11-22-2022 10:30 AM

Three Contributions to the Theory and Practice of Optimizing Compilers

Linxiao Wang, *The University of Western Ontario*

Supervisor: Moreno Maza, Marc, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Computer Science

Abstract

The theory and practice of optimizing compilers gather techniques that, from input computer programs, aim at generating code making best use of modern computer hardware. On the theory side, this thesis contributes new results and algorithms in polyhedral geometry. On the practical side, this thesis contributes techniques for the tuning of parameters of programs targeting GPUs. We detailed these two fronts of our work below.

Consider a convex polyhedral set P given by a system of linear inequalities $\mathbf{A}\vec{x} \leq \vec{b}$, where $\mathbf{A}$ is an integer matrix and $\vec{b}$ is an integer vector. We are interested in the integer hull P_I of P which is the smallest convex polyhedral set that contains all the integer points in P. In Chapter 3 we discuss our findings on the pseudo-periodicity of the vertices of P_I when the input vector $\vec{b}$ is parametric, that is, the coordinates $b_1, \ldots, b_m$ of $\vec{b}$ are treated as parameters while the coefficients of $\mathbf{A}$ have fixed values. We observe that the number of vertices of P_I has a pseudo-period T_i w.r.t each b_i. This result and its proof lead us to propose a new algorithm for computing the integer hull of a rational convex polyhedral set, see Chapter 4. We have implemented in the C programming language our algorithm for the case of polyhedral sets in dimensions 2 and 3. We have also realized a Maple implementation of our algorithm for polyhedral sets of arbitrary dimension. Our experimental results show that our algorithm computes integer hulls efficiently and can deal with polyhedral sets with large numbers of integer points.

On another front, we present KLARAPTOR (Kernel LAunch parameters RAtional Program estimaTOR), a freely available tool built on top of the LLVM Pass Framework and NVIDIA CUPTI API to dynamically determine the optimal values of launch parameters of a CUDA kernel. We describe a technique that, for a CUDA kernel, builds at compile-time, a so-called *rational program*. This rational program, based on some performance prediction model, and knowing particular data and hardware parameters at runtime, can be executed to automatically and dynamically determine the values of launch parameters, for the CUDA kernel, that will yield nearly optimal performance.

Keywords: Integer hull, polyhedral set, parametric polyhedron, pseudo-periodic functions, performance estimation, performance portability, CUDA, manycore accelerators, LLVM Pass Framework

Summary for Lay Audience

The theory and practice of optimizing compilers gather techniques that, when given input in the form of computer programs, aim at generating code that makes the best use of modern computer hardware to run those computer programs. On the theory side, this thesis contributes new results and algorithms in polyhedral geometry. On the practical side, this thesis contributes techniques for the tuning of parameters of programs targeting Graphic Processing Units (GPUs). We give a high-level introduction of these two fronts of our work below.

The many constraints of real-world problems, such as finding the number of construction workers required to complete a given task within a budget and before a deadline, can be translated into systems of multiple linear expressions. A solution to the system of expressions represents a solution to the real-world problem that satisfies all the constraints. Solving a linear system on real numbers can be simple but impractical, such as assigning $9\frac{3}{4}$ construction workers to the project, while practical solutions are often integer and require a long time to compute. Linear systems can often be represented as convex polyhedral sets, such as a triangle in 2 dimensional space or a cube in 3 dimensional space. If the integer points within a polyhedral set can be computed, these points can be used to solve the integer linear system. Therefore, we present the theory and an algorithm to find the integer points in a polyhedral set, and we show that our method is significantly faster than existing ones.

Software can often be launched with different parameters. Sometimes, parameters are requirements of the users (data parameters) and affect the correctness of the results. In other cases, the parameters are fixed with regard to the hardware (hardware parameters). We focus on parameters that have no impact on the results, but instead are related to the efficiency of the software (program parameters). Optimizing the program parameters can save resources such as time and memory but can be difficult without being an expert on the specific program. We present KLARAPTOR (Kernel LAunch parameters RAtional Program estimaTOR), a tool that dynamically computes program parameters that result in a good performance of the program. KLARAPTOR works on CUDA; the fine control of the hardware resources afforded by GPUs are preferable to CPUs and CUDA is the most popular programming language for GPUs. The underlying technique of KLARAPTOR can be applied to parallel programs in general, given a performance prediction model which accounts for data, program and hardware parameters.

Co-Authorship Statement

- Chapter 3 is a joint work with Marc Moreno Maza. It is published in [49]. The contributions of the thesis author include: proposed the assumption based on the primarily experimentation results, proved the lemmas and theorems, generate plots and graphs to illustrate our theory.
- Chapter 4 is a joint work with Marc Moreno Maza. It is published in [46]. The contributions of the thesis author include: implementation and testing of the algorithm in both MAPLE and C, generalization of the algorithm to arbitrary dimensional inputs, collection of experimental results.
- Chapter 5 is a joint work with Alexander Brandt, Marc Moreno Maza, Davood Mohajerani and Jeeva Paudel. A preprint of this work is published in [14]. The contributions of the thesis author include: reviewing and selecting performance models, generation of instrumented intermediate code (PTX) using the LLVM Pass Framework, implementation of model metrics computation.

Acknowlegements

First and foremost, I want to thank my supervisor Professor Marc Moreno Maza for his guidance and mentoring over the past six years in both research and life. Whenever there seems to be no path forward, his suggestions and encouragements are always the most reassuring words that lead me through. And of course, thank you Marc and Yuzhen for all the good food and wine.

I want to thank all my co-authors and co-workers Alex Brandt, Davood Mohajerani, Delaram Talaashrafi, Taaabish Jeshani, Erik Postma and Jürgen Gerhard for their enlightening discussions and suggestions.

I also want to thank my examiners Dr. Taylor Brysiewicz, Dr. Dan Steffy, Dr. Mike Domaratzki and Dr. Mostafa Milani for their inspiring questions and careful reviews.

During this unexpectedly challenging time, I cannot imagine what it would be like without my family and friends. I especially want to thank my parents; they are my most powerful backing. I want to thank all my friends (especially my dear bug group) for listening to my babbling when I am anxious, excited or overly-caffeinated and for all the random midnight discussions about the weirdest stuff in the world. I want to thank Dave and Rose for making me feel like I have a home here in a foreign country. And to my partner, Andrew, I will go insane without you.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

The theory and practice of optimizing compilers gather techniques that, from input computer programs, aim at generating code making best use of modern computer hardware. Several core areas of computer science and mathematics contribute such techniques: automata theory, computer architecture, distributed and parallel programming to name a few.

One important set of techniques for optimizing compilers are used for the problem of analyzing, scheduling and transforming for-loop nests in computer programs. Because the iterations of a for-loop nest can be seen as the integer points of a rational polyhedral set, the area of *polyhedral geometry* plays a central role. In many instances of that problem, the polyhedral set is clearly given while its integer points need to be described in a synthetic way from which useful information can be extracted. Describing the integer points of a rational polyhedral set is a hard problem which is still actively researched today; we introduce it in Section 1.1.

Another important set of techniques for optimizing compilers are used to solve the following problem: given an executable program, the performance of which is dependent on tunable parameters, how to effectively determine optimal values for those parameters. By effectively, we imply that the cost of this determination should be largely amortized against the obtained benefits. A well-known special case of that problem is the tuning of parameters related to the characteristics of the memory hierarchy (e.g. thresholds between base-cases and recursive calls) towards minimizing cache misses. Another special case is the determination of the kernel launch parameters of a program written for Graphics Processing Units (GPUs). This is another hard problem which is also actively researched today; we introduce it in Section 1.2.

1.1 Integer hulls

The integer points of rational polyhedral sets are of great interest in various areas of scientific computing. Two such areas are *combinatorial optimization* (in particular integer linear programming) and *compiler optimization* (in particular, the analysis, transformation and scheduling of for-loop nests in computer programs), where a variety of algorithms have been designed to solve questions related to the points with integer coordinates belonging to a given polyhedron. Another area is at the crossroads of computer algebra and polyhedral geometry, with topics like toric ideals and Hilbert bases, see for instance [62] by Thomas.

One can ask different questions about the integer points of a polyhedral set, ranging from

"whether or not a given rational polyhedron has integer points" to "describing all such points". Answers to the latter question can take various forms, depending on the targeted application. For plotting purposes, one may want to enumerate all the integer points of a 2D or 3D polytope. Meanwhile, in the context of combinatorial optimization or compiler optimization, more concise descriptions are sufficient and effective.

For a rational convex polyhedron $P \subseteq \mathbb{Q}^d$, defined either by the set of its facets or that of its vertices, one such description is the *integer hull* P_I of P, that is, the convex hull of $P \cap \mathbb{Z}^d$. The set P_I is itself polyhedral and can be described either by its facets, or its vertices. In addition to finding the description of the whole integer hull P_I another problem that is well studied is that of counting the integer points in a rational polyhedron. The region enclosed by the blue segments in Figure 1.1 shows an example of a convex polyhedral set P. All the green points are the integer points within P. The red segments are the facets of the integer hull P_I of P.

