phi}$ with $r \geq 0$ and $\phi \in [0, 2\pi)$. Using unitary definitions of the elementary gates, provided in Table 2.2. The definition can be expanded to:

$$S_2(z) = \exp(r(e^{i\phi}a_1^\dagger a_2^\dagger - e^{-i\phi}a_1 a_2)) \tag{3.2}$$

For the X8 chip, this can be further simplified by restricting the squeezing parameters ($r = 1$ or $0, \phi = 0$).

$$\phi = 0 \rightarrow e^{\pm i\phi} = 1 \tag{3.3}$$

$$S_2(r, \phi = 0) = \exp(r(e^{i0}a_1^\dagger a_2^\dagger - e^{-i0}a_1 a_2)) \tag{3.4}$$

$$S_2(r) = \exp(r(a_1^\dagger a_2^\dagger - a_1 a_2)) \tag{3.5}$$

$$S_2(r = 0) = \exp(0) = 1 \tag{3.6}$$

$$S_2(r = 1) = \exp(a_1^\dagger a_2^\dagger - a_1 a_2) \tag{3.7}$$

3.1.2 Programmable Interferometers

The actual processing for this quantum circuit takes place in the programmable 4x4 unitary block seen in Figure 3.1. The limitation in the processing is that the same operations are applied to each pair of qumodes, meaning that the U4 transformations depicted in Figure 3.1 have to be programmed identically. These U4 transformations are composed of BSgate (Beamsplitter), MZgate (Mach-Zehnder Interferometer), Rgate (Rotation Gate), and Interferometer operators. Xanadu defines these as primitive operations, but interferometer and BSgate operations are further decomposed to Mach-Zehnder Interferometers and Rotation gates.

3.1.2.1 Interferometer Meshes and Decompositions

The interferometer used by the X8 chip is implemented using universal multiport interferometers. An ideal multiport interferometer between N channels performs optical transformations which are described by a NxN unitary matrix U acting on electrical fields as $E_{out} = UE_{in}$. In quantum optics, U describes the transformation on the operators (position & momentum) of the input modes to those of the output modes. The authors of [18] propose a programmable multiport interferometer that is able to apply any linear transformation (U) between multiple channels (qumodes). The interferometers are composed of a mesh of beamsplitters and phase shifters which are scalable and straightforward to manufacture. A mesh, example seen in Figure 3.3, is composed of lines corresponding to an optical mode, and intersections between two modes correspond to a variable beam splitter, which are implemented using a Mach-Zehnder interferometer consisting of two 50:50 directional couplers (50% Beamsplitter), preceded by a phase shift at one input port [18].

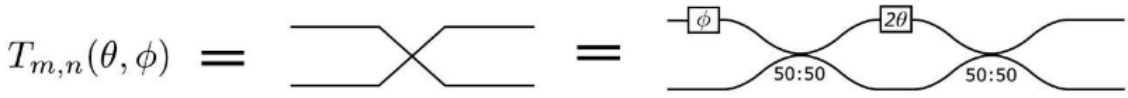


Figure 3.2: Mach-Zehneder interferometer diagram decomposition

The four interferometer meshes that Strawberry Fields supports are:

- **Rectangular:** rectangular mesh, with local phase shifts applied between interferometers. Uses the scheme described in [18], resulting in a rectangular array consisting of M(M-1)/2 beamsplitters, where M is the number of channels (qumodes) acted upon by the interferometer. By default this mesh decomposes the interferometer into beamsplitter gate (BSgate) operations.

- **Rectangular Phase End:** rectangular mesh that applies local phase shifts after all interferometers. Decomposed into beamsplitter gate operations.

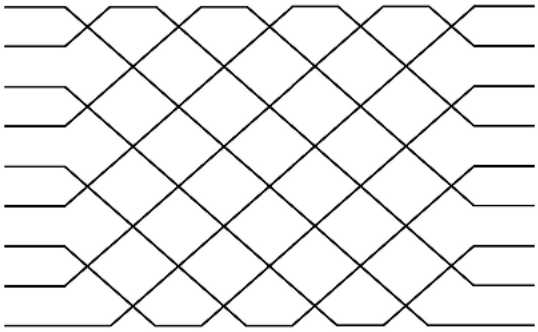


Figure 3.3: Interferometer Rectangular Mesh

- **Rectangular Symmetric:** Similar to rectangular phase end mesh but all beamsplitter gates are decomposed into pairs of symmetric beamsplitters and phase shifters. This is the mesh implemented in the X8 hardware.

- **Triangular:** triangular array of M(M-1)/2 beamsplitters. While identical to rectangular mesh, in amount of beamsplitters, [18] mentions that this design suffers from propagation loss, due to the top modes propagating for some distance before interacting with other nodes. In addition, the high symmetry of the rectangular mesh improves the loss tolerance when compared to the triangular mesh. Because of these factors, it's not expected for this mesh to be used in hardware implementations.

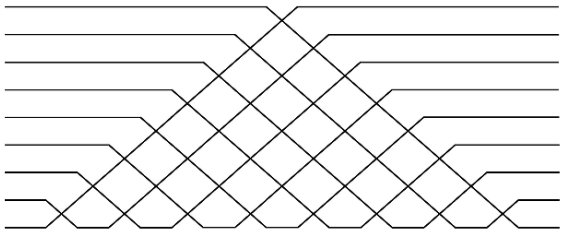


Figure 3.4: Interferometer Triangular Mesh

3.1.2.2 Mach-Zehnder Interferometer

The Mach-Zehnder interferometer is one of the elementary gates used by the X8 architecture. Its primary purpose is to determine relative phase shift variations between

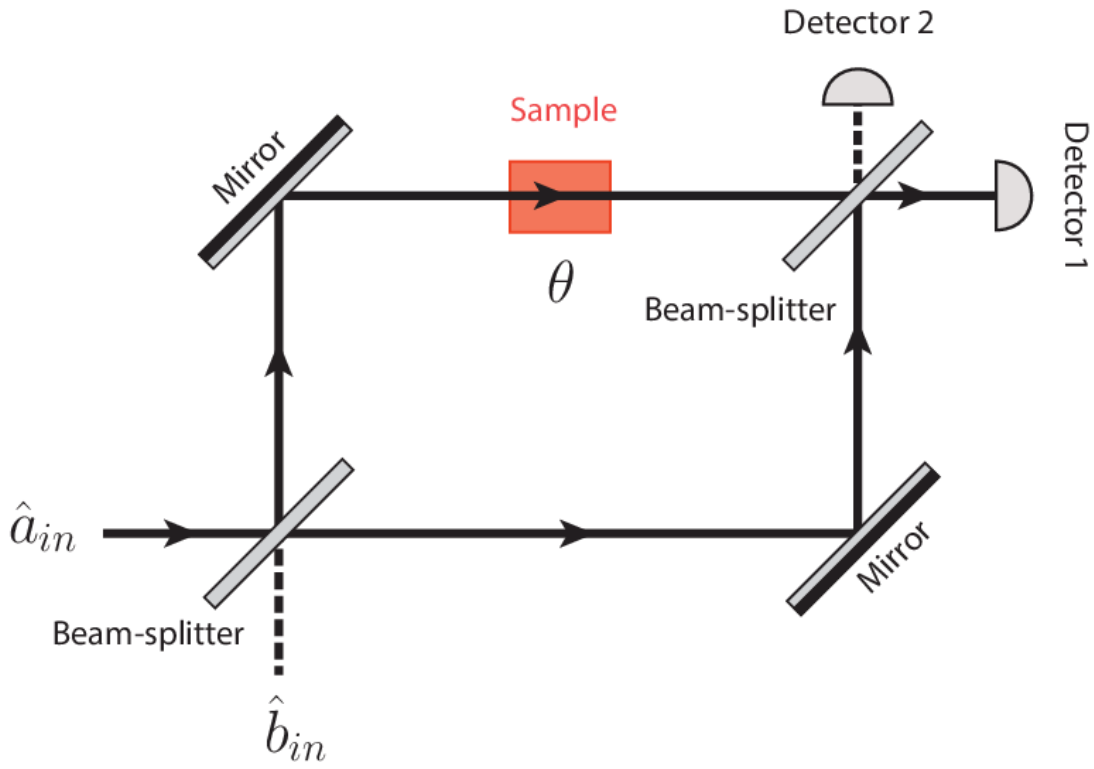


Figure 3.5: Schematic Diagram of a Mach-Zehneder Interferometer

two sources of light. It achieves this by having a two-mode light state get mixed on a 50/50 beamsplitter. Then, a relative phase θ is performed on one of mixed light beams. Lastly, the two light sources are mixed again using a 50/50 beamsplitter, from then on the sources of light can be passed through another interferometer or be measured using photon counters. The schematic showing the process can be seen in Figure 3.5.

From a mathematical perspective, Xanadu defines the Mach-Zehneder gate (MZ-gate) operation as:

$$MZ(\phi_{in}, \phi_{ext}) = BS\left(\frac{\pi}{4}, \frac{\pi}{2}\right)(R(\phi_{in}) \otimes I)BS\left(\frac{\pi}{4}, \frac{\pi}{2}\right)(R(\phi_{ext}) \otimes I) \tag{3.8}$$

This gate is similar to that shown in Figure 3.2. Where $\theta = \phi_{in}$ and $\phi = \phi_{ext}$.

With the difference being that the ϕ_{ext} is applied at the end of the interferometer, rather than before it.

As mentioned in Section 3.1.2.1, the X8 architecture implements the rectangular symmetric interferometer, this means two things:

1. Local phase shifts are performed after all interferometers. This phase shift is represented by ϕ_{ext} in (3.8).

2. All beamsplitters are decomposed into pairs of symmetric beamsplitters $BS\left(\frac{\pi}{4}, \frac{\pi}{2}\right)$, and phase shifters, which are achieved with the rotations on the Mach-Zehnder interferometer and elementary rotation gates.

3.2 Scalable Fault-tolerant Photonic Hardware

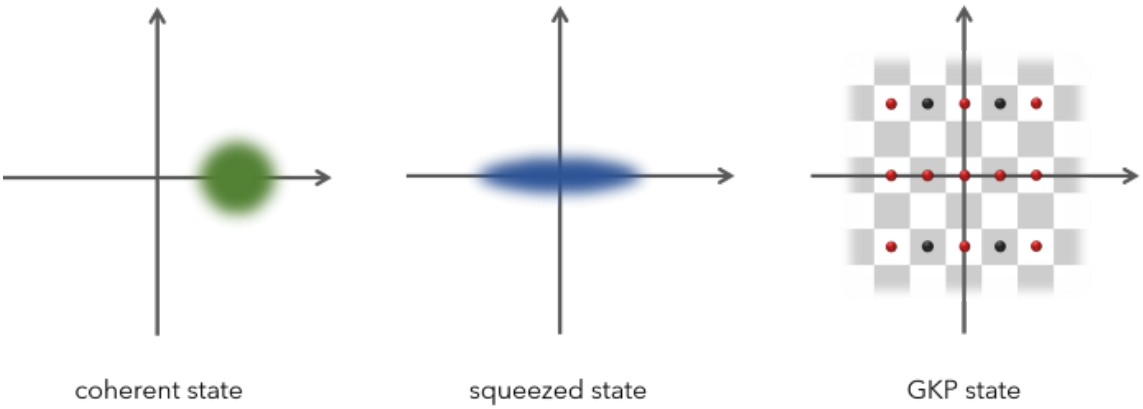


Figure 3.6: Visual Comparison between CV states Relevant to Future Photonic Hardware

Xanadu intents the X8 chip family to be the stepping stone to fault-tolerant scalable photonic hardware. The X8 chip, while perfect for GBS problems, is not a universal quantum computer, due to lacking the ability to perform non-Gaussian operations. The authors of [19] present a design for a photonic universal quantum computer, which use qubits encoded with a state of light mentioned in a method proposed by Gottesman, Kitaev and Preskill (GKP) in [20]. These qubits have several

benefits: quantum gates, operations and measurements on these states can be performed with Gaussian resources, which are natively available in photonic devices; they are robust against noise and optical losses; and can be operated at room-temperature making the design perfect for scalable fabrication and easy operation.

