Cryogenic Control Compiler Optimization Formalism

1 Notation and Definitions

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 a partially 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. If $p_i \in \mathcal{F}$, then p_i is a function call, otherwise $p_i \in \mathcal{G}$ so p_i is a primitive gate operation. The set of all *operations* $\mathcal{O} = \mathcal{F} \cup \mathcal{G}$.

Definition 1.3. For each function $f \in \mathcal{F}$, define $P_{\mathcal{F}}$, a list of function calls in f:

$$P_{\mathcal{F}} = \{p \in f \mid p \in \mathcal{F}\} \tag{1}$$

Definition 1.4. 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 $P_{\mathcal{F}}$.

Definition 1.5. When a function α is called several times in a function β, we can label each α with an index corresponding to the tree depth of the function call within the dependency DAG of the function β. More precisely, for any given function α, there may exist $\alpha_1, \alpha_2, ..., \alpha_k \in \beta$, where indices denote the tree depth corresponding to each α_i-node in the dependency DAG. We also define $f_\alpha = k = \max_i\{\alpha_i\}$ as the *frequency* of function α. There may be other operations between function calls in β. These α_i are denoted *Function Instances.*

Definition 1.6. Given an input program P, *Code Size* (CS) of P is defined as:

$$CS_P = |P| + \sum_{\alpha \in P_{\mathcal{F}}} |\alpha| \tag{2}$$

Each element of P is considered to be a single operation, and every element of $P_{\mathcal{F}}$ is considered as a list of instructions.

Definition 1.7. $p_j \ \delta \ p_i \iff$ operation p_i *depends* on operation p_j.

Definition 1.8. $\text{Pred}(p_i) = \{p \in P \mid p \ \delta \ p_i\}$, and $\text{Succ}(p_i) = \{p \in P \mid p_i \ \delta \ p\}$. Predecessor and Successor sets, respesctively, contain the *immediate* predecessor or successor operation in the dependency DAG.

2 Problem Statement

Given an input program P, we wish to schedule the program with maximal parallelism and minimal code size. Maximizing parallelism requires examining each operation in as much scope as possible, so that compatible operations for parallelization may be discovered. 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. All of these techniques require a profiling of the *parallelism factor* of particular functions.

2.1.1 Parallelism Factor

For each function α, and for each call site of this function α_i the *parallelism factor* indicates how many lines of this particular function call site could be parallelized, if it was to be inlined. For ease of notation, let *parallelism factor* of *function call site* α_i be denoted as $\mathcal{P}_{\alpha_i}$. The *parallelism factor* of a function α is the summation of *parallelism factor*s of each call site of α:

$$\mathcal{P}_\alpha = \sum_{i=1}^{k} \mathcal{P}_{\alpha_i} \qquad (3)$$

To calculate this parallelism factor, a variant of the canonical As-Soon-As-Possible (ASAP) scheduling technique and the linearized List Scheduling technique can be employed. Given a function P, a function call site α_i, the dependence graphs of P and α_i (external and internal DAG, respectively), and the sets $\text{Pred}(p)$ and $\text{Succ}(p)$ for all $p \in \alpha_i$, see algorithm 1.

Note three important characteristics of profiling.

1. The algorithm does not perform *local basic block operation scheduling.* Greedy list scheduling is sufficient to perform such a task, and the algorithm is intended *only* to optimize across procedure boundaries. The scope of the program is that of *partial inlining*, and therefore is exempt from the task of such local basic block optimizations. It is important to realize that scheduling the primitive gate operations of the functions under analysis is assumed to have already been performed.

2. The profiling is performing a task equivalent to inlining the function call α_i and performing standard ASAP scheduling on the newly modified basic block.

3. The schedules output by *local basic block operation scheduling* are compatible with *all* inlining decisions. No schedule output from performing list scheduling on primitive gate operations will preclude any inlining decision, or will alter information available during parallelism profiling. This is due to the *MIMD* assumption, that all operations are capable of being performed during a single operation time step, and that list scheduling preserves precedence relations among scheduled operations.

Algorithm 1 ASAP Scheduling Variation for Parallelism Profiling, from [Aik16]

```
**Inline** $\alpha_i$ and merge dependency graphs
L $\leftarrow$ [ ]
**for** each starting operation $\boldsymbol{p}_k \in \alpha_i[1, |\alpha|]$ :
  $\mathcal{P}_k \leftarrow 0$
  **for** each operation $p_j \in \alpha_i[k, |\alpha|]$ do:
    $l_0(p_j) \leftarrow j$ where $j$ is index of $p$ in prescribed sequential order
    $l(p_j) \leftarrow 1$
  **endfor**
  **do**
    **for** each operation $p_j$, $j \in [k, |\alpha|]$ do:
      **for** each operation $s \in$ Pred$(p_j)$ do:
        $l(p_j) \leftarrow \max\{l(p_j), l(s)\}$
      **endfor**
      **if** $l(p) < l_0(p)$, then:
        $\mathcal{P}_k \leftarrow \mathcal{P}_k + 1$
      **else**
        break
      **endif**
    **endfor**
  **until** there are no changes to $l(\cdot)$
  append $\mathcal{P}_k$ to $L$
**endfor**
**return** L
```

The output of the algorithm is a set of values $\mathcal{P}_k$ $k \in [1, |\alpha|]$, which correspond to the maximal length of a parallelizable subsequence of operations of α_i beginning at instruction $\boldsymbol{p}_k$.