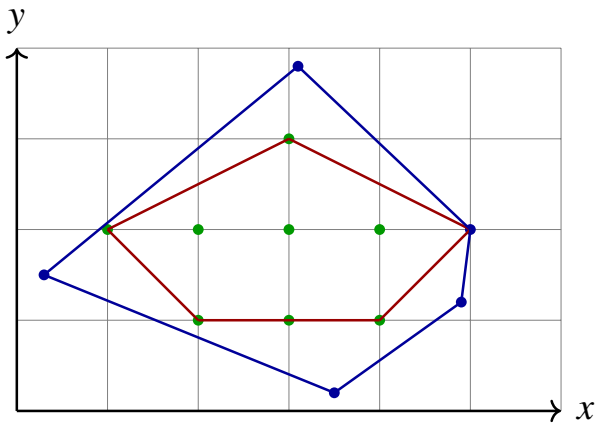


Figure 1.1: A convex polyhedral set and its integer hull

In practice, polyhedral sets are often *parametric*. Consider, for instance, the for-loop nest, written in a programming language (say C) of a dense matrix multiplication algorithm (see Listing 1). At compile time, the upper bound of the value range of each loop counter is a symbol. To be more precise, the iterations of that for-loop nest are the integer points of a polyhedral set P given by a system of linear inequalities $A\vec{x} \leq \vec{b}$ where A is a matrix with integer coefficients, $\vec{b}$ is a vector of symbols (actually the parameters of the polyhedral set) and $\vec{x}$ is the vector of the loop counters. For Listing 1, the iteration space is a cube shaped polyhedral set with the following constrains

$$\begin{cases} i, j, k \in \mathbb{Z} \\ 0 \leq i \leq n \\ 0 \leq j \leq m \\ 0 \leq k \leq p \end{cases} \tag{1.1}$$

At execution time, different values of $\vec{b}$ yield different shapes and numbers of vertices for P_I. Figure 1.2 shows the integer hulls of a parametric polyhedral set given by

$$\begin{cases} -x & \leq & 0 \\ -2x + y & \leq & 0 \\ 2x + y & \leq & b \end{cases}$$

when $b = 10$ and $b = 13$. We can see that the shapes and the numbers of the vertices of the integer hulls vary while b is taking different values. So what can be done at compile time when analyzing a parametric polyhedral set? This is another question motivating our work.

```
for (int i = 0; i < n; i ++)
    for (int j = 0; j < m; j ++)
        for (int k = 0; k < p; k ++)
            C[i][j] += A[i][k] * B[k][j];
```

Listing 1: Loop-nest of a dense matrix multiplication

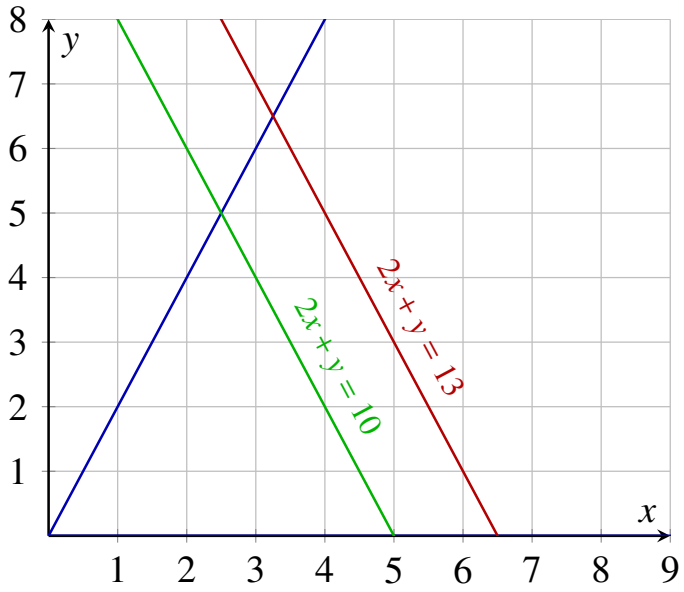


Figure 1.2: The integer hulls of a parametric polyhedral set with different values of $\vec{b}$

For a parametric polyhedral set, it is challenging to give a description of all the integer points since the number and positions of the integer points depend on the values of the parameters. It is natural to ask whether we can describe the vertices of the integer hull and a first step would be asking for the number of vertices in an integer hull of a polyhedral set. Note that this latter problem only considers the vertices not all the lattice points.

With all these observations in mind, we propose to study the following two problems.

1. For a non-parametric polyhedral set in arbitrary dimension, can we design an effective algorithm to compute its integer hull? By effective, we mean an algorithm which can support problems arising in practice, in particular in the analysis of for-loop nests. As the literature review will show, existing algorithms make little use of geometric considerations. In loose terms, the key geometric observation that we have in mind is the following: if P is "almost identical" to P_I then the cost of computing the vertices of P_I

from the vertices of P should be "cheap" and "nearly independent" of the size (or volume) of P. In other words, computing P_I should be achieved by "deforming" P in an economical way until P becomes P_I.

2. For a parametric polyhedral set $P(\vec{b})$, which information on the map $\vec{b} \longmapsto P(\vec{b})_I$ can be computed? Our key observation is that, when the values of the coordinates of $\vec{b}$ are large enough, this map is *pseudo-periodic*. As the literature review will show, existing results have studied this phenomenon. Our preliminary results for the two-dimensional case enhance what was known. Meanwhile, generalizing our results to higher dimension is left for future work. Knowing that the map $\vec{b} \longmapsto P(\vec{b})_I$ is pseudo-periodic can help computing information about $P(\vec{b})_I$ before the value of $\vec{b}$ is known, that is, at *compile-time*, to speak in the language of compiler theory.

1.2 Kernel launch parameters of CUDA kernels

CUDA is the most popular computation model and programming language for GPUs, see [50] where CUDA was originally proposed. A CUDA program interleaves C code, meant to be executed on the CPU, and multithreaded code meant to be executed on the GPU. This multithreaded code takes the form of calls to functions, called kernels.

A CUDA kernel is typically launched by specifying *launch parameters* that dictate the number of device threads executing the kernel in parallel, and the grouping of thread processors on the GPU. For instance, a kernel for adding two vectors would be specified as:

```
sum<<<G,B,S>>>(d_a, d_b, d_c);
```

where:

1. `G` specifies the dimension and size of the grid,
2. `B` specifies the dimension and size of each block,
3. `S` specifies the number of bytes in shared memory that is dynamically allocated per block,
4. `d_a`, `d_b`, and `d_c` are arrays allocated in the GPU memory.

Because they define the execution configuration of the kernels, B, T, and G are called *Program Parameters*. Because they represent the data upon which a kernel operates, `d_a`, `d_b`, and `d_c` are called *Data Parameters*.

Data parameters are independent from program parameters, and are determined by users' needs and available hardware resources. Program parameters, however, are intimately related to data and hardware resources. Meanwhile, the choice of program parameters can largely affect the performance of the program. Therefore, determining optimal values of program parameters that yield the best program performance for a given confluence of hardware and data parameter values is critical. Further, determining such values automatically is important to enable users to execute the same parallel program efficiently on different hardware platforms. This is the third problem studied in this thesis.

To understand the challenge, we observe that trying at execution-time all the possible values of the program parameters would take an amount of time that would not be acceptable in practice. Determining the best values of those program parameters at compile-time is also challenging since at compile-time the values of the data parameters may not be known!

1.3 Contributions

The contributions of this thesis address the two problems proposed in Section 1.1 as well as the problem proposed in Section 1.2. Our solutions are presented respectively in Chapters 3, 4 and 5. Chapter 2 gathers background material and literature review for Chapters 3 and 4. Meanwhile, background material and literature review for Chapter 5 can be found in that same chapter.

On the pseudo-periodicity of the integer hull of parametric convex polygons

Consider a rational convex polygon given by a system of linear inequalities $A\vec{x} \leq \vec{b}$, where A is a matrix over $\mathbb{Z}$, with m rows and 2 columns, and $\vec{b}$ is an integer vector. The coordinates $b_1, \ldots, b_m$ of $\vec{b}$ are treated as parameters while the coefficients of A have fixed values. We observe that for every $1 \leq i \leq m$, there exists a positive integer T_i so that, when each $b_1, \ldots, b_m$ is large enough, the vertex sets V and V' of the respective integer hulls of $P := P(b_1, \ldots, b_{i-1}, b_i, b_{i+1}, \ldots, b_m)$ and $P' := P(b_1, \ldots, b_{i-1}, b_i + T_i, b_{i+1}, \ldots, b_m)$, are in a "simple" one-to-one correspondence. We state and prove explicit formulas for the pseudo-period T_i and that correspondence between V and V'. This result and the ingredients of its proof lead us to propose a new algorithm for computing the integer hull of a rational convex polygon.

Computing the Integer Hull of Convex Polyhedral Sets

We discuss a new algorithm for computing the integer hull P_I of a rational polyhedral set P, together with its implementation in Maple and in the C programming language. Our presentation focuses on the two-dimensional and three-dimensional cases. However, we state our algorithm in arbitrary dimension. Moreover, the Maple implementation works in arbitrary dimension too. Our experimental results show that our algorithm computes integer hulls efficiently and can deal with polyhedral sets with large numbers of integer points.

KLARAPTOR: A Tool for Dynamically Finding Optimal Kernel Launch Parameters Targeting CUDA Programs

We present KLARAPTOR (Kernel LAunch parameters RAtional Program estimaTOR), a freely available tool built on top of the LLVM Pass Framework and NVIDIA CUPTI API to dynamically determine the optimal values of kernel launch parameters of a CUDA kernel. We describe a technique that, for a CUDA kernel, builds at compile-time, a so-called *rational program*. This rational program, based on some performance prediction model, and knowing particular data and hardware parameters at runtime, can be executed to automatically and dynamically determine the values of launch parameters, for the CUDA kernel, that will yield nearly optimal performance. Our underlying technique could be applied to parallel programs in general, given a performance prediction model which accounts for program and hardware parameters. We have implemented and successfully tested our technique in the context of GPU kernels written in CUDA.

Chapter 2

Background on integer hull

This chapter gathers background material and literature review for the integer hull problem which we study in Chapter 3 and Chapter 4.

