A Geometric Programming Framework for Optimal Multi-level Tiling

Lakshminaryanan Renganarayana
Computer Science Department
Colorado State University
Ft. Collins, CO 80523
ln@cs.colostate.edu

Sanjay Rajopadhye
Computer Science Department
Colorado State University
Ft. Collins, CO 80523
Sanjay.Rajopadhye@colostate.edu

ABSTRACT
Determining the optimal tile size—one that minimizes the execution time—is a classical problem in compilation and performance tuning of loop kernels. Designing a model of the overall execution time of a tiled loop nest is an important subproblem. Both problems become harder when tiling is applied at multiple levels. We present a framework for determining the optimal tile sizes for a fully permutable, perfectly nested, rectangular loop with uniform dependences. Our framework supports multiple levels of tiling and uses a BSP style high level model for estimating the overall execution time of a loop program. In our framework, the problem of determining the optimal tile sizes, subject to memory capacity and bandwidth constraints, is modeled as a geometric program and transformed into a convex optimization problem, which can be solved efficiently. The model is validated through experimental results obtained by running twenty loop programs for different levels of tiling and different program and tile parameters. Our framework is very general and can also be used to solve the optimal tile size problem with many other models of execution time.

1. INTRODUCTION

Achieving high performance on modern processors requires efficient utilization of the memory hierarchy. Program transformations like tiling try to match the characteristics of a memory hierarchy to the size and order of the data accesses [38, 18, 30, 36]. Multiple levels of tiling are required to match the multiple levels of memory [6, 23]. Determining the optimal tile parameters—those that minimize the execution time—is a fundamental problem. A model of the overall execution time of a tiled loop nest is an important subproblem. The non-linearity of the functions that describe fundamental properties of a tile, like computation/communication volume, memory footprint, access characteristics, etc., make the problem very hard, even under many simplifying assumptions.

In this paper we address the optimal multi-level tiling problem for fully permutable rectangular loop nests with uniform dependences. For such loops, orthogonal tiling, i.e., the tile boundaries are normal to the canonical axes, is legal and the simplest. Thus, the tile shape is fixed and the “only” values that need to be determined are the tile sizes in each of the dimensions. Despite these simplifying assumptions, the problem remains difficult.

Specifically, the multi-level tiling problem involves the determination of the optimal tiling parameters at each level. Usually the optimality is defined relative a cost function which models some aspect of the program execution, for example, number of cache misses, total CPU idle time, etc. Applying tiling at multiple levels with an independent goal or cost function at each level may lead to globally sub-optimal performance [23], since tiling choices from different levels interact with each other. A global metric, like overall execution time, that accounts for interactions from different levels should be used. To use such a global metric, we need a high level analytical model of the overall execution time of the tiled loop nest. Using such a high level model, the optimal tiling problem can be formulated as a numerical optimization problem.

Our contributions in this paper are:

- A high level analytical cost model, similar in
spirit to Valiant’s BSP model [33] (for parallel programs), for estimating the overall execution time of multi-tiled perfectly nested rectangular loops with uniform dependences. We also discuss how our cost model can be extended to include different processor/memory features and compiler optimizations. We present experimental results that validate our model.

- A formulation of the multi-level optimal tiling problem as an efficiently solvable (convex) optimization problem. Our formulation permits an arbitrary number of loops to be tiled and also arbitrary levels of tiling: $m$-levels of tiling of an $n$-depth loop.

- A geometric programming based unifying framework for optimal tiling. We present a single unifying framework that can be used for solving a variety of formulations of both single and multi-level optimal tiling problems. It requires certain architectural parameters (effective bandwidth at each level of the memory hierarchy), and certain program parameters (time to execute a single instance of the loop body) which can be readily determined/estimated at compile time. The generality of our framework is shown by directly mapping almost all of the single and multi-level cost functions and measure functions (that describe quantities like memory footprint, etc.) used in the literature, into functions that are suitable for our framework.

In the next section we present our high level analytical cost model. In Section 3 we formulate the optimal tiling problem, first for a single level, and then generalize it to multiple levels of tiling. In Section 4, we describe the Geometric Programming framework and show how the optimal multi-level tiling problem can be cast as a geometric program. Section 5 shows the generality of our framework and the extensibility of our cost model. In Section 6 we first describe the experimental setup used for the validation of our cost model and then present the results. We describe related work in Section 7 and conclude in Section 8 with some pointers to future work. All proofs can be found in Appendix A.

2. ANALYTICAL COST MODEL

In this section we develop a cost model for the total execution time of a tiled loop nest. First we discuss some fundamental measures that can be directly derived from the program. These are the computation and communication volume of a given tile and the loop overhead of a tiled program. The processor architecture dependent parameters, which we call architectural parameters, are described next. Then we show how the total execution time can be calculated using the fundamental measures and the architectural parameters. The concepts discussed in this section are in the context of single-level tiling. Extensions to multi-level tiling are discussed in Section 3.

2.1 Program and Tiling Model

We consider an $n$-dimensional loop nest with constant upper and lower bounds. The loop body contains statements with uniform dependences. Figure 1 shows our program and tiling model. We consider a compute bound loop nest, i.e., a loop nest in which the amount of computation is at least an order greater than the amount of memory operations. We consider orthogonal (or rectangular) loop tiling: tiling the loop nest with hyper-rectangles whose boundaries are orthogonal to the canonic axes, as are the iteration space boundaries. We assume that orthogonal loop tiling is valid for the given loop nest [38]. Figure 1.b shows the $2n$-dimensional tiled loop nest. Note that the tiled loops are fully permutable. For example, we can permute the $2n$ loops such that the $n^{th}$ and the $(n+1)^{st}$ loops together correspond to a single loop in the original program. In such a case, we fuse them together.

Let us consider the rectangular iteration space as given in Figure 1.a and a tiling of this with rectangular tiles as shown in Figure 1.b. The tile graph is the graph where each node represents a tile and each arc represents a dependency between tiles. In our case each node of the tile graph is a hyper-rectangle of size $t_1 \times t_2 \times \cdots \times t_n$ and the tile graph itself is a hyper-rectangle of size $n_1 \times n_2 \times \cdots \times n_n$, where $n_i = \frac{2^n}{t_i}$. It is well known [18, 2] that if the $t_i$’s are large as compared to the elements of the dependency vectors, then the dependencies between the tiles are unit vectors (or binary combinations thereof, which can be neglected for analysis purposes without loss of generality). In general, this implies that the feasible value of each $t_i$ is bounded from below by some constant. For the sake of notational simplicity, in this paper we assume that this is 1.

