Cryogenic Control Compiler Optimization Formalism

1 Notation and Definitions

Given a quantum program, we wish to compile selectively to target parallelism and code size. Specifically, we wish to extract as much parallelism as possible while retaining the smallest possible code size.

Definition 1.1. A *Primitive Gate Operation* is an element of the set $\mathcal{G} = \{H, X, Z, CNOT, ...\}$. These are elementary quantum operations chosen from any particular set of universal operations. Shown here is one particular choice.

Definition 1.2. A *Function* is an ordered list of operations $(p_1, p_2, ...p_n)$. Each p_i is an *Operation*, which is a member of one of two sets: $\mathcal{F} = \{\alpha, \beta, \gamma...\}$ a list of functions, or $\mathcal{G} = \{H, X, CNOT, T, ...\}$ the set of primitive gate operations. The set of all *Operations* $\mathcal{O} = \mathcal{F} \cup \mathcal{G}$.

Definition 1.3. An input program P is itself a *Function*, with a corresponding $P_\mathcal{F}$, a list of functions used within the program. Each element $p_i \in P$ is a member of either $\mathcal{G}$ or $P_\mathcal{F}$. Each element of the set $P_\mathcal{F}$ is a function, comprised itself of elements of either $\mathcal{G}$ or $P_\mathcal{F}$.

Definition 1.4. When a function α is called several times in P, we can label each α with an index corresponding to the order of a particular function call within the list of function calls. More precisely, for any given function α, there may exist $\alpha_1, \alpha_2, ..., \alpha_k \in P$, where indices denote the position of each function call within this list. There may be other operations between function calls in P. These α_i are denoted *Function Instances*.

Definition 1.5. Given an input program P, *Code Size* (CS) of P is defined as:

$$CS_P = |P| + \sum_{\alpha \in P_\mathcal{F}} |\alpha| \tag{1}$$

This definition captures code size in the unit of instructions: each element of P is considered to be a single instruction, and every element of $P_\mathcal{F}$ is considered as a list of instructions. The total code size is the number of instructions in the main input program P plus the total size of all of the functions called within the program.

Definition 1.6. Associated with each $p_i \in P$ is a corresponding list DEP_{p_i}, defined as a list of operations upon which p_i is *dependent*. The list DEP_{p_i} is a list of operations $p_j \in P$, where $j < i$. Indices are considered as operation steps in the original, sequential program P.

2 Problem Statement

Given an input program P, we wish to schedule the program so as to maximize parallelism and minimize code size. Maximizing parallelism requires examining each instruction in as much scope as possible, so that a suitably parallel instruction may be found with which this instruction may be packed. Minimizing code size requires preserving as much modularity as is present in an original sequential version of a program.

Consider the subproblem restricted to a single function call $\alpha \in P$. Denote $|\alpha| = m$, $|P| = n$. Within P there may be several calls to function α, and for convenience we will label these with subscripts according to their call order: $\{\alpha_1, \alpha_2, ..., \alpha_{f_\alpha}\}$, where f_α is the *frequency* of α. Each α_i is a *Function Instance* of the original α, with potentially varied arguments. In order to optimize the trade off, a choice must be made about what to do with α.

2.1 Inlining Techniques

There are several inlining techniques that can be used, differing in their flexibility and sensitivity to various parameters.

For each function α, and for each instance of this function α_i, calculate a *parallelism factor*, a number indicating how many lines of this particular function instance could be parallelized, if it was to be inlined. For ease of notation, let *parallelism factor* be denoted as $\mathcal{P}$. Logistically, this can be calculated by, for each operation $p_j \in \alpha_i$, iterating through the instructions between the latest indexed operation in DEP_{p_i} to this particular instruction p_i. If a compatible instruction is discovered in this iteration, the $\mathcal{P}$ for α_i is incremented. If global function parallelism is desired, then:

$$\mathcal{P}_\alpha = \sum_{i=1}^{k} \mathcal{P}_{\alpha_i} \tag{2}$$

1. **All or Nothing Inlining**:
   If $\mathcal{P}_\alpha$ exceeds a threshold, inline all instances of α. The threshold can be shown to be:

   $$\mathcal{P}_\alpha \cdot V \geq f_\alpha \cdot |\alpha| - |\alpha| - f_\alpha \tag{3}$$

   where V is the overhead associated with parallelizing instructions in a VLIW fashion. If a VLIW instruction costs c single operation instructions, $V = \frac{1}{c}$.