2.1 Preliminaries

In this review of polyhedral geometry, we follow the concepts and notations of Schrijver's book [56], As usual, we denote by $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ the ring of integers, the field of rational numbers and the field of real numbers. Unless specified otherwise, all matrices and vectors have their coefficients in $\mathbb{Z}$. A subset $P \subseteq \mathbb{Q}^d$ is called a *convex polyhedron* (or simply a *polyhedron*) if $P = \{\mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} \leq \vec{b}\}$ holds, for a matrix $A \in \mathbb{Q}^{m\times d}$ and a vector $\vec{b} \in \mathbb{Q}^m$, where m and d are positive integers; we call the linear system $\{A\mathbf{x} \leq \vec{b}\}$ an *H-representation* of P. Hence, a polyhedron is the intersection of finitely many affine half-spaces. Here an affine half-space is a set of the form $\{\mathbf{x} \in \mathbb{Q}^d \mid \vec{w}^t\mathbf{x} \leq \delta\}$ for some nonzero vector $\vec{w} \in \mathbb{Z}^d$ and an integer number δ. When $d = 2$, as in the rest of this thesis, the term *convex polygon* is used for convex polyhedron.

A non-empty subset $F \subseteq P$ is a *face* of P if $F = \{\mathbf{x} \in P | A'\mathbf{x} = \vec{b'}\}$ for some subsystem $A'\mathbf{x} \leq \vec{b'}$ of $A\mathbf{x} \leq b$. A face distinct from P and with maximum dimension is a *facet* of P. The *lineality space* of P is $\{\mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} = \vec{0}\}$ and P is said *pointed* if its lineality space has dimension zero. Note that, in this thesis, we only consider pointed polyhedra. For a pointed polyhedron P, the inclusion-minimal faces are the *vertices* of P.

We are interested in computing P_I the *integer hull* of P, that is, the smallest convex polyhedron containing the integer points of P. In other words, P_I is the intersection of all convex polyhedra containing $P \cap \mathbb{Z}^d$. If P is pointed, then $P = P_I$ if and only if every vertex of P is integral [56]. Therefore, the convex hull of all the vertices of P_I is P_I itself.

In this thesis, we also talk about parametric polyhedra. In particular, we use the notation $P(\vec{b}) = \{\mathbf{x} \mid A\mathbf{x} \leq \vec{b}\}$ where $\vec{b}$ is unknown and $P(b_i) = \{\mathbf{x} \mid A\mathbf{x} \leq \vec{b}\}$ where b_i is an unknown coordinate of the vector $\vec{b}$.

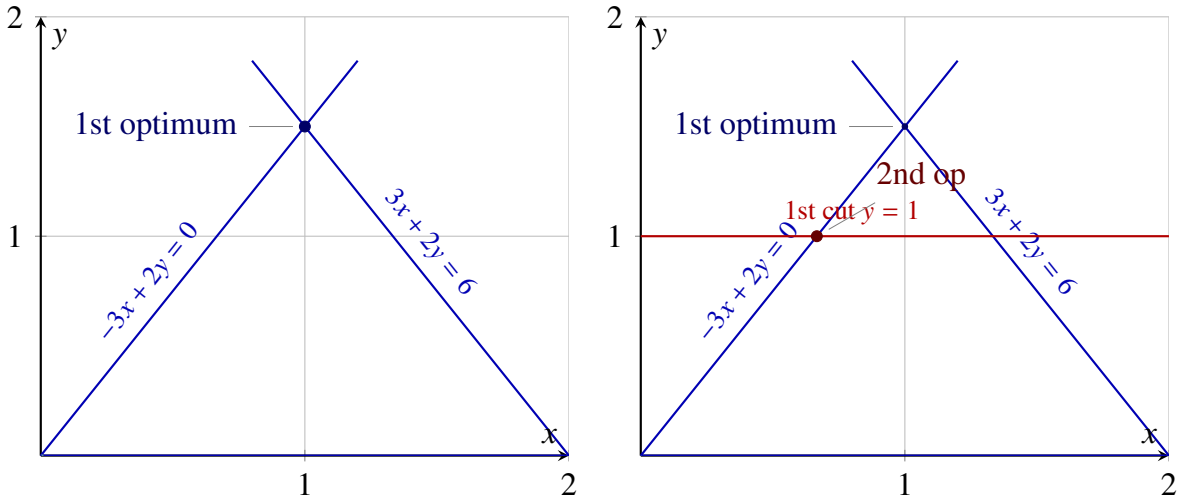
2.2 Literature review

2.2.1 Integer hull

One important family of algorithms for computing P_I relies on the *cutting plane method*, originally introduced by Gomory in [28] to solve integer linear programming (ILP) and mixed-integer programming (MILP) problems. This method is based on finding a sequence of linear inequalities (cuts) to reduce the feasible region to the original ILP problem. Figure 2.1 and 2.2 show an example of using the cutting-plane method to solve the ILP problem given by (2.1).

$$\begin{array}{ll} \text{maximize} & y \\ \text{subject to} & x, y \in \mathbb{Z} \\ & x \geq 0 \\ & -3x + 2y \leq 0 \\ & 3x + 2y \leq 6 \end{array} \tag{2.1}$$

The first step is to solve the problem as a LP problem and find the general optimum solution which is not necessarily an integer solution. Solving the above problem gives a fractional optimum $(1, \frac{3}{2})$. If the current optimum is integer, then it is the final result. If it is a fractional solution, the next step is to add a new constraint to reduce the feasible region. The current fractional optimum must not satisfy the new constraint while all the integer feasible solutions of the original LP should remain feasible. The first cut added is $y \leq 1$ and we have our second optimum $(\frac{2}{3}, 1)$ which is still a fractional solution. So the second cut, $y \leq x$, is added and we obtain a integer optimum $(1, 1)$ which is the final solution to our ILP.

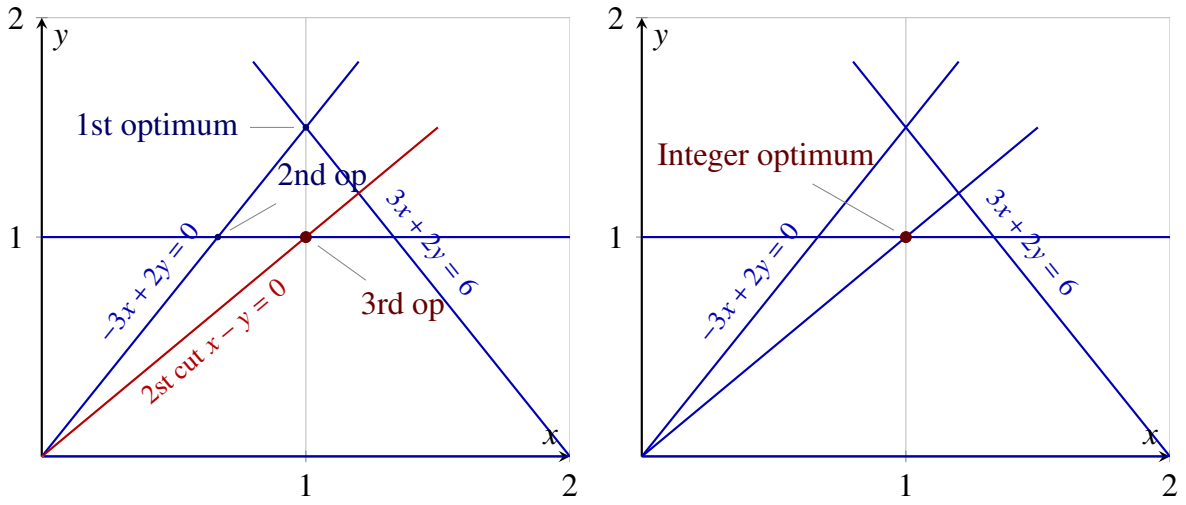


(a) An integer linear programming problem

(b) Adding the first cut

Figure 2.1: Solving ILP with cutting-plane method

Chvátal [17] and Schrijver [55] gave a geometrical description of the cutting plane method and developed a procedure to compute P_I based on it. Schrijver gave a full proof and a complexity study of this method in [56].



(a) Adding the second cut

(b) Optimum integer solution

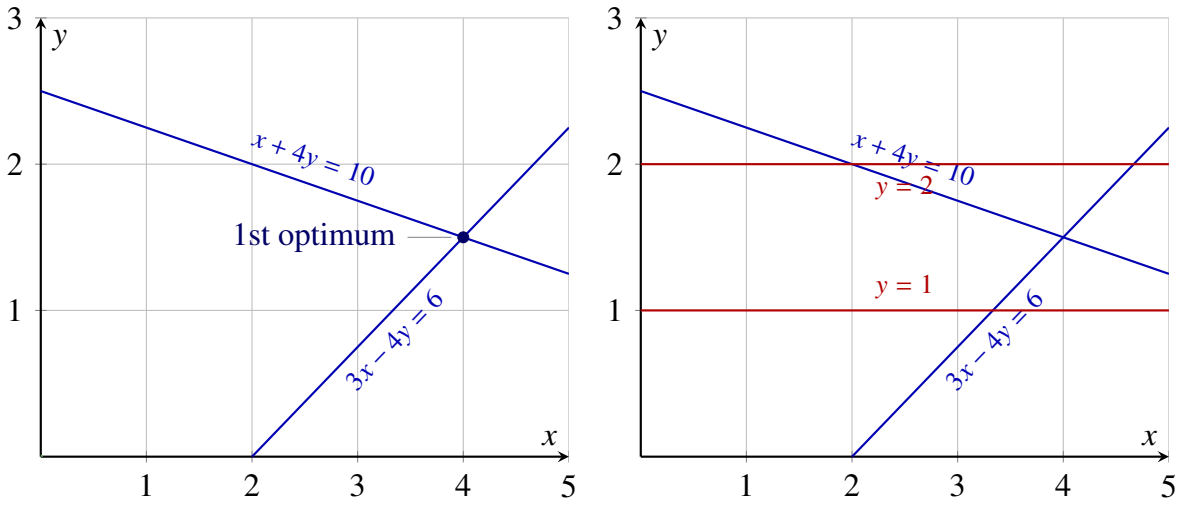
Figure 2.2: Solving ILP with cutting-plane method II