2.2 Fundamental measures

A single level of tiling is specified by a vector $\vec{t} = \langle t_1, \ldots, t_n \rangle$. Let $x$ denote the largest component of $\vec{t}$. The computation volume, $\Theta(\vec{t})$, of a tile is the amount of computation done in a given tile. The computation volume of the tile is the volume of the $n$-dimensional hyper-rectangle, $\Theta(\vec{t}) = \prod_{i=1}^{n} t_i$.

The communication volume of a tile, denoted by $\Delta(\vec{t})$, is the total amount of data that is input to and output from the tile. It is also called the footprint of a tile, $F(\vec{t})$. For an $n$-dimensional compute bound tile,
for $i_1 = 1$ to $N_1$
for $i_2 = 1$ to $N_2$
for $i_{n} = 1$ to $N_{n}$

$$A_1[i] = f(A_2[i + c_2], \ldots, A_k[i + c_k])$$


(a) Original program

(b) After tiling all the loops once

Figure 1: The program and tiling model. The loop body is shown as single assignment in order to clearly describe the uniform dependencies. However, the actual code is expected to be multiple assignment provided exact data dependences are constant. In fact, single assignment code is I/O bound and hence will not gain significantly from tiling.

the input and output are $\Theta(x^{n-1})$. We consider the case in which the input and output are of $\Theta(x^{n-1})$. Other cases when the input or output is smaller than $\Theta(x^{n-1})$ can be handled in a similar way. Since our tile graph has dependence vectors that correspond to unit vectors, the $\Theta(x^{n-1})$ input/output of a tile directly correspond to the $(n-1)$ dimensional facets of the tile, and a constant multiple of every facet contributes to the communication volume of a tile. The constant is determined by the dependence distances.

There are $n$ pairs of facets, and in rectangular tiling, each of these $n$ facets is involved in a communication. If $\Delta_i$ denotes the volume of the $i^{th}$ facet, then we have

$$\Delta_i = \sum_{j=1}^{n} a_{ij}$$

where $\Delta_i = \prod_{j=1, j \neq i}^{n} t_j = \frac{\Theta(v)}{t_i}$, and $a_{ij}$ is a constant that denotes distance along the $i^{th}$ facet that is involved in the communication and is determined by the longest $i^{th}$ dimension component of any dependence vector.

The values produced on the iterations at the tile boundaries need to be saved and later used when the corresponding neighboring tile is executed. The temporary storage required for this can be (and in our model is) accounted in $\Delta_i$ since it is of the same order. However, the address computation cost related to accessing these values stored in array variables could be of the order of any face of the tile, and hence can be expressed as a weighted combination of the faces of a tile. These faces could be of any dimension from $n - 1$ to 1. Let this weighted combination of faces be denoted by $\psi(t)$, and we have

$$\psi(t) = \sum_{i=1}^{F} \rho_i \phi_i$$

where $F$ is total number of faces of all dimensions $n - 1$ to 1, and $\rho_i$ is a non-negative scalar and $\phi_i$ is a face. Note that $\rho_i$ can be directly determined from the statements of the tiled loop nest that load-and-store the boundary values.

The Loop overhead of a loop is used to account for the cost of loop termination test and loop variable increment. It is of the order of the number of times the loop body is executed. An $n$-dimensional loop nest after one level of tiling will have $2n$ loops. For ease of notation, we consider the first $1, \ldots, n$ loops (which we call the inter-tile loops) and the $n + 1, \ldots, 2n$ (intra-tile) loops separately. The $i^{th}$ inter-tile loop, $i = 0, \ldots, n$, is executed exactly $N_i$ times for each instance of the surrounding loop indices. The total overhead for the set of $n$ inter-tile loops, denoted by $\lambda$, is $\lambda = \sum_{i=1}^{n} x_i$, where $x_i = \frac{N_{i-1} \times \cdots \times N_1}{t_{1} \times \cdots \times t_{i}$}.

$\lambda$ intra-tile loop, $i = n + 1, \ldots, 2n$, is executed $t_i$ times. The overhead of the set of $n$ intra-tile loops, denoted by $\lambda(t)$, is $\lambda(t) = \sum_{i=n+1}^{2n} y_i$, where $y_i = t_{i} \times \cdots \times t_i$.

2.3 Architectural parameters

We seek an abstraction of the architecture (processor and memory features) that is suitable for use in a cost model for tiling loop programs of our program model (Figure 1). We have identified the following parameters:

- $\alpha$ - cost of an iteration: this is the cost (in cycles) of executing an instance of the loop body
in the absence of cache misses.

- $\beta$ – bulk transfer rate: this is the cost (in cycles) for transferring a word of data between memory subsystems. We may have a family of $\beta$’s, one for each level of the memory hierarchy.

- $\eta$ – loop increment and test cost: this is the cost for incrementing a loop variable and checking its bounds.

- $\bar{\rho} = (\rho_1, \ldots, \rho_F)$ – boundary values load-store cost: the saving and loading of values at the tile boundaries involves access to array variables that hold these values. Each such access involves address computation and $\bar{\rho}$ represents the $F$-dimensional vector of this address computation costs, where $F$ is the total number of faces of a tile of dimension $n - 1$ to 1. The number of such accesses can be represented by the faces of the tile (see $\psi(\bar{t})$ in Section 2.2).

- $C^{(k)}$ – size (capacity) of cache at level $k$: The capacity (in bytes) of the cache at the level $k$.

Note, that the $\beta$’s, $\eta$, and the $\rho$’s are all architectural parameters and are known at compile time. The parameter $\alpha$ is a function of the machine and the program. However, we expect that it can be determined at compile time, either through profiling or as an estimate based on the instruction count of the loop body.

### 2.4 An analytical cost model

During the actual execution of a tiled loop nest many factors like ILP, reuse, cache hits/misses, etc., affect the running time, and often these factors are extremely difficult to model precisely. Our model, however, abstracts away from all these low level details and takes a high level view of the execution. As in Valiant’s nsp model for parallel computation [33], we view the execution of a single tile as atomic, and thus the execution time of the complete program is simply the number of tiles times the time to execute a single tile.

The execution time of a single tile is calculated as the sum of the time spent in computation, the time spent in communication (data transfer of the footprint of the tile), the load-store cost of the intermediate values, and the intra-tile loop overhead.

$$T_{tile}(\bar{t}) = \alpha \Theta(\bar{t}) + \beta \Delta(\bar{t}) + \psi(\bar{t}) + \eta \lambda(\bar{t})$$  \hspace{1cm} (2.1)

where $\alpha$ is the cost of executing an iteration, $\beta$ is the bulk transfer rate, $\psi(\bar{t})$, the cost of saving and using boundary iterations, is computed using $\bar{\rho}$, and $\eta$ is the cost of one loop increment and loop termination test.