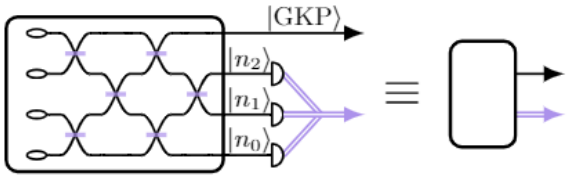


Figure 3.7: (left) GBS device for GKP state preparation. The purple lines represent classical logic for determining integrity of the GKP state. (right) Simplified representation of GBS device.

GKP states can be created using GBS devices (i.e. X8 hardware), when photons in all but one mode of light are counted, as seen in Figure 3.7. The light in the unmeasured mode emerges with something resembling little playing pieces arranged in a checkerboard pattern (GKP), as seen in Figure 3.6. This procedure is inherently probabilistic, which raises a concern as GKP qubits need to be readily available for a photonic quantum computer to work. To fix this, many GBS devices will have to be run simultaniously to increase the likelihood of making a GKP qubit, termed by [19] as *multiplexing*. Even with this method, too many GBS devices would be needed to create reliable GKP states, which would hinder the scalability of photonic architectures. Xanadu proposes a hybrid resource state composed of GKP and squeezed states. Multiplexed GBS devices will generate GKP states; however when these devices fail, the mode is instead prepared in a squeezed state, as seen in Figure 3.8. The squeezed mode is entangled with the other modes and can be operated upon much like a GKP state. With this method the number of GBS devices no longer becomes a hindrance.

The introduction of Gaussian neighbours (squeezed states) to the GKP modes creates another issue. When measured, the intrinsic structure of a GKP mode helps

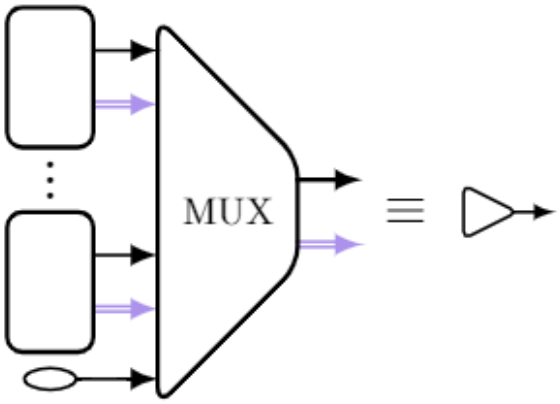


Figure 3.8: Multiplexed state generation. A successfully generated GKP state is directed from a GBS device to the output port. If no GBS device produces a GKP state, the output of the multiplexing device is swapped for a deterministically generated squeezed state (depicted by the ellipse on the bottom left). Right-hand side shows simplified diagram for the hybrid quantum light source [19].

reduce noise in the quantum state through GKP error correction. On the other hand, measuring a squeezed state introduces a known amount of random noise into neighbouring nodes, which if not accounted for will reduce the accuracy of the computation. Thus, to remedy this the authors of [19] propose a novel decoding procedure for the hybrid cluster state. The decoder takes the noisy measurements and uses the knowledge of the squeezed state locations to produce higher quality qubit readout values.

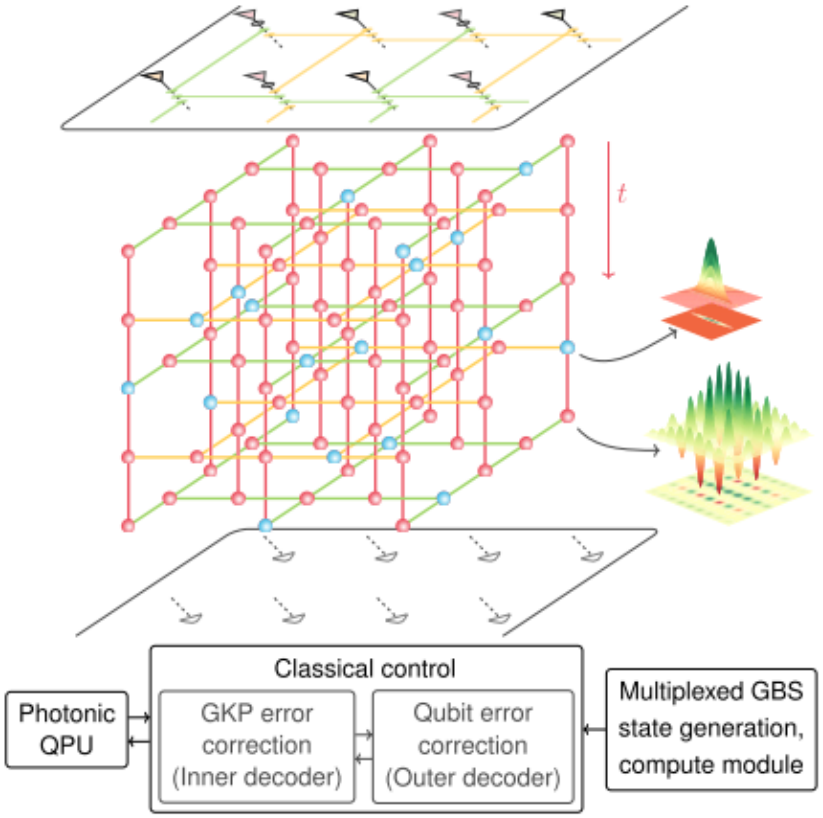


Figure 3.9: Photonic quantum computation using hybrid resource states. A planar chip (top) generates the states required for fault-tolerant quantum computation. The modes comprising the lattice are either GKP (red dots) or squeezed states (blue dots). The light is measured at the homodyne detectors (bottom), whose output is decoded [19].

3.3 Quantum Supremacy of GBS Devices

Quantum supremacy has been a focus point of researchers working on novel quantum technologies. Showing computation advantages of quantum devices over classical computers is essential to proving the feasibility of the quantum technologies. To achieve quantum supremacy with Gaussian Boson Sampling devices, a considerable amount of input squeezed states (qumodes) need to be prepared and operated upon. For example, the authors of [21] have proven quantum supremacy using a GBS device containing: 50 input states, a 100-mode interferometer, and 100 single-photon detectors. The researchers claim that the resulting output states were sampled at a rate $\sim 10^{14}$ faster than using state-of-the-art simulation strategies and supercomputers. Their success was short-lived, as the release of the paper encouraged researchers to improve the simulation algorithms to a point that a classical computer could achieve better performance than the GBS device. Even with this setback, it is apparent that as GBS devices grow in size they will be better suited to tackle quantum supremacy problem.

Chapter 4

Strawberry Fields

4.1 Compilers

Part of the objective of this thesis is to get a better understanding of what compilation entails in a continuous variable quantum computing environment entails. The compilation process is highly dependent on the underlying technology implemented and/or simulated by the SF environment. So to understand the design decisions and code of these compilers, one must have knowledge of the quantum concepts used by the hardware, specifically see 2.2 and 2.3.1. In particular, this work looks at the Strawberry Fields programming environment for Gaussian Boson Sampling developed by Xanadu. The compilation process for a Strawberry Fields program is initiated when **Program.compile()** is called. This compilation process can be simplified into three stages:

1. **Validation**: Validates properties of the design, such as the number of qumodes, operations, measurements.

2. **Decomposition**: Once the design has been validated, certain gates are transformed into sequences of simpler gates. Optimization – the action of simplifying the circuit to make it execute faster – takes place in this stage if specified by user.

3. **General Compilation**: If the program specifies a target device, the compiler

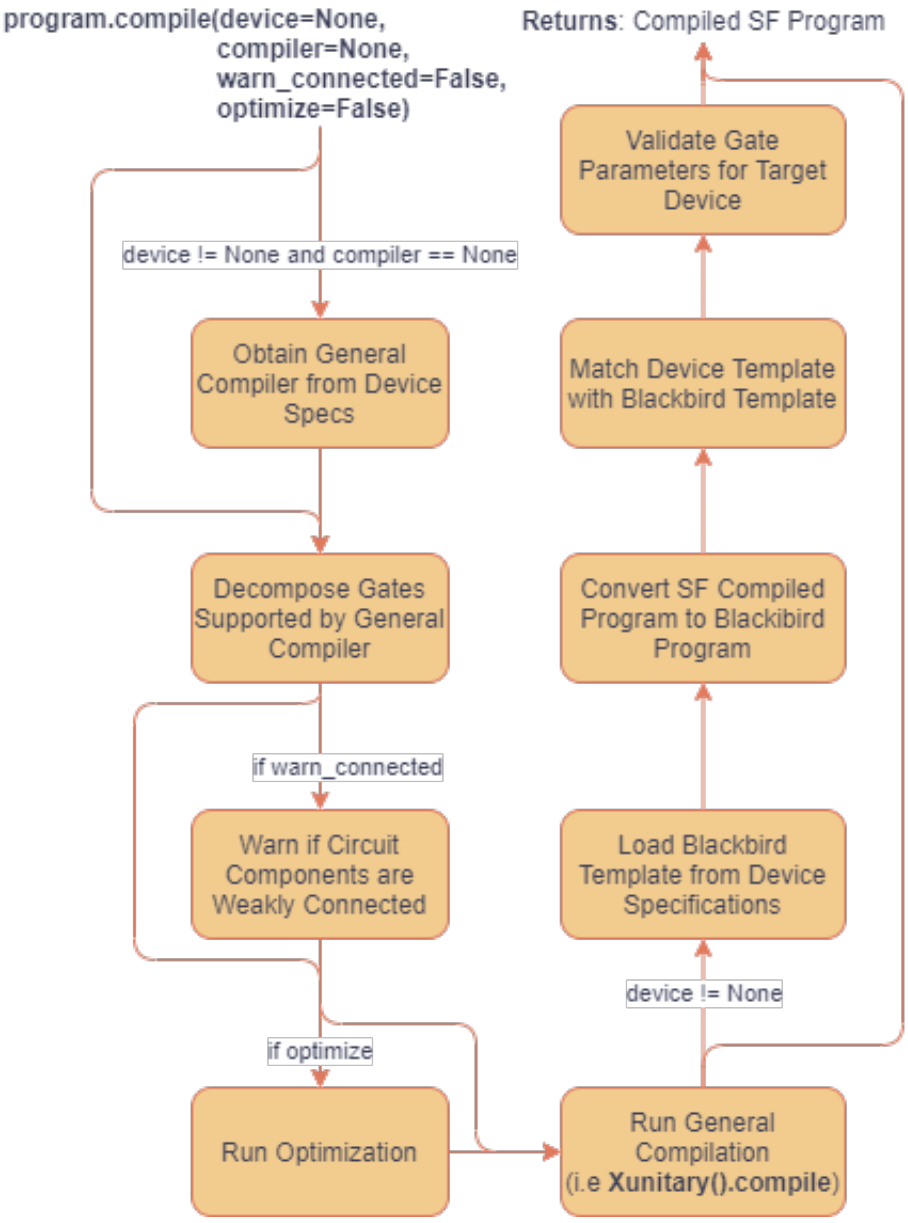


Figure 4.1: Compilation Steps Followed by all Strawberry Fields Programs. If a device is specified, extra steps take place to compile program for use in hardware

will execute some combinational logic for transforming the proposed design into an equivalent design which can be executed in the target device.