Another approach for optimizing over P_I uses the *branch-and-bound method*, introduced by Land and Doig in the early 1960s in [41]. This method recursively divides P into sub-polyhedra, then the vertices of the integer hull of each part of the partition are computed. The branch-and-bound algorithm is also first introduced to solve the integer optimization problem. It consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represents subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and the branch is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

For example, see Figure 2.3 shows the first steps of using the branch and bound algorithm to solve the following ILP.

$$\begin{array}{ll} \text{maximize} & 4x + 5y \\ \text{subject to} & x, y \in \mathbb{Z} \\ & x, y \quad \geq 0 \\ & x + 4y \leq 10 \\ & 3x - 4y \leq 6 \end{array}$$

It first found a fractional optimum of $(4, \frac{3}{2})$. Since $y = 3/2 \notin \mathbb{Z}$, the algorithm divides the original problem into two regions with $y \geq 2$ and $y \leq 1$. Then it keeps searching and branching in the two regions until the integer optimum is found.



(a) Adding the second cut

(b) Optimum integer solution

Figure 2.3: Solving ILP with brand-and-bound method

Since an integer hull is the convex hull of all the integer points within a polyhedral set, a straightforward way of computing the integer hull is an enumeration of its integer points, followed by a convex hull computation. There is a family of studies focusing on enumerating or counting the lattice points of a given polyhedral set. A well-known theory on that latter subject was proposed by Pick [51]. In particular, the celebrated Pick's theorem provides a formula for the area of a simple polygon P with integer vertex coordinates, in terms of the number of integer points within P and on its boundary. In the 1990s, Barvinok [11] created an algorithm for counting the integer points inside a polyhedron, which runs in polynomial time, for a fixed dimension of the ambient space. Later studies such as [72] gave a simpler approach for lattice point counting, which divides a polygon into *right-angle triangles* and calculates the number of lattice points within each such triangle.

Verdoolaege, Seghir, Beyls, Loechner and Bruynooghe present in [64] a novel method for lattice point counting, based on Barvinok's decomposition for counting the number of integer points in a non-parametric polytope. In [57], Seghir, Loechner and Meister deal with the more general problem of counting the number of images by an affine integer transformation of the lattice points contained in a parametric polytope. In 2004, the software package `LattE` presented in [44] for lattice point enumeration offers the first implementation of Barvinok's algorithm. Other algorithms, such as [35] by Jing and Moreno Maza, compute an irredundant representation of the integer points of P in terms of "simpler" polyhedral sets, each of them given by a triangular-by-block system of linear inequalities.

`Normaliz` [15] is a program for the computation of Hilbert bases of rational cones and the normalizations of affine monoids. The Hilbert basis of a convex cone C is a minimal set of integer vectors such that every integer vector in C is a conical combination of the vectors in the Hilbert basis with integer coefficients. For example, let $C \in \mathbb{R}^d$ be a pointed (if $x, -x \in C$ then $x = 0$) convex cone and $L \in \mathbb{Z}^d$ be a lattice. There exists a unique minimal generating set, $H = x_1, \ldots, x_n$, of $C \cap L$, such that every point $x \in C \cap L$ has an integer conical combination:

$$x = \lambda_1 x_1 + \cdots + \lambda_n x_n, \lambda_1, \ldots, \lambda_n \in \mathbb{Z}, \lambda_1, \ldots, \lambda_n \geq 0$$

H is called the Hilbert basis of C. Computation of a Hilbert basis of a simplicial cone can be done by enumerating all lattice points in the fundamental paralleltopes.

`Polymake` [8] is a software system that includes several algorithms for convex hull computation and lattice points enumeration (e.g. `LattE` and `Normaliz`), then it uses these algorithms to compute the integer hulls of various kinds of input polyhedral sets.

There are also authors studying the relations between the vertices of P_I and the vertices of P. The authors of [33] provided an algorithm for finding the vertices of a polytope associated to the Knapsack integer programming problem. This algorithm computes boxes covering the input polyhedron and such that each box contains at most one vertex of P_I. Following that same approach, the authors [13] could give an upper bound on the number of those boxes, as well as a running estimate for enumerating the integer vertices of a polytope.

Another important work for computing the integer hull of polytopes is by Warwick Harvey [32]. There are some similarities between our works. For what we called a "sector" (see Section 3.1) we both use the idea of finding the "closest integer point" to the vertex as the bound for our computing. (This paper used a transformation of the original sector such that the closest integer point is on the origin.) While we didn't give a novice way of computing the integer hull of the corner regions, this paper gives a method based on continued fractions to find cuts and eventually find the integer hull. For a general 2D polyhedral set, this paper suggests adding the inequalities one by one to a sector, and use the same "continued fractions" method at each step, where we focus on finding a corner for each sector such that our algorithm can be done parallelly. And eventually, we generalized this partition and merge method to higher dimensions.

Since the integer hull P_I of P is completely determined by its vertices, it is natural to ask for the number of vertices in an integer hull of a polyhedron. The earliest study by Cook, Hartmann, Kannan and McDiarmid, in [19], shows that the number of vertices of P_I is related to the *size* (as defined in [56]) of the coefficients of the inequalities that describe P. Let $x = p/q$ be a rational number, p and q are coprime integers, the size of x is defined as

$$size(x) = 1 + \lceil (log(|p| + 1)) \rceil + \lceil (log(|q| + 1)) \rceil$$

For a linear inequality $a_n x_n + \cdots + a_1 x_1 + a_0 \leq 0$, its size is $\sum size(a_i)$ for $i = 0, \ldots, n$. For a polyhedron $P = \{\mathbf{x} \mid A\mathbf{x} \leq \vec{b}\}$ where matrix $A \in \mathbb{Q}^{m \times n}$ and vector $\vec{b} \in \mathbb{Q}^m$, Cook, Hartmann, Kannan and McDiarmid showed that the number of vertices of the integer hull of P is bounded over by $2m^n(6n^2\varphi)^{n-1}$ where φ is the maximum size of any of the m inequalities. More recent studies such as [65] and [13] use different approaches to reach similar or slightly improved estimates. We also discussed this question in our previous paper [49].

2.2.2 Parametric integer hull and periodicity

During our research, we found that the integer hulls of the parametric polyhedral sets have some kind of periodical property. Let's look at some studies that focus on "periodicity".

We recall the work of Eugème Ehrhart from his articles [21] and [22]. For each integer $n \geq 1$, Ehrhart defined the *dilation* of the polyhedron P by n as the polyhedron $nP = \{nq \in \mathbb{Q}^d \mid q \in P\}$. Ehrhart studied the number of lattice points in nP, that is:

$$i(P, n) \;=\; \#(nP \cap \mathbb{Z}^d) \;=\; \#\{q \in P \mid nq \in Z^d\}.$$

He proved that there exists an integer $N > 0$ and polynomials $f_0, f_1, \ldots, f_{N-1}$ such that $i(P, n) = f_i(n)$ if $n \equiv i \mod N$. The quantity $i(P, n)$ is called the *Ehrhart quasi-polynomial* of P, in the dilation variable n. Ehrhart's study on quasi-polynomials is focused on counting the lattice points and can be seen as a higher-dimensional generalization of Pick's theorem.

In [48] Meister presents a new method for computing the integer hull of a parameterized rational polyhedron. The author introduces a concept of periodic polyhedron (with facets given by equalities depending on periodic numbers). Hence, the word "periodic" means that the polyhedron can be defined in a periodic manner.

In our paper [49] we showed that the number and coordinates of the vertices of a parametric integer hull follows a pseudo-periodic pattern. The details are introduced in Chapter 3.

Chapter 3

On the pseudo-periodicity of the integer hull of parametric convex polygons

In this chapter, we show our observation that the number of vertices of the integer hull P_I of a convex polyhedral set P has a pseudo-periodicity behavior w.r.t vector $\vec{b}$ in the system $\mathbf{A}\,\mathbf{x} \leq \vec{b}$ that defines P. We state and prove explicitly the formulas for the pseudo-period.

While the arguments yielding to our main result are elementary, the proof is relatively long and technical. The first and main step is a study of the *pseudo-periodicity* of a parametric angular section, see Section 3.1. Since a convex polygon is an intersection of finitely many angular sectors, angular sectors are the building blocks of our main result, see Section 3.2, where the partitions of $V_1, \ldots, V_c$ of V, $V'_1, \ldots, V'_c$ of V', and the vectors $\vec{u}_1, \ldots, \vec{u}_c$ are explicitly given. This result and the ingredients of its proof lead us to propose a new algorithm for computing the integer hull of a rational convex polygon, see Section 3.4.