The execution time of a tiled loop nest, $T_{base}(\bar{t})$ is the sum of the execution time for each tile times number of tiles and the inter-tile loop overhead, $\Delta(\bar{t})$ as defined below:

$$T_{base}(\bar{t}) = N T_{tile}(\bar{t}) + \eta \Lambda(\bar{t})$$ \hspace{1cm} (2.2)

where $N$ is the total number of tiles.

This abstraction is best justified if we assume that the footprint of a tile fits in the cache. This can be stated as $F(\bar{t}) \leq C$, where $F(\bar{t})$ is the memory footprint (c.f., Section 2.2) of a tile and $C$ is the cache capacity. Note that even without this constraint, the model describes a valid running time, though it will be optimistic since the effects of the cache misses are counted only once. However, we do expect that the optimal solution to the tile size problem will always satisfy this constraint.

### 3. Problem Formulation

Using the fundamental measures, architectural parameters and cost model discussed in the previous sections, we formulate the single level tiling problem.

#### 3.1 Single-level Problem Formulation

For a single level of tiling, the problem of choosing the tile sizes $t_i$, $i = 1, \ldots, n$ can be formulated as an optimization problem as follows. The objective function $T(\bar{t})$, is the total execution time and we want to minimize it subject to the following constraints: the memory footprint of the tile, $F(\bar{t})$, fits in the cache and the tile sizes ($t_i$’s) are positive. The generic problem is

$$\begin{align*}
\text{minimize} & \quad T(\bar{t}) \\
\text{subject to} & \quad F(\bar{t}) \leq C \\
& \quad t_i > 0 \quad \forall i = 1, \ldots, n \\
& \quad t_i \in \mathbb{Z} \quad \forall i = 1, \ldots, n
\end{align*}$$  \hspace{1cm} (3.1)

where, $C$ is the cache capacity. The choice of the exact function that describes $T(\bar{t})$ depends on the combination of processor features and compiler optimizations we want to model. For instance, one can choose $T_{base}(\bar{t})$ (Eqn. 2.2) or any of the extended cost functions $T_{oi_nbe}(\bar{t})$, $T_{hu_fetch}(\bar{t})$, or $T_{hi_opt}(\bar{t})$ (discussed in Section 5.1) and use in the place of $T(\bar{t})$ in (3.1) to obtain a concrete problem.

3For multi-level tiling, we will assume that the footprint of the innermost tile fits in the fastest cache, that of the next level fits in the next level cache, and so on.

4We can easily include a constraint like $t_i > d_i$ where $d_i$ is the maximum of the projections of the dependence vectors along the $i^{th}$ dimension. However for sake of notational simplicity we stick to $t_i \geq 1$. 

3.2 Multi-level Problem Formulation

Let us consider \( m \) levels of tiling of an \( n \) dimensional loop nest. In the tiled program there are \((m+1) \times n\) loops. Let \( T^{(j)} \) denote the execution time of the \( n \)-dimensional loop nest tiled \( j \) levels, i.e., the execution time of the innermost \((j+1) \times n\) loops. We can define \( T^{(j)} \) recursively as follows:

\[
T^{(j)} = \begin{cases} 
  j = 1 : N^{(1)}(\alpha \Theta(t^{(1)}) + \beta^{(1)} \Delta(t^{(1)}) + \psi(t^{(1)}) + \eta \lambda(t^{(1)})) + \eta \Lambda^{(1)} \\
  j > 1 : N^{(j)}(T^{(j-1)} + \beta^{(j)} \Delta(t^{(j)})) + \eta \Lambda^{(j)}
\end{cases}
\]

where,

- for \( j = 1, \ldots, m : N^{(j)} = \frac{t^{(j+1)}_i \times t^{(j+1)}_n}{t^{(j)}_i \times t^{(j)}_n} \), with \( t^{(m+1)}_i = N_i \), for \( i = 1, \ldots, n \).
- for \( j = 1, \ldots, m : \Delta(t^{(j)}) = \sum_{i=1}^{n} a^{(j)}_i \Delta_i(t^{(j)}) \), with \( \Delta_i(t^{(j)}) = \prod_{k=1, k \neq i}^{n} t^{(j+1)}_k \), for \( i = 1, \ldots, n \).
- for \( j = 1, \ldots, m : \Lambda^{(j)} = \sum_{i=1}^{n} \prod_{k=1}^{j-1} t^{(j+1)}_k \), with \( t^{(m+1)}_i = N_i \), for \( l = 1, \ldots, n \).
- for \( j = 1, \ldots, m : \beta^{(j)} \) is the bulk transfer rate for moving a word of data from a memory at level \( j+1 \) into a memory at level \( j \).
- for \( j = 1, \ldots, m : \eta^{(j)} = \eta \), since the loop variable increment and termination check cost is the same for every loop at every level.

The quantities related to the cost of execution of actual statements are relevant only at the innermost level of tiling and hence contribute to the execution time of the innermost level \( T^{(1)} \). These quantities are: \( \alpha \Theta(t^{(1)}) \), the computation cost at level 1 and \( \psi(t^{(1)}) \), the load-store cost at level 1. Also note that the (inner-most) intra-tile loop overhead \( \lambda(t^{(1)}) \) contributes only to the execution time of the innermost level. Hence, \( \alpha, \Theta(t^{(1)}), \psi(t^{(1)}) \), and \( \lambda(t^{(1)}) \) are confined to the innermost level and are defined as in the single level tiling case (c.f., Section 2.2).

We formulate the multi-level tiling problem using a generic \( T^{(m)} \), which is a function of all the fundamental measures and architectural parameters at the level \( m \). Based on the combination of processor features and compiler optimizations chosen, we can substitute the corresponding \( T^{(m)} \) to get a concrete optimization problem. Let us consider \( m \) levels of tiling of an \( n \)-depth loop nest, we have the following optimization problem in \( m \times n \) variables:

\[
\begin{align*}
\text{minimize} \quad & T^{(m)} \\
\text{subject to : } \quad & F^{(j)}(t^{(j)}) \leq c^{(j)} \quad \text{for } j = 1 \ldots m \\
& 0 \leq t^{(j)}_{k,i} \leq t^{(j+1)}_{k,i} \quad \text{for } i = 1 \ldots n; \quad j = 1 \ldots m \\
& t^{(j)}_{k,i} \in \mathbb{Z} \quad \text{for } i = 1 \ldots n; \quad j = 1 \ldots m
\end{align*}
\]

Consider the problem of tiling for \( m > 1 \), levels of tiling. The optimization problem is not separable, i.e., it cannot be solved one level at a time, since the tile variables \( t^{(j+1)}_{k,i} \) at a level \((j+1)\) influences \( N^{(j)}, \Delta^{(j)} \), and upper bounds of \( t^{(j)}_{k,i} \). Further, \( T^{(j-1)} \) becomes the computation time of a tile at the next level. Hence, a globally optimal solution would require solving the whole optimization problem.