Every Strawberry Fields (SF) program goes through the process depicted by Figure 4.1. The parameters passed to program.compile() decide the compilation process for the SF program. The relevant parameters are:

- **compiler**: String that specifies the compiler used for general compilation (i.e. "Xunitary"), see Section 4.1.0.1 for more info. If not specified defaults to None.
- **warn_connected**: Boolean, when true, warns the user if a circuit is weakly connected, meaning a circuit when converted to a DAG (Directed Acyclic Graph) is weakly connected. A weakly connected DAG means that one or more of its subgraphs are not connected by an edge. It is meant to warn the user that there might be errors in their program. For example, when compiling the program in Listing 4.1, it is converted to the DAG shown in Figure 4.2, which clearly shows the two disconnected subgraphs. To get rid of the warning, one could add a Beamspliter gate between q0 and q3, ops.BSgate(0.5, 0.125) | (q(0), q(3)). The default value of warn_connected is False.

```
1    prog = sf.Program(4)
2    with prog.context as q:
3        ops.S2gate(1.0) | (q[0], q[4])
4        ops.Rgate(0.453) | q[0]
5        ops.Rgate(0.453) | q[3]
6    prog_compiled = prog.compile(warn_connected=True)
7
```

Listing 4.1: Weakly Connected Circuit Compilation

- **optimize**: Boolean, when true, optimizes the program. The optimizations are based on algebraic properties of the operations constituting the circuit. This

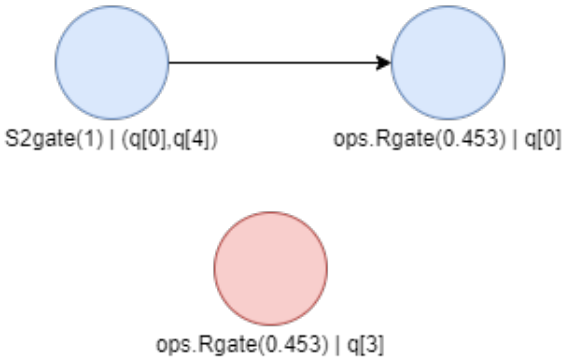


Figure 4.2: Weakly Connected Directed Acyclic Graph (DAG)

includes combining two consecutive gates of the same gate family and removing pair of gates that perform inverse operations. Thus, the simplified circuit is cheaper (resource wise) and faster to execute. The default value of optimize is False.

- **device**: Device specification object that describes the target device (hardware) in which the program will be run on. This object contains: target device name, the default general compiler (i.e. "Xunitary"), Black Bird template depicting layout of hardware, gate parameters allowed by hardware. Listing 4.2 shows how the device specification is obtained – given one has access to the hardware – and used to compile the program.

```
1    prog = sf.Program(8)
2    # Add operations....
3    eng = sf.RemoteEngine("X8")
4    device = eng.device_spec
5    prog_compiled = prog.compile(device=device, compiler="Xcov")
6
```

Listing 4.2: Compiling Program for X8 hardware

4.1.0.1 Compilation Steps

The compilation process, when specified, utilizes a general compilation step that validates and modifies a SF program to match a certain structure. This step is essential for compiling programs that run on the X class devices. The restrictions set in place by X class devices require the program to match a specific gate structure, which is achieved by the logic implemented in the compilers. There are three types of compilers that have a hierarchical structure (some serve others), as seen in Figure 4.3: backend, general, and X class.

All of the general compilers define primitive operations (operations allowed in the compiled program), and decompositions allowed to be performed by the compiler. If a program contains an operation not defined in the primitive or decomposition of the compiler being used, then the compiler will throw an error and will not allow you to compile your program. Furthermore it is important to note that if an operation is defined as primitive in a compiler, it does not mean that it wont be modified or removed from the compiled program. For example, Xcov supports squeezing gate (Sgate) as a primitive operation but because it compiles programs for X8 devices, – which do not contain Squeezing gates – no such operation is allowed in the compiled program. Primitives and decomposition operations can be seen in Table 4.1 and Table 4.2, respectively.

4.1.1 Backend Compilers (Gaussian, Fock)

These compilers define certain primitives and decompositions (see Table 4.1), but do not change structure of program. They are mainly used for programs running on the simulation backends. They are building blocks for the next layers of the compiler hierarchy, like the Gaussian Unitary compiler and the X Class compilers.

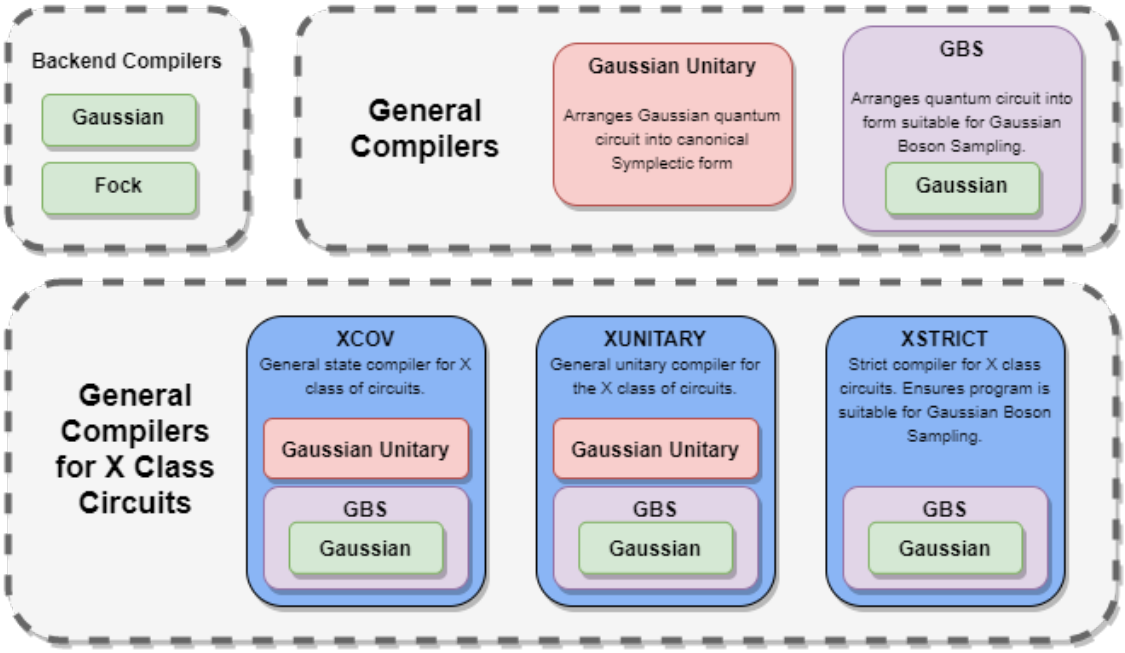


Figure 4.3: Strawberry Fields General Compiler Hierarchy

Table 4.1: Primitive Operations Allowed by General Compilers

	Primitives			
	State Preparations	Gates Single Mode	Gates Multi-Mode	Measurements
Gaussian	Vacuum, Coherent, Squeezed, Displaced Squeezed, Thermal, Gaussian	Displacement, Squeezing, Rotation	Beamsplitter	Homodyne, Heterodyne, Fock, Threshold
Fock	Vacuum, Coherent, Squeezed, Displaced Squeezed, Thermal, Fock, Catstate, Ket, Density Matrix	Displacement, Squeezing, Rotation. Cubic Phase, Kerr	Cross-Ker, Beamsplitter, Two-Mode Squeezing	Homodyne, Fock
Gaussian Unitary		Displacement, Squeezing, Rotation	Mach-Zehnder (MZgate), Beamsplitter, Two-Mode Squeezing, Interferometer, Gaussian Transform	
GBS	Same as Gaussian	Displacement, Squeezing, Rotation, Fourier	Beamsplitter	Same as Gaussian
Xunitary		Rotation	Beamsplitter, MZgate, Interferometer, Two-Mode Squeezing	Fock
Xcov		Squeezing, Rotation	Beamsplitter, MZgate, Interferometer, Two-Mode Squeezing	Fock
Xstrict		Rotation	MZgate, Two-Mode Squeezing	Fock

4.1.2 General Gaussian Compilers

These General Gaussian compilers implement compile methods that can be used by other general compilers. They currently serve as helper functions for the X class compilers. Their main job is to organize the computations in a canonical way that is suitable for the next compilation steps. The requirements to be able to implement the circuit description on actual hardware are two: on one hand, that the overall

Table 4.2: Decompositions Allowed by General Compilers

	Decompositons
Gaussian	Interferometer, GraphEmbed, BipartiteGraphEmbed, Gaussian Transform, Gaussian, Pgate, CZgate, CXgate, MZgate, Fouriergate, Two-Mode Squeezing
Fock	Interferometer, GraphEmbed, BipartiteGraphEmbed, Gaussian Transform, Gaussian, Pgate, CZgate, CXgate, MZgate, Fouriergate
Gaussian Unitary	GraphEmbed, BipartiteGraphEmbed, Gaussian, Pgate, CXgate, CZgate, Xgate, Zgate, FourierGate
GBS	Xgate, Zgate, Two-Mode Squeezing
Xunitary Xcov	BipartiteGraphEmbed

implementation can be expressed as symplectic matrices (see 4.1.2.1), and on the other, measurements that will reveal the final count on the modes need to be separated and implemented at the end of the circuit, as opposed to interleaved with other operations.

4.1.2.1 Gaussian Unitary

The Gaussian Unitary attempts to arrange a quantum program into the canonical symplectic form matrix. The symplectic matrix encapsulates the phase-space representation of the Gaussian transformation performed by the program [22]. Being able to compute the sympletic matrix ensures that the circuit can be implemented as a sequence of Gaussian operations. After compilation, the circuit will consist of two operations, a Gaussian transform and a displacement operation.

```
circuit = sf.Program(1)
with circuit.context as q:
    Xgate(0.4) | q[0]
    Zgate(0.5) | q[0]
    Sgate(0.6) | q[0]
    Dgate(1.0+2.0j) | q[0]
    Rgate(0.3) | q[0]
    Sgate(0.6, 1.0) | q[0]
```

```
9
10 compiled_circuit = circuit.compile(compiler="gaussian_unitary")
11 >>> compiled_circuit.print()
12 GaussianTransform([[ 0.3543 -1.3857]
13                     [-0.0328  2.9508]]) | (q[0])
14 Dgate(-1.151+3.91j, 0) | (q[0])
```

Listing 4.3: Compilation of a random allgaussian SF program into a Gaussian Transform operation and Displacement gate which are mathematically identical as the initial program.

Single-mode gates supported: Displacement, Squeezing, Rotation.

Multi-mode gates supported: MZgate, Beamsplitter, Two-mode Squeezing.

4.1.2.2 Gaussian Boson Sampling (GBS)

Validates and arranges quantum circuit into a form suitable for Gaussian Boson Sampling circuits, see Section 2.3.1. This involves:

1. Identifying the Fock measurement out of the list of computations
2. Re-organize Fock measurements to ensure that they are consecutive and take place at the end of the circuit.
3. Combines all Fock measurements into a *MeasureFock* command.

4.1.3 Xstrict

Besides defining the primitive operations of the X8 architecture, Xstrict ensures that the circuit matches the structure of a Gaussian Boson Sampling problem, see 2.3.1. Xstrict achieves this by calling the compile method from the GBS compiler class, creating its dependency as seen in Figure 4.3.

4.1.4 Xunitary

Xunitary is the default compiler used for compiling programs running on X8 devices. It is tasked with:

- Decomposing interferometer unitary into Mach-Zehnder and rotation gates in a symmetric rectangular mesh structure, see Section 3.1.2.1.
- Ensuring all two-mode squeezer (S2gate) operations are performed on the correct signal and idler qumode pairs. If a pair is missing the S2gate command, an S2 gate is inserted, with a squeezing parameter of zero (r=0). See Section 3.
- Converting multiple Fock measurement commands into one *MeasureFock* command at the end of the circuit.