We consider a rational convex polygon (that is, a rational polyhedral set of dimension 2) given by a system of linear inequalities $A\vec{x} \leq \vec{b}$, where A is a matrix over $\mathbb{Z}$, with m rows and $d = 2$ columns, and $\vec{b}$ is an integer vector. The coordinates $b_1, \ldots, b_m$ of $\vec{b}$ are treated as parameters, while the coefficients of A have fixed values. We observe that for every $1 \leq i \leq m$, there exists a positive integer T_i so that, when each $b_1, \ldots, b_m$ is large enough, the vertex sets V and V' of the respectively integer hulls of

$$P := P(b_1, \ldots, b_{i-1}, b_i, b_{i+1}, \ldots, b_m) \text{ and } P' := P(b_1, \ldots, b_{i-1}, b_i + T_i, b_{i+1}, \ldots, b_m),$$

respectively, are in "simple" one-to-one correspondence. Here, simple, means that one can construct a partition $V_1, \ldots, V_c$ of V and a partition $V'_1, \ldots, V'_c$ of V', together with vectors $\vec{u}_1, \ldots, \vec{u}_c$ of $\mathbb{Z}^2$ so that every vertex of V'_i is the image of a vertex of V_i by the translation of $\vec{u}_i$, for all $1 \leq i \leq c$. Section 3.3 offers various examples, including animated images, which illustrate our result. Watching those animations requires to use a modern document viewer like Okular. The animations are also available at https://github.com/lxwangruc/parametric_integer_hull.

3.1 The integer hull of an angular sector

Lemma 1 is an elementary result which gives a necessary and sufficient condition for a line in the affine plane to have integer points. With Lemma 2, we show that every angular sector S

without integer points on its facets can be replaced by a angular sector S' with integer points on both of its facets and so that S and S' have the same integer hull. With Lemma 3, we perform another reduction step: we show how the computation of the integer hull of an angular sector with integer points on its facets can be reduced to that of the integer hull of a triangle with at least two integer vertices.

Theorem 1 is our main result specialized to the case of a parametric angular sector. In other words, Theorem 1 describes the pseudo-periodical phenomenon observed when varying one of the "right-hand side" parameters over a sufficiently large range of consecutive integer values. In fact, Theorem 1 precisely gives a formula for the period as well as a formula for transforming the integer hull of the parametric angular sector over a period.

Definition 1 *An* angular sector *in an affine plane is defined by the intersection of two half-planes whose boundaries intersect in a single point, called the* vertex *of that angular sector.*

Lemma 1 *In the affine plane, with Cartesian coordinates* (x, y)*, consider a line with equation* $ax + cy = b$ *where* a, b *and* c *are all integers so that there is no common divisor among them, that is,* $\gcd(a, b, c) = 1$*. Then, three cases arise:*

Case 1. If $a \neq 0$ *and* $c \neq 0$ *then there are integer points along the line if and only if* a *and* c *are coprime. Moreover, if* $\gcd(a, c) = 1$ *holds, then a point* (x, y) *on the line is integral if and only if we have:*

$$x \equiv \frac{b}{a} \mod c.$$

Case 2. If $a = 0$*, then* c *must equal to* 1 *for the line to have integer points. Moreover, if* $c = 1$*, then a point* (x, y) *on the line is integral if and only if* x *is an integer.*

Case 3. If $c = 0$*, then* a *must equal to* 1 *for the line to have integer points. Moreover, if* $a = 1$ *holds, then a point* (x, y) *on the line is integral if and only if* y *is an integer.*

Proof ▷ For Case 1, the y coordinate of a point (x, y) on the line must satisfy:

$$y = \frac{b - ax}{c}$$

For each integer x, the above y is an integer if and only if we have:

$$b - ax \equiv 0 \mod c.$$

Therefore, every point (x, y) on the line is an integer point if and only if x is an integer satisfying

$$b \equiv ax \mod c.$$

If $\gcd(a, c) = 1$ holds, then a is invertible modulo c and every integer x congruent to $\frac{b}{a} \mod c$ is a solution. If a and c are not coprime and if the above equation has a solution in x then $\gcd(a, b, c) = 1$ cannot hold, which is a contradiction. Therefore, the line admits integer points if and only if $\gcd(a, c) = 1$ holds. Moreover, when this holds, those points (x, y) satisfy:

$$x \equiv \frac{b}{a} \mod c,$$

For Case 2, with $a = 0$, the condition becomes $\gcd(b, c) = 1$ and the line now writes $cy = b$. Therefore, $\frac{b}{c}$ must be integer in order to have integer points on the line, which means c must equal to 1. Case 3 is similar to Case 2. ◁

Lemma 2 *In the affine plane, with Cartesian coordinates (x, y), let S be a angular sector defined by*

$$\begin{cases} a_1\, x + c_1\, y & \leq & b_1 \\ a_2\, x + c_2\, y & \leq & b_2. \end{cases}$$

Then, one can find another angular sector S', given by

$$\begin{cases} a_1\, x + c_1\, y & \leq & b'_1 \\ a_2\, x + c_2\, y & \leq & b_2 \end{cases}$$

such that $\frac{a_1}{g}$ and $\frac{c_1}{g}$ are coprime where $g = \gcd(a_1, c_1, b'_1) \geq 1$ and so that the integer hull of S' is the same as that of S.

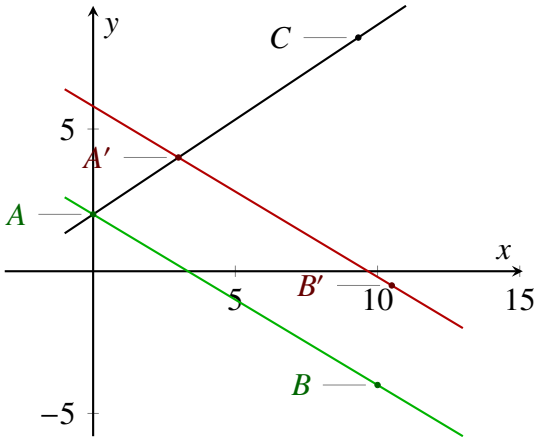


Figure 3.1: The integer hull of sector BAC is the same as that of sector $B'AC'$

PROOF ▷ Let A be the vertex of S. Let B (resp. C) be a point on the facet of S with equation $a_1\, x + c_1\, y = b_1$ (resp. $a_2\, x + c_2\, y = b_2$). The general idea is to construct S' by sliding A to the vertex A' of S' along the line (AC), with the facets of S' being given by $(A'C)$ and $(A'B')$ so that

1. $(A'B')$ and (AB) are parallel lines with no integer points between them, meanwhile
2. $(A'B')$ has integer points.

Details follow, including corner cases. Three cases arise:

Case 1. If a_1 and c_1 are non-zero integers and coprime, then, by Lemma 1, one can choose $S' = S$, thus $A' = A$ and $b'_1 = b_1$.

Case 2. If a_1 and c_1 are non-zero but a_1 is not coprime to c_1, then we have $g := \gcd(a_1, c_1) > 1$. Let C have coordinate (x_C, y_C). Two cases arise.

Case 2.1. If $y_C > -\frac{a_1}{c_1}\, x_C + \frac{b_1}{c_1}$ (as in Figure 3.1), then we can choose $b'_1 = \lceil \frac{b_1}{g} \rceil g$. Since $b'_1 > b_1$ and C is above (AB), the line $(A'B')$ is closer to C than (AB). We want to prove that there's no integer point between $(A'B')$ and (AB). Assume, by contradiction, there is an integer point X between $(A'B')$ and (AB). Then, a line $a_1\, x + c_1\, y = b''_1$ must pass through X such that $b_1 < b''_1 < b'_1$ and $b''_1 \mod g \equiv 0$ both hold. Since we chose $b'_1 = \lceil \frac{b_1}{g} \rceil g$, the integer b''_1 cannot exist. Therefore, there is no integer point between $(A'B')$ and (AB). Since all the integer points in S are also in S', the integer hull of S' must be the same as that of S.

Case 2.2. If $y_C < -\frac{a_1}{c_1}\, x_C + \frac{b_1}{c_1}$ holds, then we can choose $b'_1 = \lfloor \frac{b_1}{g} \rfloor g$. And the proof is similar to that of the previous case.

Case 3. Now we consider the case where either a_1 or c_1 is zero. Three cases arise:

Case 3.1. Assume $a_1 = 0$, if $\frac{b_1}{c_1}$ is an integer, then we can choose $b'_1 = b_1$, that is $S' = S$.

Case 3.2. If $a_1 = 0$ and $\frac{b_1}{c_1}$ is not an integer, then we can choose b_1 to be $\lfloor \frac{b_1}{c_1} \rfloor \times c_1$ or $\lceil \frac{b_1}{c_1} \rceil \times c_1$ depending on the relationship between C and (AB). Similarly to the discussion above, there is no integer point between (AB) and $(A'B')$.

Case 3.3. Finally, If $c_1 = 0$, we can use the same proof as when $a_1 = 0$, except we need to see if $\frac{b_1}{a_1}$ is an integer or not.

◁

Lemma 3 *In the affine plane, with Cartesian coordinates (x, y), let S be an angular sector defined by*

$$\begin{cases} a_1\, x + c_1\, y & \leq & b_1 \\ a_2\, x + c_2\, y & \leq & b_2 \end{cases}$$

where $\gcd(a_i, b_i, c_i) = 1$ *for* $i \in \{1, 2\}$ *and a_i, b_i, c_i are all integers (see Figure 3.2). We assume that both facets of S admit integer points. Let S_I be the integer hull of S and let A be the vertex of S. Let B and C be integer points on each facet of S, with $A \neq B$ and $A \neq C$, chosen so that there is no integer point between A and B (on the facet given by A and B) and no integer point between A and C (on the facet given by A and C). Then, one of the following properties hold:*

1. *A is an integer point and $S = S_I$,*
2. *A is not an integer point and the vertex set V of S_I is equal to the vertex set V' of the integer hull $\triangle_I$ of the triangle $\triangle ABC$.*

Proof ▷ We write $S = \triangle ABC \cup T$, where T is the convex hull of $\{B, C\} \cup (S \setminus \triangle ABC)$. Therefore, we have:

$$S_I = \triangle_I \cup T_I. \tag{3.1}$$

where T_I is the integer hull of T. The convex polygon T has 2 vertices (namely B and C, which are integer points) and 3 facets (the segment $[B, C]$ and two unbounded facets). From Lemma 1, the two unbounded facets of T have infinitely many integer points. It follows that $T_I = T$ holds. Therefore, with Equation (3.1) we deduce:

$$S_I = \triangle_I \cup T. \tag{3.2}$$

We consider two cases for the vertex A.