3.3 Example: 2-level tiling of a doubly nested loop

To illustrate the multi-level tiling formulation, we present the concrete optimization problem for the base cost model \( T_{\text{base}}(\hat{f}) \) used for a loop nest of depth two \((n = 2)\) tiled twice \((m = 2)\). We start from the innermost level of tiling \((j = 1)\) and move to the outer level \((j = 2)\). For the innermost level we have

\[
T^{(1)} = \lambda^{(1)}(\Theta(t^{(1)}) + \Delta(t^{(1)})\beta^{(1)} + \psi(t^{(1)}) + \eta \lambda(t^{(1)}) + \lambda^{(1)} \eta)
\]

where

- \( \Theta(t^{(1)}) = t^{(1)}_1 \times t^{(1)}_2 \),
- \( \Delta(t^{(1)}) = a^{(1)}_1 t^{(1)}_2 + a^{(1)}_2 t^{(1)}_1 \),
- \( \lambda^{(1)} = \frac{t^{(1)}_2}{t^{(1)}_1} + \frac{t^{(1)}_1}{t^{(1)}_2} \),
- \( \psi(t^{(1)}) = t^{(1)}_1 \rho_1 + t^{(1)}_2 \rho_2 \),
- \( \lambda^{(1)} = t^{(1)}_1 + t^{(1)}_2 + \beta^{(1)} \)
- \( \lambda^{(1)} = \frac{t^{(1)}_1 + t^{(1)}_2}{t^{(1)}_1 \times t^{(1)}_2} \),
- \( \beta^{(1)} \) is the bulk transfer rate between memory levels 1 and 2,
- and \( \rho_1, \rho_2 \) are the cost of the load-store statements executed \( t^{(1)}_1 \) and \( t^{(1)}_2 \) times respectively.

The memory footprint at this level is \( F(t^{(1)}) = \lambda^{(1)} \). For the next outer level, \( j = 2 \), we have

\[
T^{(2)} = N^{(2)}(T^{(1)} + \beta^{(1)} \Delta(t^{(2)}) + \eta \Lambda^{(2)})
\]

where, \( \lambda^{(2)} = \frac{N_1}{t^{(2)}_1} + \frac{N_1 \times N_2}{t^{(2)}_1 \times t^{(2)}_2} \), \( \lambda^{(2)} = \frac{N_1 \times N_2}{t^{(2)}_1 \times t^{(2)}_2} \), and \( \beta^{(2)} \) is the bulk transfer rate between memory levels 2 and 3. Note that, \( t^{(2)}_1 = N_1 \) and \( t^{(2)}_2 = N_2 \) since we are tiling a doubly
nested loop of size $N_1 \times N_2$. The memory footprint at this level is $\mathcal{F}(t^{(2)}) = \Delta(t^{(2)})$.

Now, the optimization problem that selects the optimal tile sizes $t_1^{(1)}, t_2^{(1)}, t_2^{(2)}$, and $t_2^{(2)}$ is

$$
\begin{align*}
\text{min.} & \quad T(t^{(2)}) \\
\text{subject to} & \quad \mathcal{F}(t^{(2)}) \leq C^{(2)} \\
& \quad \mathcal{F}(t^{(1)}) \leq C^{(1)} \\
& \quad 0 < t_1^{(2)} \leq N_1 \\
& \quad 0 < t_2^{(2)} \leq N_2 \\
& \quad 0 < t_1^{(1)} \leq t_2^{(2)} \\
& \quad 0 < t_2^{(1)} \leq t_2^{(2)} \\
& \quad t_1^{(1)}, t_2^{(1)}, t_1^{(2)}, t_2^{(2)} \in \mathbb{Z}
\end{align*}
$$

4. GEOMETRIC PROGRAM FRAMEWORK FOR OPTIMAL TILING

We formulate the optimal tile size selection problem as a Geometric Program (GP) [12]. We first introduce the GP framework and then show how the single and multiple-level optimal tiling problems can be cast as an Integer Geometric Program (IGP).

4.1 Geometric Programming

Let $x$ denote the vector $(x_1, x_2, \ldots, x_n)$ of $n$ real, positive variables. A function $f$ is called a posynomial function of $x$ if it has the form

$$
f(x_1, x_2, \ldots, x_n) = \sum_{k=1}^{t} c_k x_1^{\alpha_{1k}} x_2^{\alpha_{2k}} \cdots x_n^{\alpha_{nk}}
$$

where $c_k \geq 0$ and $\alpha_{ij} \in \mathbb{R}$. Note that the coefficients $c_k$ must be nonnegative, but the exponents $\alpha_{ij}$ can be any real numbers, including negative or fractional. When there is exactly one nonzero term in the sum, i.e., $t = 1$ and $c_1 > 0$, we call $f$ a monomial function.\footnote{Note that this definition of monomial is different from the standard one used in algebra.}

where $f_0, \ldots, f_m$ are posynomials, and $g_1, \ldots, g_p$ are monomials. If $\forall i = 1 \ldots n : x_i \in \mathbb{Z}$, we call the GP an Integer Geometric Program (IGP).

4.2 Simple extensions of GP

Using the properties of the monomials and posynomials, several extensions can be easily handled. If $f$ is a posynomial and $g$ is a monomial, then the constraint $f(x) < g(x)$ can be expressed as $f(x)/g(x) \leq 1$, since $f/g$ is a posynomial. For example, we can easily handle constraints of the form $f(x) \leq a$, where $f$ is posynomial and $a > 0$. Similarly, if $g_1(x)$ and $g_2(x)$ are monomials, then a constraint of the form $g_1(x) = g_2(x)$ can be expressed as $g_1(x)/g_2(x) = 1$. We say $h$ is an inverse posynomial if $1/h$ is a posynomial. For an inverse posynomial $h$ and posynomial $f$, we can express constraints of the form $f(x) < h(x)$ as $f(x)/h(x) \leq 1$. We can also maximize an inverse posynomial $h$ by minimizing $1/h$.