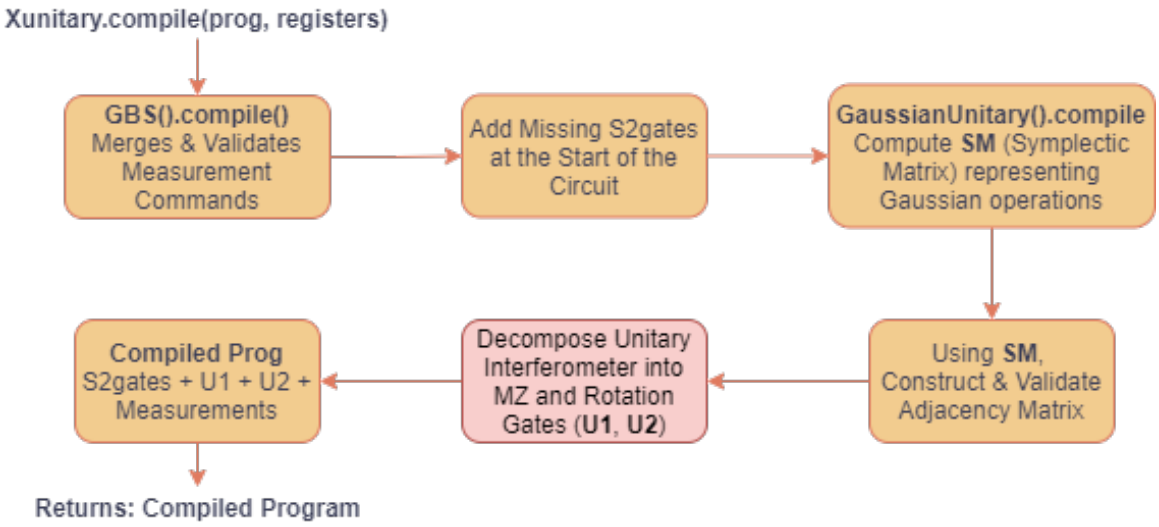


Figure 4.4: Xunitary compilation steps. The step that drives the cubic growth in compilation time is outlined in red, see Section 4.1.6.

Furthermore, the canonical symplectic form returned by the Gaussian Unitary compiler is used to construct the adjacency matrix of the interferometer unitary. A circuit running on hardware must have an adjacency matrix with the following structure:

$$A = \begin{bmatrix} 0 & B \\ B^T & 0 \end{bmatrix}$$

Where B is the adjacency matrix of a graph representing the rectangular symmetric interferometer in the X8 device. The edges represent the operations performed by the interferometer and squeezing gates, while the vertices represent the qumodes. Xanadu refers to this type of matrix as a embedded bipartite graph. In this case the bipartite graph is said to be the matrix representation of the interferometer. Thus, it can be decomposed to extract a sequence of MZ and rotation gates, which is used to create the compiled program, as seen in Figure 4.4. The sequence extracted (operations in the **U1** interferometer in Figure 3.1) applies only to the signal modes, to match the hardware, the sequence is copied and qumodes are re-mapped following signal and idler mode pairs. For example, the command MZgate(3.14, 0) (q[2], q[3]) is copied and modified to MZgate(3.14, 0) (q[5], q[6]).

Lastly, the compiled program is packaged by combining the S2 gates, interferometer on signal & idler modes, and measurement commands, in that order.

4.1.5 Xcov

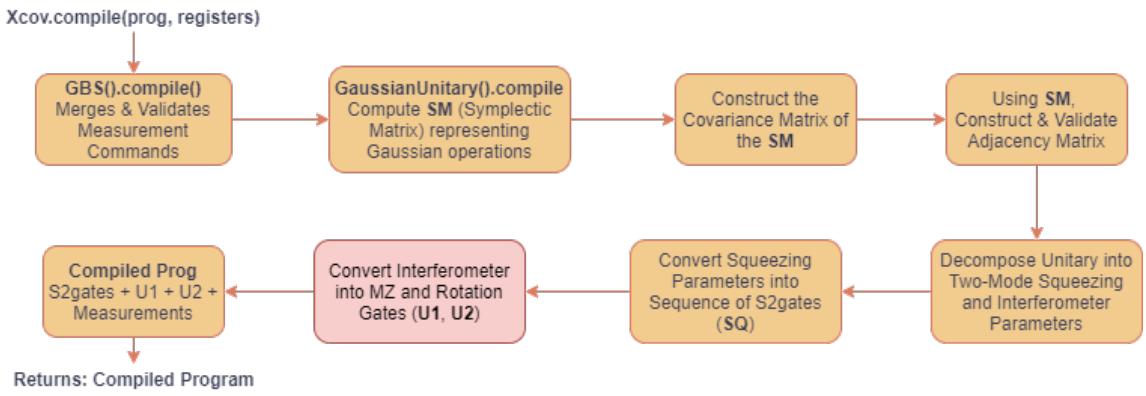


Figure 4.5: Xcov compilation steps. The step that drives the cubic growth in compilation time is outlined in red, see Section 4.1.6.

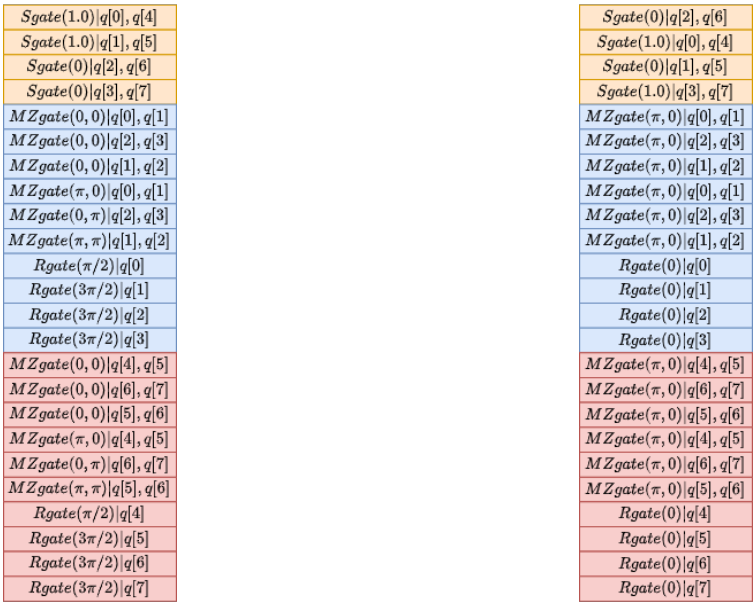
The Xcov compiler serves as an alternative to Xunitary for compiling quantum

programs targeting X8 hardware. Even though, these compilation steps perform the same task, their outputs can differ even if given the same SF program. In Xunitary, the adjacency matrix – an embedded bipartite graph in Xanadu's terminology – is calculated from the Symplectic Matrix (SM). Alternatively, Xcov uses the covariance matrix of the SM to create the adjacency matrix. The main difference is being able to include the squeezing gates (S2gates) Gaussian operations into the SM matrix – as seen in Figure 4.5 – rather than excluding them from the SM matrix, and having to deal with them beforehand. The issue with this method is that it is guaranteed that the first squeezer is the one with the largest squeezing parameter, with the second squeezer containing the second largest and so on. This implies that for trivial circuits where the original unitary is the identity, the Xcov compiler will end up reordering the squeezers causing the unitary to become a permutation matrix, which may or may not be a desirable feature. This behaviour can be shown by compiling the SF program in Listing 4.4. Figure 4.6 shows that the output programs for Xunitary and Xcov differ when given the same input, shown in Listing 4.4.

```
prog = sf.Program(8)
U = np.identity(4)
with prog.context as q:
    # Initial squeezed states
    ops.S2gate(1.0) | (q[0], q[4])
    ops.S2gate(0) | (q[1], q[5])
    ops.S2gate(1.0) | (q[3], q[7])
    # Identity Interferometer on the signal modes (0-3)
    ops.Interferometer(U) | (q[0], q[1], q[2], q[3])
    # Identity interferometer on the idler modes (4-7)
    ops.Interferometer(U) | (q[4], q[5], q[6], q[7])
    ops.MeasureFock() | q

xcov_prog = prog.compile(compiler="Xcov")
xuni_prog = prog.compile(compiler="Xunitary")
```

Listing 4.4: Identity Interferometer Example Program



(a) Xcov **(b)** Xunitary

Figure 4.6: Compilation Results from Program shown in Listing 4.4

4.1.6 Timing Comparisons between Xcov and Xunitary

Compilation time for SF programs targeting X8 devices is irrelevant at current circuit sizes (8 qumodes). But what would happen to the execution time of SF compilers for larger GBS circuits? This is a specially important question for researchers – i.e. [21] – attempting to prove quantum supremacy using a hundreds of modes in GBS devices, refer to Section 3.3. If there is ever work on creating large GBS devices using the SF environment, it would be useful to examine how the SF compilers perform at large circuit sizes.

The compilers that will be analyzed, Xcov and Xunitary, were chosen due to their complexity and their role in compiling programs targeted for X8 devices. The most essential part of performing timing analysis is developing SF programs that can be generated at different qumode sizes and be random per iteration of the timing test. To achieve this a python function was created, whose parameter defined the qumodes in the program, that would populate the S2gate operations on all qumode pairs – specified by X8 architecture – and generate a random interferometer for the unitary. The parametrization of SF programs allowed for a wide range of samples which in turn make for more precise profiling of the compilers.

```
def create_prog(qumodes=8):
    half_qmode = qumodes // 2
    U = random_interferometer(half_qmode)
    prog = sf.Program(qumodes)

    @operation(half_qmode)
    def unitary(q):
        ops.Interferometer(U) | q

    with prog.context as q:
        # Initial squeezed states
```

```
12          for i in range(half_qmode):
13              ops.S2gate(1.0) | (q[i], q[i+half_qmode])
14
15          unitary() | q[:half_qmode]
16          unitary() | q[half_qmode:]
17
18          ops.MeasureFock() | q
19      return prog
```

Listing 4.5: Function for SF program creation used in timing analysis of compilers

The *timeit* python module was used to perform the timing measurements. One of the issues with python timing analysis is that a python programs performance is often affected by background processes. This module addresses this issue by implementing a method that repeats a function call timing measurement and returns a list of measurements results, which gives us more samples to choose from. Generally for our purposes compiler timing measurements for each circuit size were run 10 times, with bigger circuits (i.e. > 150 qumodes) only running five times per compiler. The minimum compilation time of these iterations was stored, as recommended by the documentation of the *timeit* module. The measurement results were stored in a *csv* file, for easier data plotting and analysis.