1. Assume that A is an integer point. Then, all points A, B, C are integer points, and since $\triangle ABC$ is pointed, we deduce $\triangle_I = \triangle ABC$. Thus, with Equation (3.2) we deduce $S = S_I$, as desired.
2. If A is not an integer point, if suffices to observe from Equation (3.2) that all vertices of T are vertices of $\triangle_I$ which yield the conclusion.

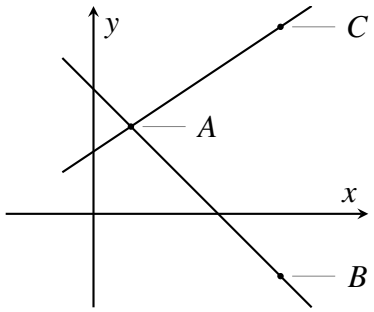


Figure 3.2: The vertices of the integer hull of $\triangle ABC$ are the same as that of section S

$\triangleleft$

Theorem 1 *Let us consider a parametric angular sector $S(b_i)$ defined by*

$$\begin{cases} a_1\,x + c_1\,y & \leq & b_1 \\ a_2\,x + c_2\,y & \leq & b_2 \end{cases}$$

where $\gcd(a_i, b_i, c_i) = 1$ *for* $i \in \{1, 2\}$ *and* a_i, b_i, c_i *are all integers,* $b_i \in \{b_1, b_2\}$*. Let* $S_I(b_i)$ *be the integer hull of* $S(b_i)$*. Then, there exists an integer* T *and a vector* $\vec{u}$ *such that* $S_I(b_i + T)$ *is the translation of* $S_I(b_i)$ *by* $\vec{u}$*.*

The integer T *is given by* $\frac{1}{g_2}\,|a_2\,c_1 - a_1\,c_2|$ *or* $\frac{1}{g_1}\,|a_2\,c_1 - a_1\,c_2|$ *and* $\vec{u} = (\frac{c_2\,T}{a_2\,c_1 - a_1\,c_2}, \frac{a_2\,T}{a_2\,c_1 - a_1\,c_2})$ *or* $\vec{u} = (\frac{c_1\,T}{a_1\,c_2 - a_2\,c_1}, \frac{a_1\,T}{a_1\,c_2 - a_2\,c_1})$ *for* $b_i = b_1$ *or* $b_i = b_2$ *respectively, where* $g_i = \gcd(a_i, c_i)$*. Note that* $a_2\,c_1 - a_1\,c_2 \neq 0$ *holds.*

PROOF $\triangleright$ Let A be the vertex of $S(b)$. Let $B(x_B, y_B)$ be a point such that

$$\begin{cases} a_1\,x_B + c_1\,y_B & = & b_1 \\ a_2\,x_B + c_2\,y_B & \leq & b_2 \end{cases}$$

and $C(x_C, y_C)$ be a point such that

$$\begin{cases} a_1\,x_C + c_1\,y_C & \leq & b_1 \\ a_2\,x_C + c_2\,y_C & = & b_2 \end{cases}$$

with $A \neq B$ and $A \neq C$. Without loss of generality, assume $b_i = b_2$ and $T = \frac{1}{g_1}|a_2\,c_1 - a_1\,c_2|$. Consider the angular sector S' is given by

$$\begin{cases} a_1\,x + c_1\,y & \leq & b_1 \\ a_2\,x + c_2\,y & \leq & b'_2 = b_2 + T \end{cases} \tag{3.3}$$

where A' is the vertex of S' and B' is on the facet of S' contained in the line (AB). We distinguish three cases.

Case 1. Assume that for each $i \in \{1, 2\}$, the integers a_i and c_i are non-zero coprime. With this assumption, the integer T becomes $|a_2\,c_1 - a_1\,c_2|$. Let D and E be two integer points where $\overrightarrow{AD} = t\,\overrightarrow{AC}$ and $\overrightarrow{AE} = k\,\overrightarrow{AB}$ where t and k are positive real numbers. Such points exist since a_i and c_i are coprime integers for $i \in \{1, 2\}$. We choose D and E so that there

is no other integer point on the segments $[A, D]$ and $[A, E]$. The points D' and E' are defined in a similar way on the angular sector S'(see Figure 3.3).

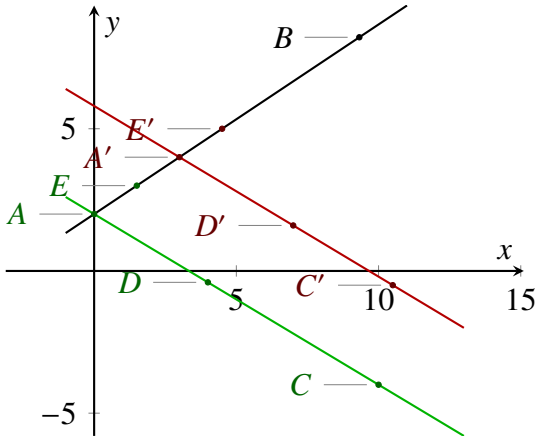


Figure 3.3: We want to prove that $\overrightarrow{AE} = \overrightarrow{A'E'}$ and $\overrightarrow{AD} = \overrightarrow{A'D'}$

We shall prove that the integer hull of $\triangle ADE$ is a translation of the integer hull of $\triangle A'D'E'$. This fact will follow from the following two vector equalities:

$$\overrightarrow{AE} = \overrightarrow{A'E'} \text{ and } \overrightarrow{AD} = \overrightarrow{A'D'}, \tag{3.4}$$

which we shall prove now. Let (x_A, y_A) be the coordinates of A. Since A is the vertex of S, we have:

$$x_A = \frac{b_2c_1 - b_1c_2}{a_2c_1 - a_1c_2} = x_1 + x_0$$

where $x_1 = \lfloor x_A \rfloor$ and $x_0 = x_A - x_1$. The coordinate of A', $(x_{A'}, y_{A'})$, would become

$$x_{A'} = \frac{b_2'c_1 - b_1c_2}{a_2c_1 - a_1c_2} = \frac{b_2c_1 - b_1c_2}{a_2c_1 - a_1c_2} + \frac{T\,c_1}{T} = x_A + c_1 = x_1 + c_1 + x_0$$

Proof of $\overrightarrow{AE} = \overrightarrow{A'E'}$. By definition of the point E, its x-coordinate has the form

$$x_E = x_A - x_0 + \Delta x_1 = x_1 + \Delta x_1, \tag{3.5}$$

where Δx_1 is a integer number. Since a_1 and c_1 are non-zero and coprime, from Lemma 1, we have:

$$\begin{aligned} x_E &\equiv \frac{b_1}{a_1} \mod c_1 \\ \Delta x_1 + x_1 &\equiv \frac{b_1}{a_1} \mod c_1 \\ \Delta x_1 &\equiv \frac{b_1}{a_1} - x_1 \mod c_1 \end{aligned}$$

Similarly, the x-coordinate $x_{E'}$ of E' satisfies $x_{E'} = x_{A'} - x_0 + \Delta x'_1 = x_1 + c_1 + \Delta x'_1$, where $\Delta x'_1$ is a integer number. From Lemma 1 we have:

$$\begin{aligned} x_{E'} &\equiv \frac{b_1}{a_1} \mod c_1 \\ \Delta x'_1 + x_1 + c_1 &\equiv \frac{b_1}{a_1} \mod c_1 \\ \Delta x'_1 &\equiv \frac{b_1}{a_1} - x_1 - c_1 \mod c_1 \\ \Delta x'_1 &\equiv \frac{b_1}{a_1} - x_1 \equiv \Delta x_1 \mod c_1 \end{aligned}$$

Since we choose E (resp. E') as close as possible to A (resp. A') we can assume that $\Delta x'_1 - \Delta x_1$ is less than c_1. Thus we have

$$\Delta x'_1 = \Delta x_1. \tag{3.6}$$

Therefore, we have

$$x_E - x_A = \Delta x_1 - x_0 = x_{E'} - x_{A'} \tag{3.7}$$

Since A, A', E, E' are all on the line $a_1 x + c_1 y = b_1$, we easily deduce:

$$y_E - y_A = \frac{-a_1 (\Delta x_1 - x_0)}{c_1} = y_{E'} - y_{A'} \tag{3.8}$$

With Equations 3.7 and 3.8 we have proved:

$$\overrightarrow{AE} = \overrightarrow{A'E'} \tag{3.9}$$

Proof of $\overrightarrow{AD} = \overrightarrow{A'D'}$. Let $x_D = x_A - x_0 + \Delta x_2 = x_1 + \Delta x_2$, where Δx_2 is a integer number. From Lemma 1 we know that

$$\begin{aligned} x_D &\equiv \frac{b_2}{a_2} \mod c_2 \\ \Delta x_2 + x_1 &\equiv \frac{b_2}{a_2} \mod c_2 \\ \Delta x_2 &\equiv \frac{b_2}{a_2} - x_1 \mod c_2 \end{aligned}$$

Similarly, let $x_{D'} = x_{A'} - x_0 + \Delta x'_2 = x_1 + c_1 + \Delta x'_2$, where $\Delta x'_2$ is a integer number.