4.3 Optimal tiling problem is an IGP

The optimal tiling problem seeks to choose tile sizes that minimize some criteria and satisfy some constraints. The key insight is that the variables of this optimization problem, tile sizes, are always positive. So, polynomial kind of functions of tile sizes naturally become posynomials, when the coefficients are non-negative. We first show that the single-level optimal tiling problem is an IGP, and use the properties of posynomials and GPs to show that the multi-level tiling problem is also an IGP.

**Lemma 4.1** The fundamental measures $\Theta(\vec{t}), \Delta(\vec{t}), \psi(\vec{t}), \eta(\vec{t})$ and $\Lambda(\vec{t})$ are posynomials.

**Theorem 4.2** The single level tiling problem (3.1) is an IGP for all posynomial objective functions.

From Lemma 4.1 and Theorem 4.2, we can observe the cost function $T_{base}(\vec{t})$ introduced in Section 2.4 (Eqn. (2.2)), is a posynomial and using it as $T(\vec{t})$ in the single-level optimal tiling formulation (c.f. Eqn.(3.1)) will yield an IGP.

**Theorem 4.3** The multi-level optimal tiling problem (3.2) is an IGP for all posynomial objective functions.

From Lemma 4.1, we can observe that $T_{base}(\vec{t})$ (c.f. Section 2.4, Equation (2.2)) is a posynomial. Repeated composition of $T_{base}(\vec{t})$ with other posynomials through addition at multiple levels would yield a posynomial since posynomials are closed under addition. Hence, from Theorem 4.3, we can observe that using $T_{base}(\vec{t})$ repeatedly at $m$ levels to construct a $T^{(m)}$ will yield a posynomial $T^{(m)}$ which can
Algorithm 1 Integer solutions of a GP


2. Remove the integrality constraints from $P$ to obtain a continuous relaxation $P'$. Solve $P'$, and let $x = (x_1, \ldots, x_n)$ be its optimal solution.

3. For each $x_i$, compute the neighboring integer values. Let $c_i = \lceil x_i \rceil$ and $f_i = \lfloor x_i \rfloor$, $i = 1 \ldots n$. For each variable, we can choose a ceil or a floor value and use it to construct one combination of integer values. Let $S$ be the set of all such combinations of integer values, constructed from $c_i$’s and $f_i$’s, for $i = 1 \ldots n$.

4. Check the feasibility (with respect to the constraints of $P$) of each element of $S$, and let $F$ be the set of feasible elements. If $F$ is empty exit and report “No integer solutions found.”

5. For each element $e \in F$ compute the value of the objective function of $P$ at $e$. Let $z$ be the element of $F$ that has the least objective function value. Return $z$.

be used in the multi-level optimal tiling formulation (c.f. Eqn. (3.2)) to get an IGP.

4.4 Solving the IGP

The GP optimization problem (4.1) can be transformed to a convex form as shown in Appendix B. The convexified GP can be solved efficiently using interior point methods. Kortanek et. al [20] has proposed an interior-point algorithm for solving GPs. This algorithm is implemented in the optimization library COPL_GP [21]. We use this library to solve a continuous relaxation of the optimal tiling problem, which is an IGP. There are commercial solvers like MOSEK [24] that can also be used for solving the GP. Algorithm 1 shows how to compute the integer solutions from the solution of a continuous relaxation.

It is important to note that theoretically, the integer point closest to the rational/real solution returned by the standard GP solvers may be arbitrarily far from the true optimal of the IGP. However, since the constraints defining the IGP are smooth, we do not expect that the approximation will be too bad. This needs to be validated experimentally.

5. Generality and Extensions

At a first look our model might seem simple, however it is general and can be easily extended. In this section, first we show how our analytical cost model can be extended to include various architectural features and compiler optimizations. Then we show the generality of the GP based framework in accommodating other cost models and functions.

5.1 Extensibility of the cost model

The cost model can be easily refined to include more details about processor features and compiler optimizations. See Table 1 for a list of processor features and compiler optimization influence memory access cost and execution times. Such refinements would either affect the miss rate or miss penalty and can be accommodated by appropriately scaling the bulk transfer rate $\beta$ or by changing the number of misses $\Delta(t)$. For example, consider the following three scenarios:

- **Out-of-order issue & non-blocking cache:** Consider an out-of-order issue processor with a non-blocking cache. The out-of-order issue together with a non-blocking cache can hide the miss penalty for accesses that are a miss at L1 but a hit at L2, given sufficient ILP in the code. This effect can be modeled by reducing the miss penalty for such misses. We can capture this by determining the number of L1 misses for which the miss penalty is reduced and then scaling down $\beta$ by an appropriate factor, say $f_{mr}$.

$$T_{\text{other}, \text{cache}}(t) = \alpha \Theta(t) + (f_{mr, \beta}) \Delta(t) + \psi(t) + \eta \lambda(t) + \eta \lambda(t)$$

To account for set-associativity of the cache, we may have to scale down the cache capacity to an effective cache capacity, as discussed later in this section.

- **Hardware prefetching:** Hardware prefetching is useful for programs that have good spatial locality. For such programs, hardware prefetching can substantially decrease the miss penalty (but not completely remove it, since the hit time to a prefetch buffer is slightly higher than the cache hit time) by prefetching subsequent blocks and storing them in stream buffer. The effect of hardware prefetching can be modelled by scaling down $\beta$ by a factor $f_{np}$. For a reasonably large tile size, $f_{np}$ can be calculated by using the size and latency of the stream buffer.

$$T_{\text{other}, \text{prefetch}}(t) = \alpha \Theta(t) + (f_{np, \beta}) \Delta(t) + \psi(t) + \eta \lambda(t) + \eta \lambda(t)$$

- **Highly optimized execution:** Consider now an advanced processor with all the features listed in Table 1 together with a compiler that can
perform all the optimizations listed in Table 1. The net effect of would be an almost complete overlap of computation and the data movement. In such a case the execution time is the maximum of the time taken for computation and the memory access time.

\[
T_{opt}(\vec{t}) = \max(\alpha \Psi(\vec{t}) + \psi(\vec{t}) + \eta \lambda(\vec{t}), \beta \Delta(\vec{t}) + \psi(\vec{t}) + \eta \lambda(\vec{t}) + \eta \lambda(\vec{t}))
\]

Such a scenario is very common with respect to the hardware features. However, some compiler optimizations like memory reduction, padding and data remapping are not available in all compilers, though the techniques are well understood in the research community.