2. **Complete Partial Inlining**:
   If a given contiguous subsequence of instructions $X \subset \alpha$ has a parallelism factor $\mathcal{P}_X$ exceeding a threshold, then inline that subsequence X for all function instances α_i, and clone the remaining instructions into a new function call α'. All function instances α_i will be replaced by the sequence X inlined, followed by the call(s) to α'_i. The threshold becomes:

   $$\mathcal{P}_X \cdot V \geq f_\alpha \cdot |X| + |X| \tag{4}$$

If the subsequence is contained in the center of a function, then two clones must be made α_i' and α_i'', corresponding to the preceding and successor blocks of operations in α. The threshold then becomes:

$$\mathcal{P}_X \cdot V \geq f_\alpha \cdot (|X| + 1) + |X| \tag{5}$$

3. **Selective Partial Inlining**:
   For each function instance α_i, identify the contiguous subsequence of operations X with the largest $\mathcal{P}_X$. Based upon global information about each X_{α_i}, decide upon an inlining method that may vary between α_i. Each decision to inline one particular X_{α_i} introduces $|\alpha| - |X_{\alpha_i}| - \mathcal{P}_{X_{\alpha_i}} \cdot V - 1$ lines to code size, as the corresponding $\alpha_i' = \alpha \backslash \{X_{\alpha_i}\}$ is added to $P_{\mathcal{F}}$, the list of functions used in program P.

Clearly (1) and (2) are special cases of (3). Now, we devise a procedure by which we can decide optimally which subsequences to inline, in which function instances throughout P.

3 Linear Program Formalism and Transformation

3.1 Graph Transformation

For each function α, let $|\alpha| = m$. We will construct one m-partite graph G. First construct a set of vertices $U = \{u_1, u_2, ..., u_m\}$. Now let $x_1, x_2, ..., x_m$ denote the first, second, ... , and last operations of α respectively. Additionally, for each x_i, construct a set of vertices $V_i = \{\alpha_1^i, \alpha_2^i, ..., \alpha_{f_\alpha}^i\}$. These vertices represent each function instance.

For each x_i, and for each α_j^i, calculate $\mathcal{P}_{x_i}^{\alpha_j}$ by iterating from instruction x_i until an instruction is reached that is not parallelizable. Next, we will add edges from V_i to U when a particular α_j contains a contiguously parallelizable subsequence X of size $|X| = \mathcal{P}_X$. Edges are added from α_j to the corresponding vertices in U, where the indices of u represent the size of the discovered sequence. So, edges $(\alpha_j, u_{|X|}), (\alpha_j, u_{|X|-1}), ..., (\alpha_j, u_1)$ will be added.

More precisely, add edges $(\alpha_j^i, u_{\mathcal{P}_{x_i}^{\alpha_j}})$, $(\alpha_j^i, u_{\mathcal{P}_{x_i}^{\alpha_j}-1})$, ... , (α_j^i, u_1).

The end result is a m-partite graph $G = (U \bigcup_{i=1}^{m} V_i, \bigcup_{i=1}^{m} E_i)$, where each $V_i = \{\alpha_1^i, \alpha_2^i, ..., \alpha_{f_\alpha}^i\}$, and each $E_i = \{(\alpha_j^i, u_k)\}$ is a set of edges from V_i to U.

3.2 Linear Program

We now introduce variables $X_e \in \{0, 1\}$, and place one such variable on every edge of G. We wish to maximize an objective function that incorporates parallelism weighted by code size. A viable objective function is:

$$\max \sum_{u \in U} \left(\sum_{e \in \delta(u)} X_e \right) \cdot i_u - i_u \tag{6}$$

where $\delta(u)$ denotes the set of incident edges to vertex u, and i_u denotes the index corresponding to vertex $u \in U$.

This objective function is subject to the following constraints:

$$\sum_{e \in \delta(\alpha_j^i)} X_e \leq 1 \quad \forall j \in [1, f_\alpha],\ i \in [1, m] \tag{1}$$

$$\sum_{e \in \delta(\alpha_j^i)} X_e = 0 \quad \text{if } i < k + l_{max} \quad \forall i \in [1, m],\ j \in [1, f_\alpha],\ k < i \tag{2}$$

where

$$l_{max} = \max_l \{X_e = 1, e = (\alpha_j^k, u_l), e \in E_k\} \tag{7}$$

Constraint (1) enforces that each α_j^i only has a single subsequence selected with given starting location i. Constraint (2) enforces that subsequences chosen for a given α_i do not overlap with other choices.

3.3 Analysis: Complexity

With this graph transformation and encoding as a linear program (LP), the LP is formed over the total number of edges in the graph G:

$$|E| = |\bigcup_{i=1}^{m} E_i| \leq f_\alpha m^2 \tag{8}$$

As an LP, solving is linear in the number of input variables: $\mathcal{O}(f_\alpha m^2)$.

The construction of the graph is somewhat compute intensive. For each α_i, and for each starting location $x \in \alpha$, a parallelism factor is calculated. For function instances that are completely parallelizable, this costs $m + (m-1) + ... + 1 = \frac{m(m+1)}{2}$ iterations with constant work per iteration to determine compatibility of instructions. This is performed f_α times, resulting in complexity of $\mathcal{O}(f_\alpha \frac{m(m+1)}{2})$.

These combine for overall complexity of $\mathcal{O}(f_\alpha m^2)$.