The data was plotted, in logarithmic scale, using the *matplotlib* python module, and can be seen in Figure 4.7. Compilation time for the biggest circuit size (300 qumodes) took around 20 seconds, while the smallest (8 qumodes) took 4ms to compile. The big spike in the data corresponds to a spike in execution time from 64 to 66 qumodes, which were 74ms and 477ms, respectively. The 650% increase in compilation time was caused by the interferometer unitary decomposition into Mach-Zehnder (MZgate) and Rotation operations. The probable cause for the spike is a lack of cachable memory in the decomposition function that takes place for circuits larger than 64 qumodes. This behaviour could be explored further but it was concluded

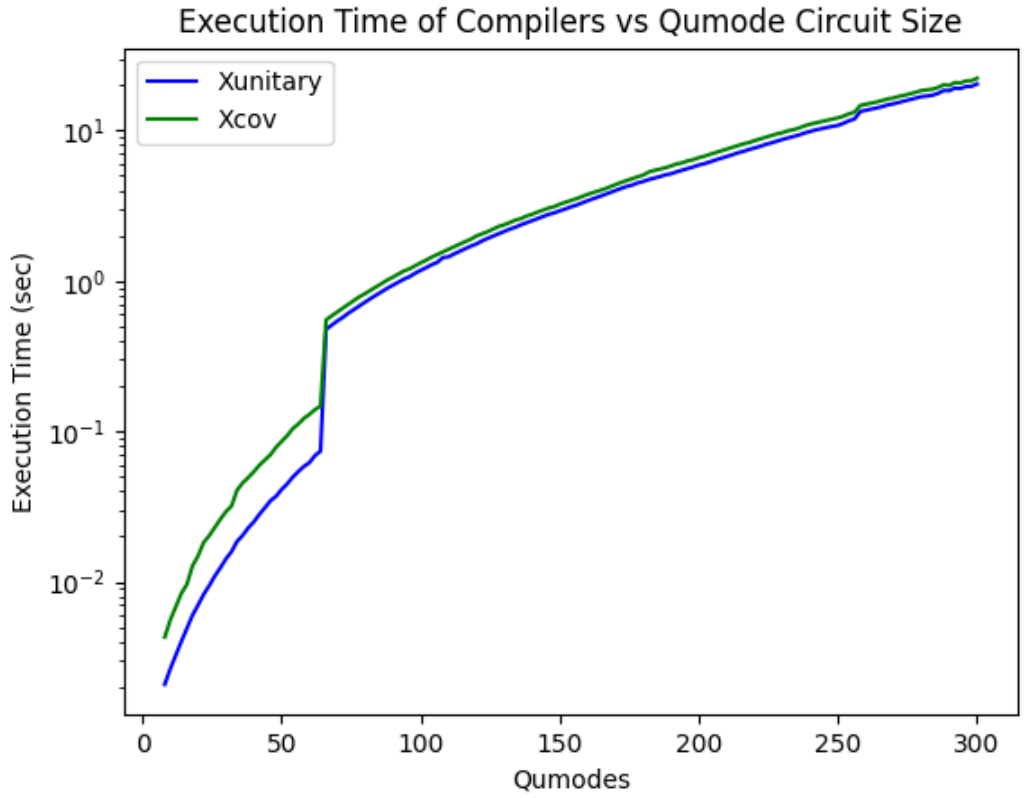


Figure 4.7: Execution time for Xunitary and Xcov compiler at various qumode program sizes, in logarithmic scale.

that the spike will have negligible effects on the compilation of large programs, which is the focus of this timing analysis.

The data points from 66 qumodes and on follow cubic growth, allowing for data extrapolation. Figure 4.8 shows the extrapolation of the timing data up to 1000 qumodes. The compilation time with 1000 qumodes would theoretically take 1000 seconds and 966 seconds – $\sim$ 16 minutes – for Xcov and Xunitary, respectively. Compiling for 1000 qumodes takes a considerable amount of time, but the difference between Xcov and Xunitary is negligible. This concludes that execution time is not an important characteristic when trying to differentiate between Xcov and Xunitary.

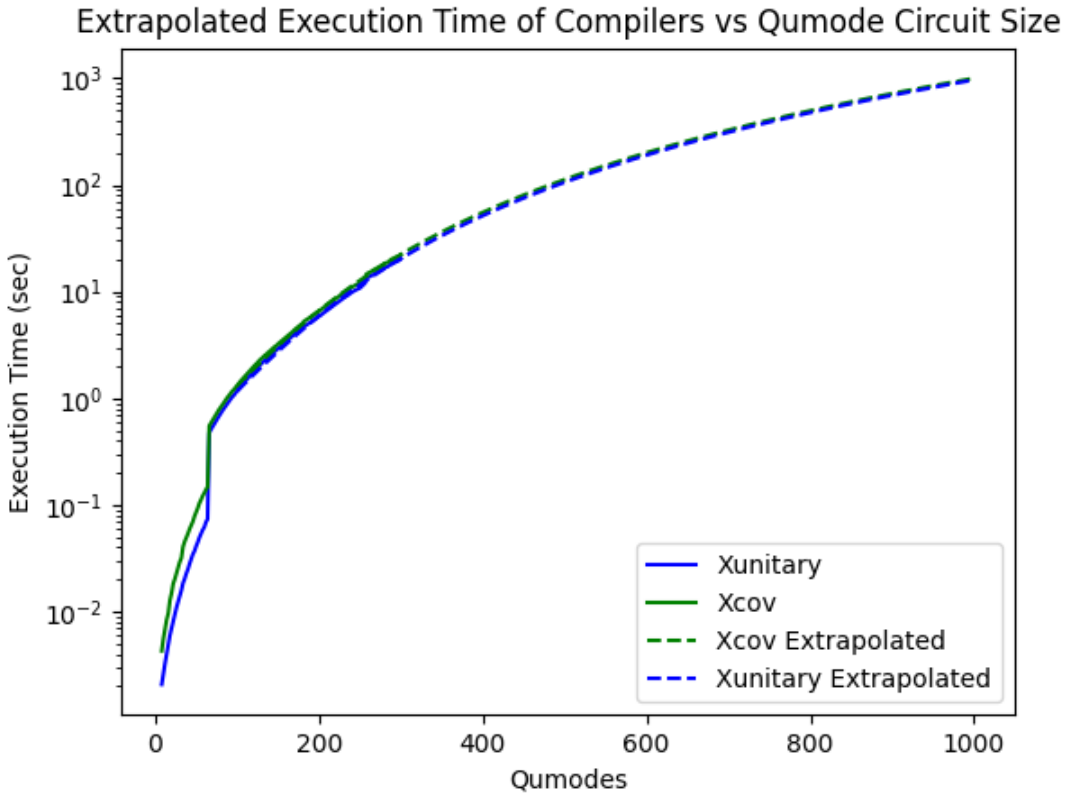


Figure 4.8: Extrapolated execution time for Xunitary and Xcov compilers.

Consequently, software profiling was used to find the operations that drive the compilation time, as well as the cause of the spike in compilation time seen in the timing analysis results. The *cProfile* python module was used to achieve this. This module allows us to create a *profile* – set of statistics that describes how often and for how long various parts take of the program executed – that can be formatted into reports. In this case the reports seen on Figure 4.9 and Figure 4.10 shows 30 function calls that had the longest cumulative time (time spent from invocation till exit) for compiling a 200 qumode SF program in Xcov and Xunitary, respectively. Using these results it is apparent that the compilation spent most of its time decomposing the interferometer unitary into Mach-Zehnder and rotation gates. Specifically for Xcov, the Mach-Zehnder decomposition calls took 4.4 seconds out of the 6.2 seconds for the whole compilation, which accounted for 70% of the overall compile time. As expected

the same behaviour can be seen for Xunitary in Figure 4.10, due to both compilers utilizing this decomposition function in their compilation steps, see Figure 4.5 and Figure 4.4.

```
Running Timing test for 200 qumodes
Tue Apr 27 14:20:15 2021    Xcov

         1299145 function calls (1165717 primitive calls) in 6.961 seconds

   Ordered by: cumulative time
   List reduced from 340 to 30 due to restriction <30>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    6.961    6.961 {built-in method builtins.exec}
        1    0.004    0.004    6.961    6.961 <string>:1(<module>)
        1    0.001    0.001    6.957    6.957 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\program.py:467(compile)
        1    0.007    0.007    6.954    6.954 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\compilers\xcov.py:87(compile)
        1    0.032    0.032    5.799    5.799 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\ops.py:2280(_decompose)
        1    0.036    0.036    5.735    5.735 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\decompositions.py:548(rectangular_symmetric)
        1    1.110    1.110    5.681    5.681 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\decompositions.py:495(rectangular_MZ)
     4950    3.259    0.001    4.399    0.001 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\decompositions.py:384(mach_zehnder)
     2500    0.007    0.000    2.291    0.001 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\decompositions.py:422(mach_zehnder_inv)
21063/15055    0.027    0.000    1.050    0.000 {built-in method numpy.core._multiarray_umath.implement_array_function}
     4952    0.009    0.000    0.809    0.000 <__array_function__ internals>:2(round_)
     4952    0.005    0.000    0.794    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\numpy\core\fromnumeric.py:3628(round_)
     4952    0.004    0.000    0.789    0.000 <__array_function__ internals>:2(around)
     4952    0.007    0.000    0.781    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\numpy\core\fromnumeric.py:3168(around)
     4961    0.006    0.000    0.774    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\numpy\core\fromnumeric.py:52(_wrapfunc)
     4952    0.766    0.000    0.766    0.000 {method 'round' of 'numpy.ndarray' objects}
        1    0.221    0.221    0.652    0.652 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\compilers\gaussian_unitary.py:104(compile)
    14963    0.021    0.000    0.396    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\numpy\core\numeric.py:2084(identity)
  101601/1    0.112    0.000    0.359    0.359 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\copy.py:132(deepcopy)
   10201/1    0.010    0.000    0.359    0.359 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\copy.py:210(_deepcopy_list)
        1    0.006    0.006    0.347    0.347 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\ops.py:2559(__init__)
        1    0.049    0.049    0.340    0.340 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\decompositions.py:707(bloch_messiah)
15251/5051    0.097    0.000    0.332    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\copy.py:268(_reconstruct)
10200/5050    0.019    0.000    0.299    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\copy.py:236(_deepcopy_dict)
    14964    0.113    0.000    0.299    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\numpy\lib\twodim_base.py:152(eye)
    14972    0.186    0.000    0.186    0.000 {built-in method numpy.zeros}
        2    0.009    0.004    0.174    0.087 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\decompositions.py:28(takagi)
        1    0.001    0.001    0.151    0.151 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\scipy\linalg\_decomp_polar.py:8(polar)
        1    0.146    0.146    0.147    0.147 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\scipy\linalg\decomp_svd.py:13(svd)
    14964    0.070    0.000    0.077    0.000 <frozen importlib._bootstrap>:997(_handle_fromlist)
```

Figure 4.9: Xcov compiler software profile for compilation of 200 qumode program. The red box outlines the function call that drives the cubic growth in compilation time.

Software profiling helped determine that the compilation time spike, in Figure 4.7, was caused by the interferometer decomposition function. In the case of Xcov, at 66 qumodes the interferometer decomposition function had a cumulative time of 0.471 seconds, while at 64 qumodes it only took 0.065 seconds. That corresponds to a 625% increase in computation time, which explains the spike behaviour in the plot shown in Figure 4.7.

```
Running Timing test for 200 qumodes
Tue Apr 27 14:03:43 2021    Xunitary

         1302369 function calls (1169032 primitive calls) in 6.263 seconds

   Ordered by: cumulative time
   List reduced from 247 to 30 due to restriction <30>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    6.263    6.263 {built-in method builtins.exec}
        1    0.004    0.004    6.263    6.263 <string>:1(<module>)
        1    0.000    0.000    6.259    6.259 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\program.py:467(compile)
        1    0.006    0.006    6.257    6.257 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\compilers\xunitary.py:100(compile)
        1    0.031    0.031    5.852    5.852 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\ops.py:2280(_decompose)
        1    0.035    0.035    5.789    5.789 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\decompositions.py:548(rectangular_symmetric)
        1    1.134    1.134    5.737    5.737 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\decompositions.py:495(rectangular_MZ)
     4950    3.271    0.001    4.428    0.001 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\decompositions.py:384(mach_zehnder)
     2500    0.007    0.000    2.248    0.001 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\decompositions.py:422(mach_zehnder_inv)
20998/15011    0.024    0.000    0.888    0.000 {built-in method numpy.core._multiarray_umath.implement_array_function}
     4950    0.008    0.000    0.820    0.000 <__array_function__ internals>:2(round_)
     4950    0.006    0.000    0.805    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\numpy\core\fromnumeric.py:3628(round_)
     4950    0.004    0.000    0.799    0.000 <__array_function__ internals>:2(around)
     4950    0.006    0.000    0.790    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\numpy\core\fromnumeric.py:3168(around)
     4950    0.005    0.000    0.784    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\numpy\core\fromnumeric.py:52(_wrapfunc)
     4950    0.776    0.000    0.776    0.000 {method 'round' of 'numpy.ndarray' objects}
 101601/1    0.113    0.000    0.359    0.359 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\copy.py:132(deepcopy)
  10201/1    0.010    0.000    0.359    0.359 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\copy.py:210(_deepcopy_list)
    14858    0.020    0.000    0.341    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\numpy\core\numeric.py:2084(identity)
15251/5051    0.097    0.000    0.333    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\copy.py:268(_reconstruct)
10200/5050    0.019    0.000    0.300    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\copy.py:236(_deepcopy_dict)
    14858    0.053    0.000    0.242    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\numpy\lib\twodim_base.py:152(eye)
    14859    0.189    0.000    0.189    0.000 {built-in method numpy.zeros}
    14858    0.072    0.000    0.078    0.000 <frozen importlib._bootstrap>:997(_handle_fromlist)
     9850    0.009    0.000    0.070    0.000 <__array_function__ internals>:2(angle)
     2500    0.041    0.000    0.068    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\decompositions.py:429(nullMZi)
     2450    0.040    0.000    0.067    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\strawberryfields\decompositions.py:462(nullMZ)
     9850    0.025    0.000    0.045    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\site-packages\numpy\lib\function_base.py:1430(angle)
     2501    0.032    0.000    0.032    0.000 {method 'conj' of 'numpy.ndarray' objects}
    30500    0.005    0.000    0.029    0.000 C:\Users\fxr8250\Anaconda3\envs\quantumpy\lib\copy.py:273(<genexpr>)
```

Figure 4.10: Xcov compiler software profile for compilation of 200 qumode program. The red box outlines the function call that drives the cubic growth in compilation time.