Other combinations of processor features and compiler optimizations can also be easily included. For example, low-associativity of caches and the use of padding and data remapping [31, 26, 32] can be included by appropriately scaling down the cache capacity \(C\) to an effective cache size. This is a well studied [26, 31, 32, 28] and widely used technique. An algorithm of how to compute the effective cache size can be found in [28]. Such a technique is also used by other researchers in the similar context of single-level and multi-level tiling [22, 35, 6, 23].

Observe that the cost functions \(T_{hi\_prefetch}(\vec{t})\) and \(T_{oi\_nbc}(\vec{t})\) are both posynomials by construction. The function \(T_{hi\_opt}(\vec{t})\) is directly not a posynomial. However, it can be transformed into a posynomial qualified with posynomial inequality constraints using the max elimination technique shown in Lemma A.1 in the Appendix. In a multi-level tiling, if \(T_{hi\_opt}(\vec{t})\) is used repeatedly at each levels, then we will have a function with nested \(\max()\)'s. For this case, we can start from the innermost \(\max()\) and repeatedly apply the max elimination technique shown in Lemma A.1 to obtain a single \(IGP\). Hence, all the three functions can be used in the context of single or multi-level tiling to obtain an \(IGP\) and hence can be solved efficiently. This shows how one can combine the extensibility of our cost model with the generality of the GP based framework to include advanced processor features and compiler optimizations.

### 5.2 Generality of the GP framework

Just as in the design of any optimization model, here too in casting an optimal tiling problem as a GP, the ingenuity of the analyst and a good understanding of what is being modeled are important. However, the GP based optimization framework is very rich and a broad class of cost functions and measure functions can be cast as posynomials, since they are all functions of the positive variables, viz., tile sizes. First, we observe that almost all the cost functions that are used in the context of optimal tile size selection [17], are posynomials. Second, many of the functions used in the constraints (that characterize “valid” or “good” tiles) can also be cast as posynomial inequalities.

As observed by Hsu et. al [17], the single level cost functions used in the context of tile size selection for doubly nested loops [22, 35, 11, 17, 8] are all functions of the tile variables, cache capacity and cache line size. An inspection of their comparison [17, Figure 2] clearly shows that these cost functions are posynomials. There are only few multi-level objective functions proposed. The function (miss count formula) used by Mitchell et. al [23] is a posynomial. The function proposed by Sarkar [28] for use in the IBM-XL FORTRAN compiler, quantifies the memory cost per iteration of a loop nest, and can be cast as a posynomial, after a continuous relaxation of the ceiling (\((\cdot)\)) function. And our cost functions too turned out to be posynomials!6

Measure functions estimate quantities of interest, like the number of cache misses, number of arithmetic operations performed, etc., and are often used in the constraints that characterize “good” tiles. The memory footprint estimation of Ferrante et. al [13] can be cast as a posynomial after a continuous relaxation of the ceiling function. Sarkar [28] presents a refinement of the memory footprint estimation of Ferrante et. al [13], and this too can be cast as a posynomial with the same relaxation. However, there are some measure functions that, to the best of our knowledge, cannot be directly cast as posynomials. For example, the precise estimations of cache misses done through Cache Miss Equations [14] or using Presburger formulae [9], result in Ehrhart polynomials [10], which are psuedo-polynomials, and not even polynomials, leave alone being a posynomial!

### 5.3 Extension to Multiple Levels

Once we have a single level optimal tiling formulation that can be fit into our framework, then the nice closure properties of the posynomials and monomials allows natural extension of the formulation to multiple levels. We have shown this for our formulation in Sections 3 and 4, where we have exploited the fact that posynomials are closed under addition, multiplication and scalar multiplication. One can observe, that such closure properties in fact provide very powerful tools for recursive and compositional formulations.

### 6. EXPERIMENTAL RESULTS

6We first formulated the cost functions, and were pleasantly surprised to observe that they are posynomials.
<table>
<thead>
<tr>
<th>Processor Feature</th>
<th>Impact on Miss Rate / Miss Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-blocking cache and out-of-order issue</td>
<td>Hides L1 miss latency and reduces L1 miss rate</td>
</tr>
<tr>
<td>Critical word first</td>
<td>Reduces miss penalty</td>
</tr>
<tr>
<td>Priority to read misses and merging write-buffers</td>
<td>Reduces miss penalty</td>
</tr>
<tr>
<td>Hardware prefetching</td>
<td>Reduces miss rate or miss penalty</td>
</tr>
<tr>
<td>Larger cache size or line size</td>
<td>Reduces capacity misses</td>
</tr>
<tr>
<td>Higher associativity</td>
<td>Reduces conflict / replacement misses</td>
</tr>
<tr>
<td>Victim caches</td>
<td>Reduces conflict misses</td>
</tr>
<tr>
<td><strong>Compiler Optimization</strong></td>
<td></td>
</tr>
<tr>
<td>Padding for alignment</td>
<td>Reduces conflict misses</td>
</tr>
<tr>
<td>Compiler controlled prefetching</td>
<td>Reduces (or removes) miss penalty</td>
</tr>
<tr>
<td>Projective memory allocation</td>
<td>Reduces memory requirement and [thereby the] number of misses</td>
</tr>
<tr>
<td>Data remapping</td>
<td>Improves locality and reduces number of misses</td>
</tr>
</tbody>
</table>

Table 1: Widely used processor features and compiler optimizations that influence memory access cost and execution time

For our experiments we used the sim-outorder simulator from SimpleScalar tool set [5]. It is a cycle accurate processor simulator with two levels of cache and a TLB. We configured it for an in-order issue processor and we set the caches to be fully associative with sizes 4k(L1) and 64k(L2). As stated in Section 5.1, it is well known that the results obtained for fully associative caches can be adapted to set-associative caches by using standard techniques like padding and using an effective cache size, for example see [27, 26, 22, 13, 28]. We experimented with single and multiple levels of tiling of doubly and triply nested loops. We considered five different programs (dep1, dep1-LF, dep2, stat2, and var2) and four different tiling scenarios: one-level tiled doubly nested loop \((m = 1, n = 2)\), two-level tiled doubly nested loop \((m = 2, n = 2)\), one-level tiled triply nested loop \((m = 1, n = 3)\), and two-level tiled triply nested loop \((m = 2, n = 3)\). The five programs had the following features:

- **dep1**: contains a loop body with floating point addition and a dependence of depth one.

- **dep1-LF**: this is program dep1 with the tile-loop and inter-tile loop of the innermost time dimension fused.

- **dep2**: contains a loop body with floating point addition and a dependence of depth two. Note that a dependence of depth two requires saving and loading two facets of intermediate values along the dependence direction.