From Lemma 1 we know that

$$\begin{aligned}
x_{D'} &\equiv \frac{b_2'}{a_2} \mod c_2 \\
\Delta x_2' + x_1 + c_1 &\equiv \frac{b_2'}{a_2} \mod c_2 \\
\Delta x_2' &\equiv \frac{b_2'}{a_2} - x_1 - c_1 \mod c_2 \\
\Delta x_2' &\equiv \frac{b_2 + (a_2 c_1 - a_1 c_2)}{a_2} - x_1 - c_1 \mod c_2 \\
\Delta x_2' &\equiv \frac{b_2}{a_2} + \frac{(a_2 c_1 - a_1 c_2)}{a_2} - x_1 - c_1 \mod c_2 \\
\Delta x_2' &\equiv \frac{b_2}{a_2} + c_1 - \frac{n a_1 c_2}{a_2} - x_1 - c_1 \mod c_2 \\
\Delta x_2' &\equiv \frac{b_2}{a_2} - \frac{a_1 c_2}{a_2} - x_1 \mod c_2 \\
\Delta x_2' &\equiv \frac{b_2}{a_2} - x_1 \equiv \Delta x_2 \mod c_2
\end{aligned}$$

Therefore, we have

$$x_D - x_A = \Delta x_2 - x_0 = x_{D'} - x_{A'} \tag{3.10}$$

Since A, D are all on the line $a_2 x + c_2 y = b_2$, we have

$$y_D - y_A = \frac{-a_2 (\Delta x_2 - x_0)}{c_2} \tag{3.11}$$

And A', D' are all on the line $a_2 x + c_2 y = b_2 + (a_2 c_1 - a_1 c_2)$, we have

$$y_D' - y_A' = \frac{-a_2 (\Delta x_2 - x_0)}{c_2} = y_D - y_A \tag{3.12}$$

From Equation 3.10 and 3.12 we know that

$$\overrightarrow{AD} = \overrightarrow{A'D'}$$

So far we have proved that $\overrightarrow{AE} = \overrightarrow{A'E'}$ and $\overrightarrow{AD} = \overrightarrow{A'D'}$ both hold, which imply:

$$\overrightarrow{AA'} = \overrightarrow{DD'} = \overrightarrow{EE'}. \tag{3.13}$$

With the assumption that D, E, D', E' are all integer points, we deduce that for any integer point F in $\triangle ADE$, there is an integer point F' in $\triangle A'D'E'$ such that

$$\overrightarrow{FF'} = \overrightarrow{AA'}. \tag{3.14}$$

Therefore, the integer hulls of $\triangle ADE$ is a translation of that of $\triangle A'D'E'$. Finally, with Lemma 3, we conclude that (in this Case 1) there exists a vector $\vec{u} = \vec{AA'} = (\frac{c_1 T}{a_1 c_2 - a_2 c_1}, \frac{a_1 T}{a_1 c_2 - a_2 c_1})$ and an integer $T = |a_2 c_1 - a_1 c_2|$ such that $S_I(b_2 + T)$ is a translation of $S_I(b_2)$ by $\vec{u}$.

Case 2. Consider the case where a_2 and c_2 are coprime integers while a_1 and c_1 are not coprime. From Lemma 2 we know that we can find another line $a_1x + c_1y = b'_1$ such that $\frac{a_1}{g}$ and $\frac{c_1}{g}$ are coprime, where $g = g_1 = \gcd(a_1, c_1, b'_1) \geq 1$. Then, we can claim that if we re-define (AB) as $\frac{a_1}{g_1}x + \frac{c_1}{g_1}y = \frac{b'_1}{g_1}$, we will not lose any integer point in the new sector comparing to our original sector. Therefore, we have reduced this second case to the previous one.

Case 3. Consider the case where a_1 and c_1 are coprime integers while a_2 and c_2 are not coprime. Similar to Case 2, we can find another line

$$a_2x + c_2y = b'_2 \tag{3.15}$$

such that $\frac{a_2}{g}$ and $\frac{c_2}{g}$ are coprime where $g = g_2 = \gcd(a_2, c_2, b'_2) \geq 1$, also the new line is not further to C than line (AC). Then we can say that if we re-define (AC) as $\frac{a_2}{g_2}x + \frac{c_2}{g_2}y = \frac{b'_2}{g_2}$, we will not lose any integer point in the new sector comparing to our original sector. Using Case 1 we can prove that $T = |\frac{a_2c_1}{g_2} - \frac{a_1c_2}{g_2}| = \frac{1}{g_2}|a_2\,c_1 - a_1\,c_2|$ w.r.t $\frac{b'_2}{g_2}$. Therefore, returning to the original b_2 (which is b'_2 as in Equation (3.15) plus some integer constant), we have $T = g_2\,|\frac{a_2c_1}{g_2} - \frac{a_1c_2}{g_2}| = |a_2c_1 - a_1c_2|$.

◁

3.2 The integer hull of a convex polygon

3.2.1 Case of a triangle

We start by a fundamental case, that of a triangle P, say defined by

$$\begin{cases} a_1\,x + c_1\,y & \leq & b_1 \\ a_2\,x + c_2\,y & \leq & b_2 \\ a_3\,x + c_3\,y & \leq & b_3 \end{cases}$$

with $\gcd(a_i, b_i, c_i) = 1$ for $i \in \{1, 2, 3\}$. We further assume $\gcd(a_i, c_i) = 1$ for $i \in \{1, 2, 3\}$, case to which one can reduce using Lemma 2. Note that P is the intersection of three angular sectors S_1, S_2, S_3 that are defined by

$$\begin{cases} a_1\,x + c_1\,y & \leq & b_1 \\ a_2\,x + c_2\,y & \leq & b_2, \end{cases}$$

$$\begin{cases} a_2\,x + c_2\,y & \leq & b_2 \\ a_3\,x + c_3\,y & \leq & b_3, \end{cases}$$

$$\begin{cases} a_1\,x + c_1\,y & \leq & b_1 \\ a_3\,x + c_3\,y & \leq & b_3. \end{cases}$$

respectively. Hence, we have $P = \bigcap_{i=1}^{3} S_i$.

Lemma 4 *Let P_I and S_{iI} be the integer hulls of P and S_i, respectively. Then, we have $P_I = \bigcap_{i=1}^{3} S_{iI}$.*

Proof ▷ Any integer point $A \in P_I$ must be in P, that is $A \in S_i$ for $i \in \{1,2,3\}$. Since A is an integer point, the fact $A \in S_i$ holds implies that $A \in S_{iI}$ holds as well. Therefore, the point A must be in the intersection of S_{iI} for $i \in \{1,2,3\}$. Similarly, any integer point $B \in \bigcap_{i=1}^{3} S_{iI}$ must satisfy $B \in S_i$ for $i \in \{1,2,3\}$. Thus we have $B \in \bigcap_{i=1}^{3} S_i = P$. Since B is an integer point in P, we deduce $B \in P_I$. ◁

Lemma 5 *For a line defined by $ax + cy = b$, with a, b, c non-zero integers, and* $\gcd(a,c) = 1$*, and for two points $A(x_A, y_A)$ and $B(x_B, y_B)$ on that line, there are at least two integer points on the segment $[A, B]$ if and only if we have: $|x_A - x_B| \geq |c|$.*

Proof ▷ By Lemma 1, and under the hypotheses of this lemma, a point on the line $ax + cy = b$ is an integer point if and only its x-coordinate satisfies

$$x \equiv \frac{b}{a} \mod c.$$

Therefore, the distance between the x-values of any two consecutive integer points should be c. The conclusion follows. ◁

Lemma 6 *Let V, V_1, V_2, V_3 be the vertex sets of $P_I, S_{1I}, S_{2I}, S_{3I}$, respectively. Then, we have $V = V_1 \cup V_2 \cup V_3$ and the pairwise intersections of the V_i's are all empty, if the following three inequalities all hold:*

$$\begin{cases} |\frac{b_2c_1 - b_1c_2}{a_2c_1 - a_1c_2} - \frac{b_1c_3 - b_3c_1}{a_1c_3 - a_3c_1}| & \geq & |c_1| \\ |\frac{b_2c_1 - b_1c_2}{a_2c_1 - a_1c_2} - \frac{b_2c_3 - b_3c_2}{a_2c_3 - a_3c_2}| & \geq & |c_2| \\ |\frac{b_3c_1 - b_1c_3}{a_3c_1 - a_1c_3} - \frac{b_2c_3 - b_3c_2}{a_2c_3 - a_3c_2}| & \geq & |c_3|. \end{cases} \tag{3.16}$$

Proof ▷ From Lemma 5, there are at least two integer points on each facet of the triangle, whenever the three inequalities of (3.16) all hold. To find the vertex sets V_i, $i \in \{1,2,3\}$, we need to find the closest integer points to each of the three vertices of P, see Lemma 3. Since there are at least two integer points on each facet, then the triangles we find for S_i according to Lemma 3 do not overlap with each other. Therefore, $V = V_1 \cup V_2 \cup V_3$ and the pairwise intersections of the V_i's are all empty. ◁

Theorem 2 *Let $P(b_i)$ be a parametric triangle where $b_i \in \{b_1, b_2, b_3\}$, and $P_I(b_i)$ be the integer hull of $P(b_i)$. We say that $|b_i|$ is* large enough *whenever the following three inequalities all hold:*

$$\begin{cases} |\frac{b_2c_1 - b_1c_2}{a_2c_1 - a_1c_2} - \frac{b_1c_3 - b_3c_1}{a_1c_3 - a_3c_1}| & \geq & |c_1|. \\ |\frac{b_2c_1 - b_1c_2}{a_2c_1 - a_1c_2} - \frac{b_2c_3 - b_3c_2}{a_2c_3 - a_3c_2}| & \geq & |c_2|. \\ |\frac{b_3c_1 - b_1c_3}{a_3c_1 - a_1c_3} - \frac{b_2c_3 - b_3c_2}{a_2c_3 - a_3c_2}| & \geq & |c_3|. \end{cases}$$

There exists an integer T and 3 vectors $\vec{u}, \vec{v}$ and $\vec{w}$, such that for $|b|$ large enough, the integer hull $P_I(b + T)$ can be obtained from $P_I(b)$ as follows.