4.2 Gaussian Merge Compiler

As shown in the previous sections, the simulation of Gaussian states using classical computers is a resource and compute intensive task, which limits the size of programs that can be run. A way to alleviate this issue is to reduce the amount of separate Gaussian operations in a SF program to be processed in simulation by merging all possible Gaussian operations together into their symplectic matrices. A new SF compiler function was created for this purpose, called Gaussian Merge. The compiler utilizes Direct Acyclic Graphs (DAGs) to determine which operations can be merged and then are merged using the Gaussian Unitary compiler. The resulting program will contain only non-Gaussian, Gaussian Transforms (symplectic matrix), and Displacement operations. Currently, there is no performance benefit of using this compiler as the SF simulators do not support Gaussian Transform (GT) operations as primitives, instead they get decomposed back into supported operations. The benefit of this compiler is simplifying large hybrid circuits that allow for performance speedups once GT operations are supported. Once supported, a focus on optimizing GT in

simulators can introduce a overall speedup for any Gaussian circuit.

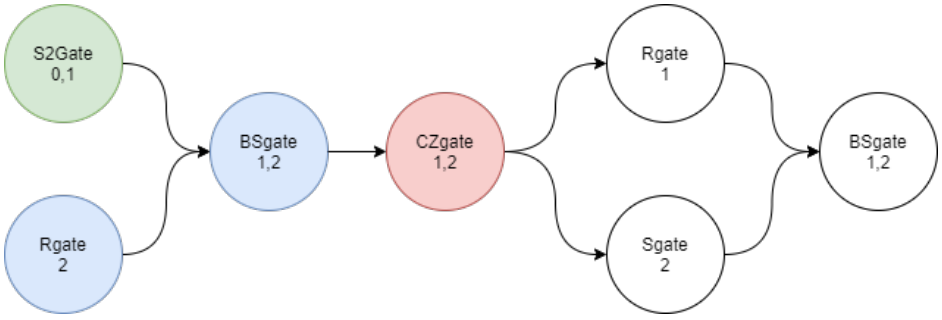
The program flow of the compiler can be seen in Figure 4.14. Each iteration of the compiler starts by picking a Gaussian operation. The compiler will then obtain all the Gaussian operations that preceded it, succeed it and happen in the same time frame, these are called valid Gaussian merge operations. The addition of non-Gaussian operations complicates the Gaussian operations that can be merged. A non-Gaussian operation being "sandwiched" between two Gaussian operations, prohibits the merge of those two Gaussian operations so the equality of the original and merged circuit is retained. This complicates the requirements of determining valid merge operations. Valid merged Gaussian operations are determined, in relation to the main Gaussian operation, by:

- Gaussian operations that succeed the main Gaussian operation, if they don't have non-Gaussian dependencies that operate on the same qumodes as the main Gaussian operation.

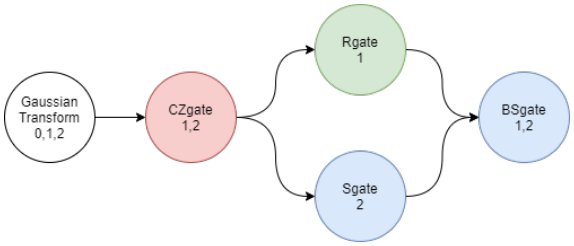
- Gaussian operations that are executed in the same time frame as the main Gaussian operation. i.e. operations that precede the successor operations of the main Gaussian operation.

- All Displacement gates that succeed the main Gaussian operation, and any other Displacement gates that follow them, recursion is used to achieve this.

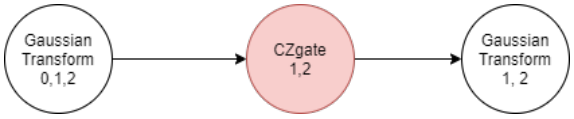
The code for determining valid merged operations can be seen in Listing 4.6. Furthermore, the logic to determine where to place edges between merged Gaussian Transform operations, successor and predecessor operations is shown in Listing 4.7. Non-Gaussian gates that succeed the original operation are merged with the logic shown in Listing 4.8. All the source code is open source and can be viewed in the strawberry fields github [23].



(a) The compiler first chooses a starting node, in this case the Sgate acting on qumodes 0 and 1. It then looks at the succeeding operations of the starting node – BSgate – and determines whether they are Gaussian or not. If they are Gaussian operations, they are merged in with the starting node. To maximize the amount of operations that are merged in one iteration, the compiler will take into account the predecessors of the succeeding operations, i.e. the Rgate. These operations, if Gaussian, can be merged with the starting node. In this example the starting node, S2gate, is merged with the Bsgate and Rgate into a Gaussian Transform (GT) operation. Once the operations are merged, the edges of the merged operations are passed over to the new GT operation, to ensure the correct order of the operations.



(b) The compiler arbitrarily chooses the next operation to merge, in this case the Rgate is chosen. While it does not have any Gaussian successor or predecessor operations, the compiler has the ability to merge Gaussian operations on the same graph level, i.e. the other predecessors of the successor of the Rgate in this case the Sgate was added by being predecessor to the BSgate. Same thing happens in the first step with the Rgate on qumode 2.



(c) Resulting DAG once the Gaussian merge compiler merges all Gaussian operations. The program in this DAG is mathematically identical to the initial program, but with a reduced amount of operations.

Figure 4.11: The steps a DAG, of a simple SF program, takes when being compiled using the Gaussian Merge compiler. The red nodes depict non-Gaussian operations. In each step, the green node outlines the Gaussian operation that the compiler uses to start its merge, while the blue nodes note the operations that will be merged with the green node. Lastly, the white nodes are operations that are untouched in that step of the compiler.

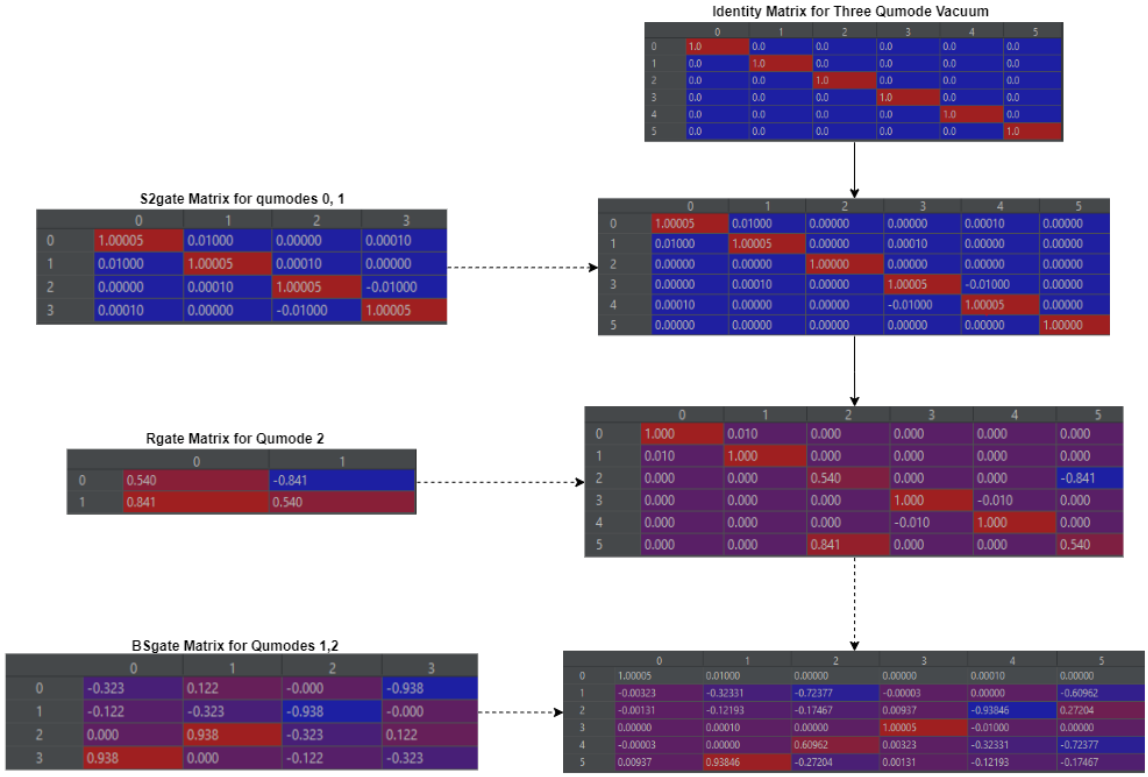


Figure 4.12: The Gaussian Unitary compiler merges multiple Gaussian operations into a single symplectic matrix. This is achieved by sequentially applying the symplectic matrix of a single gate operation (i.e Rgate) to the symplectic matrix of the overall system. The matrices on the right show the transformations the symplectic matrix takes when applying single or two-mode gates on a three qumode system. The resulting matrix corresponds to a unitary transformation identical to that of the three gates applied sequentially in a SF program.

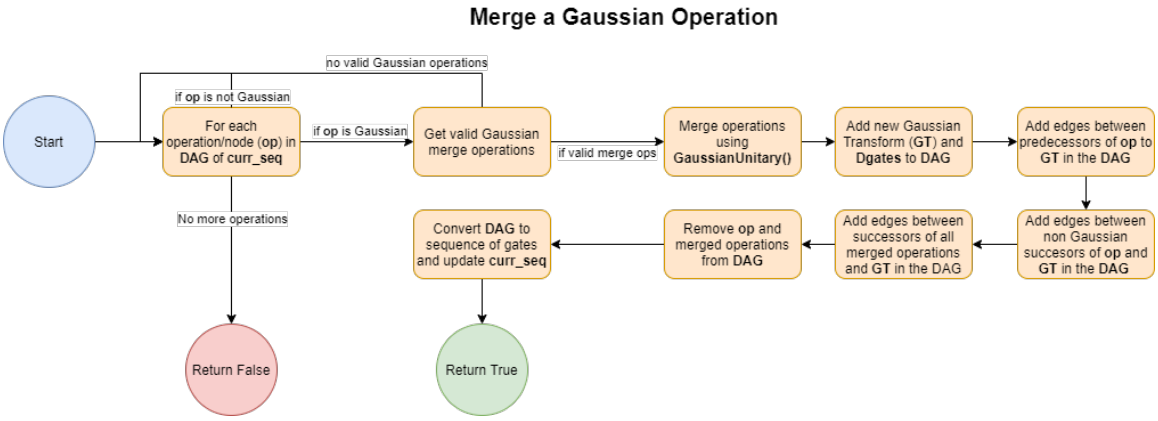


Figure 4.13: Program flow for merging a single Gaussian operation (**op**) with its Gaussian neighbors. This method returns True if Gaussian operations were merged and False if no Gaussian operations can be merged. The current sequence (**curr_seq**) variable holds the SF sequence of gates that is updated in each iteration of the merge a Gaussian operation function. Once no other Gaussian operations can be merged, this variable holds the resulting program of the compiler. The sequence of gates can be easily converted to its DAG representation using methods implemented in SF.