- **stat2**: contains a loop body with two independent statements that perform floating point additions. This loop body has instruction level parallelism and also would have exploited the pipelined floating point addition unit.

- **var2**: contains a loop body with two dependent statements (the result of the first used as an operand in the second) that do floating point operations. This loop body does not have ILP but may exploit the pipelined floating point unit.

Each of the above five programs together with the four tiling scenarios resulted in twenty different programs. We ran each of these programs on more than ten different tile and program parameter combinations, resulting in more than two hundred different runs.

We measured the percent error in prediction of the execution time by our model, i.e., the percent error between the simulated execution time and the estimated execution time. The mean and standard deviation of the percent error results are presented in Table 2. A negative mean indicates an underestimation and a positive one an overestimation of the execution time by the model. One can note from the results that our model predicts the execution time with an error (approximately) between 5 to 30 percent. For the purposes of tiling, a high level model with such an error range seems reasonable.

7. RELATED WORK

Tiling for memory hierarchy is a well studied problem and so is the problem of modeling the cache behavior of a loop nest. We classify the related work

---

7The tiled codes are available at: http://www.cs.colostate.edu/~ln/tiled-loops
<table>
<thead>
<tr>
<th>Program</th>
<th>$m = 1, n = 2$</th>
<th>$m = 2, n = 2$</th>
<th>$m = 1, n = 3$</th>
<th>$m = 2, n = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$</td>
<td>$\sigma$</td>
<td>$\mu$</td>
<td>$\sigma$</td>
</tr>
<tr>
<td>dep1</td>
<td>-1.71</td>
<td>3.35</td>
<td>15.79</td>
<td>8.71</td>
</tr>
<tr>
<td>dep1-LF</td>
<td>-1.76</td>
<td>3.33</td>
<td>15.78</td>
<td>8.43</td>
</tr>
<tr>
<td>dep2</td>
<td>-0.36</td>
<td>3.96</td>
<td>21.72</td>
<td>11.58</td>
</tr>
<tr>
<td>var2</td>
<td>17.99</td>
<td>2.36</td>
<td>22.48</td>
<td>4.09</td>
</tr>
</tbody>
</table>

Table 2: Experimental Results. Mean ($\mu$) and standard deviation ($\sigma$) of the percent error between predicted and simulated execution times. The number of levels of tiling is $m$, and $n$ is the loop nest depth.

into three categories: models of cache behavior of loop nests, single-level optimal tiling and multi-level optimal tiling. As stated earlier, we do not address tile shape determination, and the many excellent references on this are not directly relevant.

*Models of cache behavior of loop nests.* There are several analytical models that measure the number of cache misses for a given class of loop nests. These models can be classified into precise models that use sophisticated (computationally costly) methods and approximate models that provide a closed form with simple analysis. In the precise category, we have the Cache Miss Equations [14], and the refinement by Chatterjee et. al [9], that uses Ehrhart Polynomials [10] and Presburger formulae to describe the number of cache misses. Harper et. al [15] propose an analytical model of set-associative caches and Cascaval and Padua [7] give a compile time technique to estimate cache misses using stack distances. In the approximate category, Ferrante et. al [13] present techniques to estimate the number of distinct cache lines touched by a given loop nest. Sarkar [28] presents a refinement of this model. Although the precise models can be used for selecting the optimal tile sizes, only Abella et. al [1] has proposed a near optimal loop tiling using Cache Miss Equations and genetic algorithms. Sarkar and Megiddo [29] have proposed an algorithm that uses an approximate model [13] and finds the optimal tile sizes for loops of depth up to three. No previous work has used any of these models to find optimal tile sizes in the context of multi-level tiling of loop-nests of arbitrary depth. Our execution model, though an approximate one, can be used for multiple-levels of tiling as shown in this paper.

[16, 37]

*Single-level optimal tiling.* Several algorithms [22, 11, 8, 17] have been proposed for single-level tile size selection (see Hsu and Kremer [17] for good a comparison). The majority of them use a local cost function such as the number of capacity misses or conflict misses, not a global metric like ours, viz., overall execution time. Mitchell et. al [23] illustrate how such local cost functions may not lead to globally optimal performance.

Multi-level optimal tiling. Mitchell et. al [23] was the first one to quantify the multi-level interactions of tiling. They clearly point out the importance of using a global metric like execution time rather than local metrics like number of misses, etc. Further, they also show through examples, the interactions between different levels of tiling and hence the need for a framework in which the tile sizes at all the levels are chosen simultaneously with respect to a global cost function. In this paper we have proposed one such framework for a restricted class of programs.

Högstedt et. al [16] address the problem of multi-level tiling, and develop a model of the running time of such tiles programs. Their model is complicated because they address semi-oblique tiling, which is more general than orthogonal tiling. Wonnacott [37] also addresses multiple levels of tiling, one for parallelism and one for locality. This work too is in the context of semi-oblique tiling, and the author does not address the problem of tile size optimization. It would be interesting to extend our model to (semi) oblique tiling, but this is currently open.

Other results that show the application and importance of multi-level tiling include [6, 25, 19]. Empirical tools like PHIPAC [3] and ATLAS [34] use a profile-driven approach to choose the optimal tile sizes. These tools are limited to the set of programs for which they are designed and are time consuming.

8. CONCLUSIONS

We have presented a GP based unifying framework for the optimal multi-level tiling problem. We have also presented a high level analytical model that predicts the execution time of a multi-level tiled loop nest. We have shown the generality of our GP based framework in accommodating other cost models and functions and also the extensibility of our cost model in including advanced processor features and com-
piler optimizations. We have presented experimental results that validate our cost model. To the best of our knowledge, this is the first unified framework proposed for the optimal multi-level tiling problem.

As part of our ongoing work, we plan to validate our cost model with more programs and different cache and processor configurations. As a next step, we will consider two multi-level tiling scenarios: (a) an outer level of tiling for parallelism and inner level of tilings for memory hierarchy; (b) outer levels of tilings for memory hierarchy and inner level of tiling for instruction level parallelism (III). Extending the program model to include non-rectangular loop nests and non-uniform (say, affine) dependences would be the next major step, but remains a difficult problem.

9. REFERENCES


**APPENDIX**

**A. PROOFS**

**Lemma 4.1** The fundamental measures $\Theta(\bar{t}), \Delta(\bar{t}), \psi(\bar{t}), \eta(\bar{t})$ and $\Lambda(\bar{t})$ are posynomials.