This algorithm can be modified to output the parallelism factor for an entire function call site. By omitting the outer loop (i.e. setting $k = 1$ and iterating a single time), and by omitting the **else** *break*, the output of the procedure is $\mathcal{P}_{\alpha_i}$, which is used to calculate $\mathcal{P}_\alpha$:

$$\mathcal{P}_\alpha = \sum_{i=1}^{f_\alpha} \mathcal{P}_{\alpha_i} \tag{4}$$

1. **All or Nothing Inlining**:
   If $\mathcal{P}_\alpha$ exceeds a threshold, inline all call sites of α. The threshold can be shown to be:
   $$\mathcal{P}_\alpha \cdot V \geq f_\alpha \cdot |\alpha| - |\alpha| - f_\alpha \tag{5}$$
   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**:
   Let a given contiguous subsequence of instructions $X_k \subset \alpha$ have a parallelism factor $\mathcal{P}_k$ corresponding to $\mathcal{P}_k$ output by algorithm 1. If this exceeds a threshold, then inline that subsequence X_k for all function call sites α_i, and clone the remaining instructions into a new function call α'. All function call sites α_i will be replaced by the subsequence X_k inlined, followed by the call(s) to α'_i. The threshold becomes:
   $$\mathcal{P}_k \cdot V \geq f_\alpha \cdot |X| + |X| \tag{6}$$

3. **Selective Partial Inlining**:
   For each function call site α_i, identify the contiguous subsequence of operations X_k with the largest $\mathcal{P}_k$. Denote this subsequence as $X_k^{\alpha_i}$. Based upon global information about each $X_k^{\alpha_j}$, decide upon an inlining method that may vary between call sites. Each decision to inline one particular $X_k^{\alpha_i}$ introduces $|\alpha| - |X_k^{\alpha_i}| - \mathcal{P}_{X_k^{\alpha_i}} \cdot V - 1$ lines to code size, as the corresponding $\alpha'_i = \alpha \backslash \{X_k^{\alpha_i}\}$ 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 call sites 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 call site.

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 $G = (U \bigcup\limits_{i=1}^{m} V_i, \bigcup_{i=1}^{m} E_i)$, 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:

$$\max_{X_E \in \{0,1\}^{|E|}} \sum_{j \in [1,f_\alpha]} \sum_{u \in U} \left(w_p \Big(\sum_{e \in E(V_j,U)} X_e \Big) \cdot i_u - w_c i_u \right) \tag{7}$$

where $E(V_i, U)$ denotes the set of edges between V_i and 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

$$l_{max} = \max_l \{X_e = 1, e = (\alpha_j^k, u_l), e \in E_k\} \tag{8}$$

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:

$$|E| = |\bigcup_{i=1}^{m} E_i| \leq f_\alpha m^2 \tag{9}$$

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 call sites 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 $\mathcal{O}(f_\alpha m^2)$. Noting that $|P| = n$ and $m < n$, combined with $f_\alpha < n$, we have a loose upper bound on overall complexity of $\mathcal{O}(n^3)$.

3.4 Analysis: Correctness

Consider a solution $S = \{e = (\alpha_j^i, u_k) \in E \mid X_e = 1,\ \forall i \in [1, m], j \in [1, f_\alpha\}$. For each edge $e \in S$, we can construct the corresponding inlined block $X = \{p_l \in \alpha \mid l \in [i, k]\}$ and the corresponding cloned function call(s) $\alpha'_j = \{p_l \in \alpha \mid l \in [1, i]\}, \alpha''_j = \{p_l \in \alpha \mid l \in (k, m]\}$.

Restricting attention to a single α_j, we see that:

$$e = (\alpha_j^i, u_k) \in S \implies e' = (\alpha_j^i, u_{k'}) \notin S,\ \forall k' \neq k \tag{10}$$

$$\implies e' = (\alpha_j^{i'}, u_{k'}) \notin S, \forall i' < i + k', k' \in [1, m] \tag{11}$$

Where (10) arises from constraint (1), and (11) is from constraint (2). (10) asserts that when a particular contiguously parallelizable block X is chosen from α_j beginning at starting location i (i.e. X chosen from α_j^i), then no other block is chosen for α_j^i. This effectively asserts that given a particular function call site, the choice of X from α_j^i is unique and singular, therefore a single line from α_j can never be inlined more than one time.

(11) asserts that when a block X is chosen from α_j^i, the next block that can possibly be chosen must begin with $i' > i + |X|$. This again prevents overlapping inlining assignments, and ensures that a single line from α_j can only be inlined a single time at most. However, more importantly this also allows for multiple contiguous blocks from α_j to be selected as long as they do not overlap, which will allow for an optimal solution.

As both of these conditions hold for all α_j, we can be sure that this procedure produces a correct set of blocks to inline, and blocks to clone and insert as new function calls.

References

[Aik16] Alex Aiken. *Instruction level parallelism.* Springer, 2016.