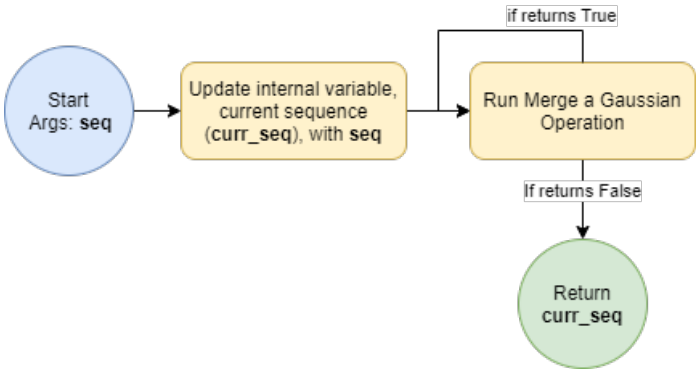


Figure 4.14: Program flow of compilation using Gaussian Merge compiler. The flow here outlines that the compilation is iterative. A group of Gaussian operations are merged in each iteration. The compilation is done when there are no more Gaussian operations to merge.

```
def get_valid_gaussian_merge_ops(self, op):
    """
    Obtains the valid gaussian operations that can be merged with op
    at the current DAG configuration.
    """
    merged_gaussian_ops = []
    for successor_op in self.DAG.successors(op):
        # If successor operation is a Gaussian operation append to
    list for merging
        if get_op_name(successor_op) in self.gaussian_ops:
            merged_gaussian_ops.append(successor_op)
            # Get displacement operations (recursively) that follow
    after successor operation
            d_gate_successors = self.recursive_d_gate_successors(
    successor_op)
            if d_gate_successors:
                merged_gaussian_ops += d_gate_successors

    # Add gaussian operations that should be executed at the same
    time (same time frame) as op
    # E.X Rgate|q[0] Rgate|q[1] -> BS|q[0]q[1]. Adds Rgate|q[1] if
    Rgate|q[0] is the op.
    for gaussian_op in merged_gaussian_ops:
        for predecessor in self.DAG.predecessors(gaussian_op):
            if predecessor is op:
                continue
            if (
                predecessor not in merged_gaussian_ops
                and get_op_name(predecessor) in self.gaussian_ops
            ):
                if self.valid_prepend_op_addition(op, predecessor,
    merged_gaussian_ops):
                    merged_gaussian_ops.append(predecessor)
```

```
27
28      merged_gaussian_ops = self.remove_invalid_operations(op,
    merged_gaussian_ops)
29      return merged_gaussian_ops
```

Listing 4.6: Function that returns a list of valid Gaussian operations that can be merged with **op**

```
30 def add_gaussian_pre_and_succ_gates(
31      self, gaussian_transform, merged_gaussian_ops,
    displacement_mapping
32 ):
33      """
34      Updated DAG by adding edges between gaussian transform/
    displacement operations to unmerged gaussian operations.
35
36      Displacement mapping is a dictionary whose key is the qumode
    being operated by the displacement gate, which is its value. i.e.
     {0: Dgate(1.5) | q[0]}
37      """
38      successor_operations_added = []
39      for gaussian_op in merged_gaussian_ops:
40          # Need special logic if there are displacement gates
41          if displacement_mapping:
42              for successor_op in self.DAG.successors(gaussian_op):
43                  placed_edge = False
44                  successor_op_qumodes = get_qumodes_operated_upon(
    successor_op)
45                  for qumode in successor_op_qumodes:
46 # If displacement gate operates on the same qumodes as the non-
    gaussian operation then don't add an edge. If register operated
    upon by successor operation has a displacement gate, add edge.
47                      if (qumode in displacement_mapping
48                          and qumode not in self.
```

```
    non_gaussian_qumodes_dependecy(successor_op)):
49                      # add edge between displacement gate and
    successor operation
50                      self.new_DAG.add_edge(displacement_mapping[
    qumode], successor_op)
51                      placed_edge = True
52 # If there is no displacement gate on qumode, add edge between GT
    operation and successor op
53                  if not placed_edge:
54                      self.new_DAG.add_edge(gaussian_transform[0],
    successor_op)
55                  successor_operations_added.append(successor_op)
56          else:
57 # If no displacement gates, just add edge between GT operation and
    successor operations
58              self.new_DAG.add_edges_from([(gaussian_transform[-1],
    post) for post in self.DAG.successors(gaussian_op)])
59
60              successor_operations_added += self.DAG.successors(
    gaussian_op)
61
62      for gaussian_op in merged_gaussian_ops:
63          # Append Predecessors to Gaussian Transform
64          for predecessor in self.DAG.predecessors(gaussian_op):
65              # Make sure adding the edge wont make a cycle
66              if predecessor not in successor_operations_added:
67                  self.new_DAG.add_edge(predecessor,
    gaussian_transform[0])
```

Listing 4.7: Function that determines where to add edges between new Gaussian Transform operation successor and predecessor operations while retaining program functionality

```
68 def add_non_gaussian_successor_gates(
69     self, gaussian_transform, successors, displacement_mapping
70 ):
71 """
72 Updates the DAG by adding edges between new gaussian transform and
      non-gaussian operations
73 from original operations.
74 """
75     for successor_op in successors:
76         if get_op_name(successor_op) not in self.gaussian_ops:
77             # If there are no displacement gates.
78             # Add edges from it to successor gates if they act upon
      the same qumodes
79             if not displacement_mapping:
80                 # Add edge from gaussian transform to successor
      operation
81                 self.new_DAG.add_edge(gaussian_transform[0],
      successor_op)
```

Listing 4.8: Function that adds gates between merged gaussian transform operation and successor non-Gaussian gates

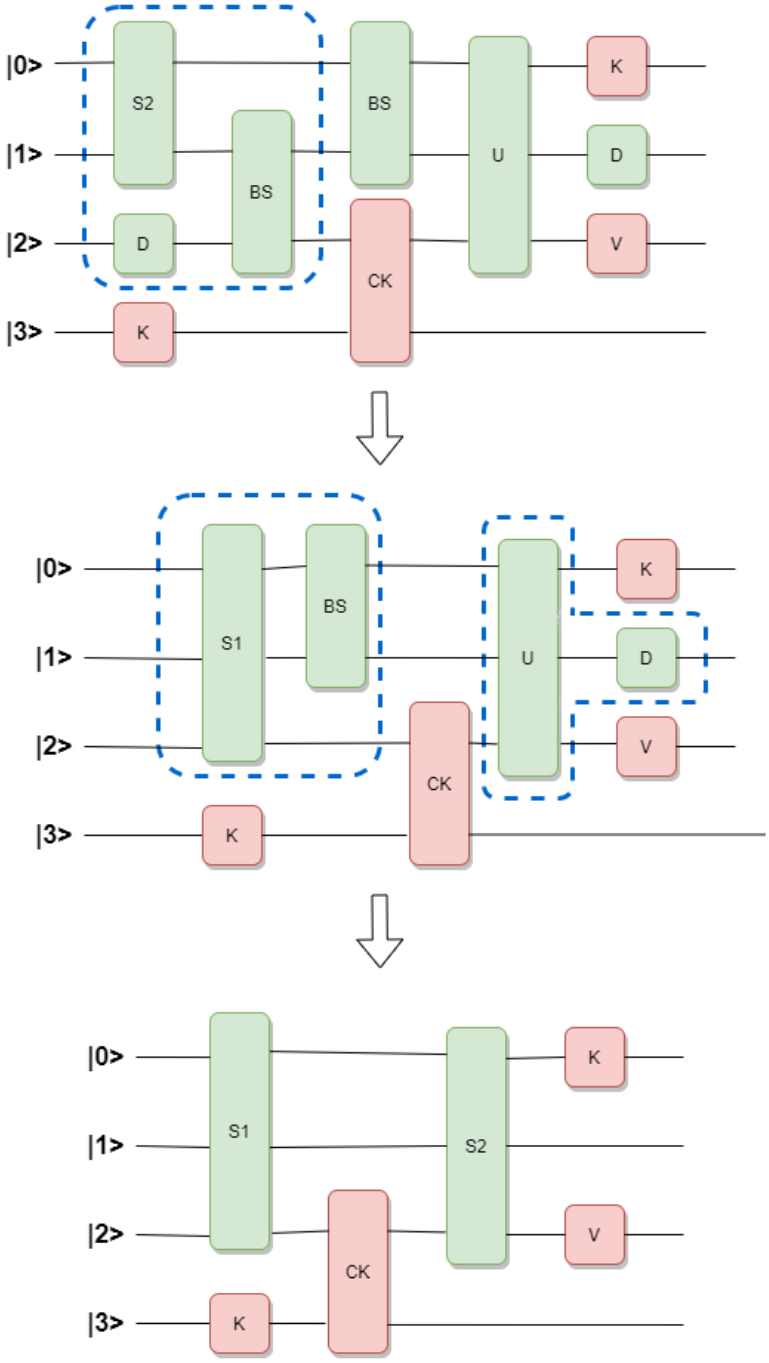


Figure 4.15: Gaussian Merge compilation step example of a four qumode SF program. The green and red operations represent Gaussian and non-Gaussian operations, respectively. Meanwhile, the blue dotted boxes depict the Gaussian operations that can and will be merged into a single Symplectic matrix operation.

4.2.0.1 Validation

Validation of the compiler involved creating random four qumode ket states that were operated upon by layers of Gaussian and non-Gaussian gates. Ket states can only be simulated in a fock backend, it means the simulation result is analogous to a 'count' based measurement. The resulting ket state can be easily compared with others to ensure equality of the Gaussian merge compiler.

```
82 # Test is parameterized depending on how many 'photons' are in each
       qumode in the initial ket state
83 pytest.mark.parametrize(
84     "init", [(1, 1, 1, 1), (0, 2, 1, 0), (0, 1, 1, 1), (0, 1, 0, 3),
       (0, 0, 0, 0)]
85 )
86 def test_complex(init):
87     qumodes = 4
88     cutoff_dim = 6
89     initial_state = np.zeros([cutoff_dim] * modes, dtype=complex)
90     # The ket below corresponds to a single photon going into each
       of the qumodes
91     initial_state[init] = 1
92
93     prog = sf.Program(qumodes)
94     # squeezing and displacement must be low enough that it doesn't
       distort the results
95     s_d_params = 0.01
96     with prog.context as q:
97         ops.Ket(initial_state) | q  # Initial state preparation
98         # Gaussian Layer
99         ops.S2gate(s_d_params, s_d_params) | (q[0], q[1])
100        ops.BSgate(1.9, 1.7) | (q[1], q[2])
101        ops.BSgate(0.9, 0.2) | (q[0], q[1])
102        # Non-Gaussian Layer
```

```
103         ops.Kgate(0.5) | q[3]
104         ops.CKgate(0.7) | (q[2], q[3])
105         # Gaussian Layer
106         ops.BSgate(1.0, 0.4) | (q[0], q[1])
107         ops.BSgate(2.0, 1.5) | (q[1], q[2])
108         ops.Dgate(s_d_params) | q[0]
109         ops.Dgate(s_d_params) | q[0]
110         ops.Sgate(s_d_params, s_d_params) | q[1]
111         # Non-Gaussian Layer
112         ops.Vgate(0.5) | q[2]
113
114     eng = sf.Engine("fock", backend_options={"cutoff_dim":
    cutoff_dim})
115     # Run simulation using normal compilation process
116     results_norm = eng.run(prog)
117     # Run simulation using gaussian merge compiler
118     prog_merged = prog.compile(compiler="gaussian_merge")
119     results_merged = eng.run(prog_merged)
120     ket = results_norm.state.ket()
121     ket_merged = results_merged.state.ket()
122     # Ensure resulting ket states are identical using conjugate of
    resulting state
123     assert np.allclose(np.abs(np.sum(np.conj(ket) * ket_merged)), 1)
```

Listing 4.9: Unit test that ensures SF programs compiled with Gaussian merge retains their functionality. Involves the creation of a random ket states simulated in a fock backend.