**Proof.** From the definition of these measures (c.f. Section 2.2) one can directly observe that they are all posynomials, since all the coefficients are non-negative, the variables (tile sizes) are always positive, and posynomials are closed under addition.

**Theorem 4.2** The single level tiling problem (3.1) is an iop for all posynomial objective functions.

**Proof.** We need to show that all the constraints
in (3.1) can be cast as posynomial inequality constraints or monomial equality constraints as in (4.1). The positivity and integrality constraints on \( t_i \) naturally maps into the constraints of \( \text{GP} \). The capacity constraint, \( F(\hat{t}) \leq C \) can also be easily cast as a posynomial inequality constraint by the following rewrite \( F(\hat{t}) \leq C \iff C^{-1}(\Delta(\hat{t})) \leq 1 \), which is a posynomial, since \( \Delta(\hat{t}) \) is a posynomial (from Lemma 4.1) and \( C \) is a constant. Hence, whenever the objective function is (also) a posynomial, the whole problem is an \( \text{IGP} \).

**Theorem 4.3** The multi-level optimal tiling problem (3.2) is an \( \text{IGP} \) for all posynomial objective functions.

**Proof.** The proof follows directly from the proof for the single-level case (Theorem 4.2) since, we have just added some more constraints that are all similar in form to the ones in (3.1). Hence, whenever the objective function is a posynomial we have an \( \text{IGP} \), and we can solve for the tile sizes directly.

**Lemma A.1 (Max Elimination)** The optimization problem

\[
\begin{align*}
\text{minimize} & \quad T = \max(p(x), q(x)) \\
\text{subject to} & \quad v \leq 1 \\
& \quad q(x) \leq v
\end{align*}
\]

where \( p(x) \) and \( q(x) \) are posynomials, is equivalent to the optimization problem

\[
\begin{align*}
\text{minimize} & \quad T = v \\
\text{subject to} & \quad \frac{p(x)}{v} \leq 1 \\
& \quad \frac{q(x)}{v} \leq 1
\end{align*}
\]

where \( v \) is a positive variable.

**Proof.** We can decompose \( T \) into two functions that are defined on the two domains that correspond to the cases when \( p(x) > q(x) \) and \( p(x) \leq q(x) \). For the former case, we have \( T_1 = p(x) \) with the constraint qualification \( p(x) > q(x) \), and the for the later, we have \( T_2 = q(x) \) with the constraint qualification \( p(x) \leq q(x) \). Note that we can relax the constraint qualification \( p(x) > q(x) \) on \( T_1 \) to \( p(x) \geq q(x) \), since in the case of equality either one of the decompositions is valid. We will show how these two cases reduces to the (required) form of equation (A.1).

**Case** \( p(x) \geq q(x) \): Let us introduce a positive variable \( v = p(x) \). Substituting the new variable \( v \) in \( T_1 \) we get

\[
\begin{align*}
\text{minimize} & \quad T_1 = v \\
\text{subject to} & \quad v = p(x) \\
& \quad v \geq q(x)
\end{align*}
\]

We can rewrite \( v = p(x) \) as \( p(x) \leq v \leq p(x) \). Since \( v \) is a positive variable that occurs on an objective function \( (T_1) \) that is minimized, the upper bound on \( v \) is redundant, and hence \( p(x) \leq v \) is sufficient. This can be written as \( \frac{p(x)}{v} \leq 1 \), since \( v \) is a monomial. Also note that we can rewrite \( v \geq q(x) \) as \( \frac{q(x)}{v} \leq 1 \), for the same reason that \( v \) is a monomial. Hence, we now have

\[
\begin{align*}
\text{minimize} & \quad T_1 = v \\
\text{subject to} & \quad \frac{p(x)}{v} \leq 1 \\
& \quad \frac{q(x)}{v} \leq 1
\end{align*}
\]

Note that both the problems (A.2) and (A.3) are the same except for the variable names \( u \) and \( v \). After all, what is in a name? We could use the same variable name, say \( v \), and would arrive at the following single problem

\[
\begin{align*}
\text{minimize} & \quad T = v \\
\text{subject to} & \quad \frac{p(x)}{v} \leq 1 \\
& \quad \frac{q(x)}{v} \leq 1
\end{align*}
\]

where the subscript on \( T \) is also dropped, since both \( T_1 \) and \( T_2 \) are same. One can see that the above form is equivalent to the minimization of the original \( T \) as in the formulation (A.1). Some intuition can be got by observing that the two constraints are essentially bounding \( p(x) \) and \( q(x) \) from above by \( v \) and the occurrence of \( v \) in the objective function ensures that \( v \) is reduced to a value as small as possible. Which implies that both \( p(x) \) and \( q(x) \) will also remain as small as possible with the maximum of them reaching their lowest possible value at the optimum and the minimum of them still having some slack.

**B. GP INCONVEX FORM**

A GP can be reformulated as a convex optimization problem, in particular, as the problem of minimizing a convex function subject to convex inequality constraints and linear equality constraints [4]. Such a reformulation permits an efficient and global solution to the \( \text{GP} \). The reformulation involves a transformation of the variables as follows. We define new variables

\[
\begin{align*}
T_1 &= v \\
\text{subject to} & \quad v = p(x) \\
& \quad v \geq q(x)
\end{align*}
\]
\[ y_i = \log x_i, \text{ and take the logarithm of a posynomial} \]
\[ f \text{ to get} \]
\[ h(y) = \log(f(e^{y_1}, \ldots, e^{y_n})) = \log \left( \sum_{k=1}^{t} e^{\alpha_k y + b_k} \right) \]
\[ \text{where } \alpha_k^T = [\alpha_{1k}, \ldots, \alpha_{nk}] \text{ and } b_k = \log c_k. \]

It is known [4, Secs. 4.5 and 3.1.5] that such a function is a convex function of the new variable \( y \). Note that if the posynomial \( f \) is a monomial, then the transformed function \( h \) is an affine function.

We can convert the GP (4.1) into a convex program by expressing it as

\[
\begin{align*}
\text{minimize} & \quad \log f_0(e^{y_1}, \ldots, e^{y_n}) \\
\text{subject to} & \quad \log f_i(e^{y_1}, \ldots, e^{y_n}) \leq 0, \quad i = 1, \ldots, (B+1) \\
& \quad \log g_i(e^{y_1}, \ldots, e^{y_n}) = 0, \quad i = 1, \ldots, p
\end{align*}
\]

This is the so-called geometric program in convex form (GP-CF). Convexity of the GP-CF (B.1) has several important implications: we can use efficient interior-point methods to solve them [20], and there is a complete and useful duality theory for them.