4.2.0.2 Simulated Results

To test the impact of the Gaussian Merge compiler in circuits which have Gaussian and non-Gaussian gates are combined, we performed the numerical simulation of several circuits at random, with the goal of comparing the final number of gates after merging the Gaussian layers. The total percentage of Gaussian gates was varied

from 0% (100% non-Gaussian gates) to 95% (5% non-Gaussian gates). We define a layer as a group of Gaussian gates that can be merged together. Our experiments generated, 20 possible test cases with a combination random numbers of Gaussian layers interleaved with non-Gaussian operators. The maximum number of layer was set to be at most 20% of the number of gates, under the assumption that a large number of gates will be acting on a large number of qumodes. This implies that the circuit is expected to hold high levels of parallelism, where gates can be scheduled to act simultaneously on different sets of qumodes. The average final count of gates after the merge is depicted in Figure 4.16. For example, for a 40% Gaussian gate count, the number of gates drops from 100 to 66.1 on average across the different layer configuration.

As it can be seen, the final average number of gates varies almost linearly with the number of Gaussian gates in the circuit. The degree of simplification varies greatly depending on the number layer in the design. A design with a single Gaussian layer where all the Gaussian gates are placed, will be able to reduce all of them to one operator, while a design where the Gaussian gates are scattered across multiple layers will only see local reductions within each layer. In the best case scenario, with 95% Gaussian gates, the number of final gates gets reduced to 13% of the original count on average. In general, we can expect a reduction of the total number of gates slightly below the total percentage number of Gaussian gates on average.

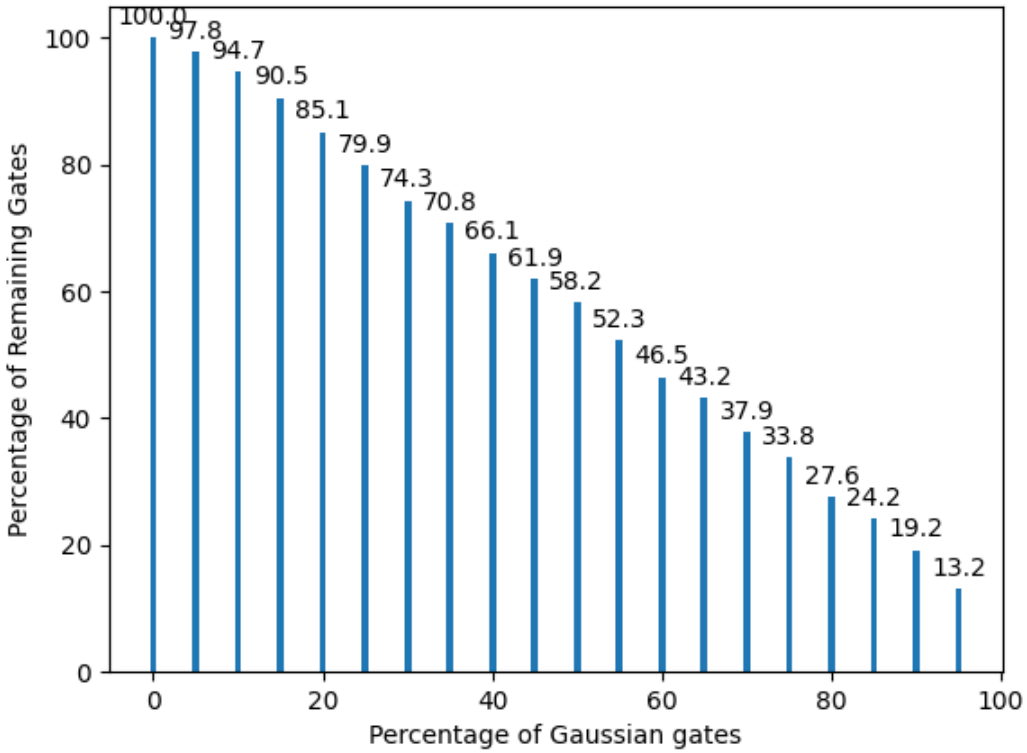


Figure 4.16: Final merged gate percentage out of an initial count of 100 gates

Chapter 5

Conclusion

Photon-based quantum computers are an emerging quantum technology whose aim is to create accessible universal quantum computers. As a stepping stone for this goal, Xanadu, has developed photonic devices that can be easily accessed through the cloud. These devices utilize Gaussian Boson Sampling (GBS) technology to perform calculations. Recent experiments show that they are likely candidates to prove quantum supremacy. It is important to analyze the software stack – specifically compilers – for these devices as they grow in size and complexity. For this reason we analyzed the steps of the Strawberry Fields compilers – the only full-stack library targeting photonic computers – and performed software profiling on them. It was found that these compilers were designed in a hierarchical structure. Specialized compilers inherit from basic compilers and implement more complex logic depending on the use of the compiler. For example, X8 photonic hardware is based on GBS photonic circuits so the SF programs targeting the hardware must match a certain structure that is enforced by GBS SF compiler. Thus, the X8 compilers inherit from the GBS compiler – creating the hierarchy – and add logic on top of it that is specific to the requirements of X8 hardware. Furthermore, the compilers targeting the X8 hardware were profiled with scalability in mind. The results showed that SF programs with $\tilde{1}000$ qumodes would theoretically take 15 minutes to compile when targeting photonic hardware. In addition, the compilation time difference between the two compilers profiled, Xu-

nitary and Xcov, which was found to be negligible. Most of the compilation time for both compilers was spent on decomposition of the interferometer operations into the primitive operations of X8 hardware. We can conclude that optimization of decomposition operations should be a priority if there's a need to speedup compilation times of X8 compilers. In addition, a new compilation step was proposed to aid with the processing of mixed Gaussian-non-Gaussian circuit designs, demonstrating a significant improvement on the complexity of these circuits, measured in number of gates. The gate reduction varied between 2% and 87%, depending on the original Gaussian gate percentage in the circuit, the layer distribution of these. The compiler will be specifically useful at reducing program overhead as the SF environment matures and the use of large hybrid programs become commonplace. Continuation of this work – specifically for Gaussian Merge compiler – would include optimization of Gaussian Transform operations in simulator backends, and research into practical uses for large hybrid SF programs where the Gaussian Merge compiler can play a role in reducing gate counts.

Bibliography

[1] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019. [Online]. Available: http://dx.doi.org/10.1038/s41586-019-1666-5

[2] N. Killoran, J. Izaac, N. Quesada, V. Bergholm, M. Amy, and C. Weedbrook, "Strawberry Fields: A Software Platform for Photonic Quantum Computing," *Quantum*, vol. 3, p. 129, 2019.

[3] M. Vogel, *Quantum Computation and Quantum Information, by M.A. Nielsen and I.L. Chuang*, 2011, vol. 52, no. 6.

[4] A. Lund, A. Laing, S. Rahimi-Keshari, T. Rudolph, J. O'Brien, and T. Ralph, "Boson sampling from a gaussian state," *Physical Review Letters*, vol. 113, no. 10, Sep 2014. [Online]. Available: http://dx.doi.org/10.1103/PhysRevLett.113.100502

[5] B. T. Gard, K. R. Motes, J. P. Olson, P. P. Rohde, and J. P. Dowling, "An introduction to boson-sampling," *From Atomic to Mesoscale*, p. 167–192, Jun 2015. [Online]. Available: http://dx.doi.org/10.1142/9789814678704_0008

[6] R. F. Streater, "Canonical quantization," *Communications in Mathematical Physics*, vol. 2, no. 1, pp. 354–374, 1966.

[7] R. P. Feynman, "Quantum mechanical computers," pp. 507–531, 1986.

[8] S. J. Lomonaco, "Shor's quantum factoring algorithm," *Proceedings of Symposia in Applied Mathematics Quantum Computation*, p. 161–179, 2002.

[9] L. K. Grover, "A fast quantum mechanical algorithm for database search," *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC 96*, 1996.

[10] L. Hales and S. Hallgren, "An improved quantum fourier transform algorithm and applications," *Proceedings 41st Annual Symposium on Foundations of Computer Science*.

[11] T. R. Bromley, J. M. Arrazola, S. Jahangiri, J. Izaac, N. Quesada, A. Delgado Gran, M. Schuld, J. Swinarton, Z. Zabaneh, and N. Killoran, "Applications of near-term photonic quantum computers: software and algorithms," *Quantum Science and Technology*, 2020.

[12] H. K. Lau and M. B. Plenio, "Universal Quantum Computing with Arbitrary Continuous-Variable Encoding," *Physical Review Letters*, vol. 117, no. 10, pp. 1–5, 2016.

[13] R. Dandliker, "Concept of modes in optics and photonics," in *PROCEEDINGS-SPIE THE INTERNATIONAL SOCIETY FOR OPTICAL ENGINEERING*. International Society for Optical Engineering; 1999, 1999, pp. 193–199.

[14] B. T. Gard, K. R. Motes, J. P. Olson, P. P. Rohde, and J. P. Dowling, "An introduction to boson-sampling," in *From atomic to mesoscale: The role of quantum coherence in systems of various complexities*. World Scientific, 2015, pp. 167–192.

[15] H. J. Ryser, "Combinatorial mathematics. the carus mathematical monographs, no. 14. published by the mathematical association of america; distributed by john wiley and sons," *Inc., New York*, 1963.

[16] C. S. Hamilton, R. Kruse, L. Sansoni, S. Barkhofen, C. Silberhorn, and I. Jex, "Gaussian boson sampling," *Physical review letters*, vol. 119, no. 17, p. 170501, 2017.

[17] D. J. Brod, E. F. Galvão, A. Crespi, R. Osellame, and N. Spagnolo, "Photonic implementation of boson sampling: a review," *Advanced Photonics*, vol. 1, no. 03, p. 1, 2019.

[18] W. R. Clements, P. C. Humphreys, B. J. Metcalf, W. S. Kolthammer, and I. A. Walsmley, "Optimal design for universal multiport interferometers," *Optica*, vol. 3, no. 12, p. 1460, 2016.

[19] J. E. Bourassa, R. N. Alexander, M. Vasmer, A. Patil, I. Tzitrin, T. Matsuura, D. Su, B. Q. Baragiola, S. Guha, G. Dauphinais, K. K. Sabapathy, N. C. Menicucci, and I. Dhand, "Blueprint for a scalable photonic fault-tolerant quantum computer," *arXiv*, pp. 1–32, 2020.

[20] D. Gottesman, A. Kitaev, and J. Preskill, "Encoding a qubit in an oscillator," *Physical Review A. Atomic, Molecular, and Optical Physics*, vol. 64, no. 1, pp. 123 101–1 231 021, 2001.

[21] H.-S. Zhong, H. Wang, Y.-H. Deng, M.-C. Chen, L.-C. Peng, Y.-H. Luo, J. Qin, D. Wu, X. Ding, Y. Hu *et al.*, "Quantum computational advantage using photons," *Science*, vol. 370, no. 6523, pp. 1460–1463, 2020.

[22] L. Chakhmakhchyan and N. J. Cerf, "Simulating arbitrary gaussian circuits with linear optics," *Physical Review A*, vol. 98, no. 6, p. 062314, 2018.

[23] XanaduAI, "strawberryfields/gaussian_merge.py at master · xanaduai/strawberryfields," Jun 2021. [Online]. Available: https://github.com/XanaduAI/strawberryfields/blob/master/strawberryfields/compilers/gaussian_merge.py