As defined above, denoting S_1, S_2, S_3 the angular sectors of $P(b)$ and by S_{1I}, S_{2I}, S_{3I} their respective integer hulls, the integer hull of $P(T + b)$ is the intersection of $f_u(S_{1I})$, $f_v(S_{2I})$, $f_w(S_{3I})$

where f_u, f_v, f_w are the translations of vectors $\vec{u}, \vec{v}$ and $\vec{w}$ respectively. Specifically, when $b = b_1$ we have

$$T = \text{lcm}(\frac{1}{g_2}|a_2\,c_1 - a_1\,c_2|, \frac{1}{g_3}|a_3\,c_1 - a_1\,c_3|)$$

Similar results apply to other b_i as well.

PROOF ▷ Without loss of generality, assume $b_i = b_1$. For S_1, defined by $a_1\,x + c_1\,y \geq b_1, a_2\,x + c_2\,y \geq b_2$, we know from Theorem 1 that there exists an integer

$$T_1 = \frac{1}{g_2}|a_2\,c_1 - a_1\,c_2|$$

and a vector

$$\vec{h_1} = (\frac{c_2\,T_1}{a_2\,c_1 - a_1\,c_2}, \frac{a_2\,T_1}{a_2\,c_1 - a_1\,c_2})$$

such that $S_{1I}(b_1 + T_1)$ is the translation of $S_{1I}(b_1)$ by $\vec{h_1}$.

Similarly, for S_3, defined by $a_1\,x + c_1\,y \geq b_1, a_3\,x + c_3\,y \geq b_3$, there exists an integer

$$T_3 = \frac{1}{g_3}|a_3\,c_1 - a_1\,c_3|$$

and a vector

$$\vec{h_3} = (\frac{c_3\,T_3}{a_3\,c_1 - a_1\,c_3}, \frac{a_3\,T_3}{a_3\,c_1 - a_1\,c_3})$$

such that $S_{3I}(b_1 + T_3)$ is the translation of $S_{3I}(b_1)$ by $\vec{h_3}$.

As for S_2, it is not affected by the change in b_1, which means for any integer k, $S_{2I}(b_1 + k)$ is the same as $S_{2I}(b_1)$, in other words, $S_{2I}(b_1 + k)$ is the translation of $S_{2I}(b_1)$ by the zero vector.

Combining the three sectors, we have proved that for $T = \text{lcm}(T_1, T_3)$, and the three vectors $\vec{u} = \frac{T}{T_1}\vec{h_1}, \vec{v} = \frac{T}{T_2}\vec{h_3}, \vec{w} = (0, 0)$, the sets $f_u(S_{1I}(b_1))$, $f_v(S_{2I}(b_1))$, $f_w(S_{3I}(b_1))$ are the same as the sets $S_{1I}(b_1 + T)$, $S_{2I}(b_1 + T)$,$S_{3I}(b_1 + T)$ respectively. Also as we have proved in Lemma 4 that $P_I = \bigcap_{i=1}^{3} S_{iI}$. Therefore, $P_I(T + b_1)$ is the intersection of $f_u(S_{1I}(b_1))$, $f_v(S_{2I}(b_1))$, $f_w(S_{3I}(b_1))$ where f_u, f_v, f_w are the translations of vectors $\vec{u}, \vec{v}, \vec{w}$ respectively.

The proofs for $b_i = b_2$ or $b_i = b_3$ are similar. ◁

3.2.2 Convex polygon of arbitrary shape

With Theorem 2 proved, we can extend it to a convex polygon of any shape.

Theorem 3 *Let $P(b)$ be a parametric polygon given by*

$$a_i\,x + c_i\,y \leq b_i$$

where $i \in \{1, \ldots, n\}$ and the parameter $b \in \{b_1, \ldots, b_n\}$ and $P_I(b)$ be the integer hull of $P(b)$. Specifically, $a_i\,x + c_i\,y \leq b_i$ and $a_{i+1}\,x + c_{i+1}\,y \leq b_{i+1}$ define an angular sector S_i of P, for all $1 \leq i \leq n$, with the convention $i + 1 = 1$ if $i = n$. Then, there exist an integer T and n vectors $\vec{v_1}, \ldots, \vec{v_n}$, such that, for $|b|$ large enough, $P_I(b + T)$ can be obtained from $P_I(b)$ as follows.

Denoting by S_{iI} the integer hull of the angular sector S_i, for all $1 \leq i \leq n$, the integer hull $P_I(b+T)$ of $P(T+b)$ is the intersection of $f_{v_i}(S_{iI})$, where f_{v_i} are the translations of vectors $\vec{v_i}$. Specifically, for $1 \leq m \leq n$, when $b = b_m$ we have

$$T = \mathrm{lcm}(\frac{1}{g_{m-1}}\,|a_{m-1}\,c_m - a_m\,c_{m-1}|, \frac{1}{g_{m+1}}\,|a_{m+1}\,c_m - a_m\,c_{m+1}|)$$

here we have $m-1 = n$ when $m = 1$, and $m+1 = 1$ when $m = n$.

The condition $|b_m|$ large enough means that all of the following inequalities hold:

$$\left\{\begin{array}{lcl} |\frac{b_{m+1}c_m - b_m c_{m+1}}{a_{m+1}c_m - a_m c_{m+1}} - \frac{b_m c_{m-1} - b_{m-1}c_m}{a_m c_{m-1} - a_{m-1}c_m}| & \geq & |c_m|. \\ |\frac{b_{m+1}c_m - b_m c_{m+1}}{a_{m+1}c_m - a_m c_{m+1}} - \frac{b_{m+1}c_{m+2} - b_{m+2}c_{m+1}}{a_{m+1}c_{m+2} - a_{m+2}c_{m+1}}| & \geq & |c_{m+1}|. \\ |\frac{b_m c_{m-1} - b_{m-1}c_m}{a_m c_{m-1} - a_{m-1}c_m} - \frac{b_{m-2}c_{m-1} - b_{m-1}c_{m-2}}{a_{m-2}c_{m-1} - a_{m-1}c_{m-2}}| & \geq & |c_{m-1}|. \end{array}\right. \tag{3.17}$$

Proof ▷ Without loss of generality, let's assume $b = b_1$. For S_1, defined by $a_1\,x + c_1\,y \geq b_1, a_2\,x + c_2\,y \geq b_2$, we know from Theorem 1 that we choose the integer

$$T_1 = \frac{1}{g_2}\,|a_2\,c_1 - a_1\,c_2|$$

and the vector

$$\vec{h_1} = (\frac{c_2\,T_1}{a_2\,c_1 - a_1\,c_2}, \frac{a_2\,T_1}{a_2\,c_1 - a_1\,c_2})$$

such that $S_{1I}(b_1+T_1)$ is the translation of $S_{1I}(b_1)$ by $\vec{h_1}$. Similarly, for S_n, defined by $a_1\,x+c_1\,y \geq b_1, a_n\,x + c_n\,y \geq b_n$, we choose the integer

$$T_n = \frac{1}{g_n}\,|a_n\,c_1 - a_1\,c_n|$$

and the vector

$$\vec{h_n} = (\frac{c_n\,T_n}{a_n\,c_1 - a_1\,c_n}, \frac{a_n\,T_n}{a_n\,c_1 - a_1\,c_n})$$

such that $S_{nI}(b_1 + T_n)$ is the translation of $S_{nI}(b_1)$ by $\vec{h_n}$.

As for each $j \in \{2, \ldots, n-1\}$, the angular sector S_j is not effected by the change in b_1, which means for any integer k, the sets $S_{jI}(b_1 + k)$ and $S_{jI}(b_1)$ are the same, in other words, $S_{jI}(b_1 + k)$ is the translation of $S_{jI}(b_1)$ by the zero vector.

Combining all n sectors, we have proved that for $T = \mathrm{lcm}(T_1, T_n)$, and the n vectors $\vec{v_1} = \frac{T}{T_1}\vec{h_1}, \vec{v_n} = \frac{T}{T_n}\vec{h_n}, \vec{v_j} = (0,0)$ for $j \in \{2, \ldots, n-1\}$, the set $f_{v_i}(S_{iI}(b_1))$, for $i \in \{1, \ldots, n\}$, is the same as the set $S_{iI}(b_1 + T)$.

Also as we have proved in Lemma 4, we have $P_I = \bigcap_{i=1}^{n} S_{iI}$. Therefore, $P_I(T + b_1)$ is the intersection of $f_{v_i}(S_{iI}(b_1))$ where f_{v_i} is the translation of vector $\vec{v_1}$. ◁

3.3 Examples

In this section, we give some examples to show the periodic phenomenon that we proved in the previous sections.

Consider a simple parametric polytope. Figure 3.4 shows a triangle $P(b)$ defined by

$$\begin{cases} x - 4y & \leq & -4 \\ -2x + y & \leq & 0 \\ x + y & \leq & b \end{cases}$$

First, we look at the angular sector $S(b)$ given by $x-4y \leq -4$ and $x+y \leq b$ (see Figure 3.4a, also available at https://github.com/lxwangruc/parametric_integer_hull.). According to Theorem 1, the integer hull of $S(b-5n)$ is a transformation of that of $S(b-5(n-1))$ by $(4,\vec{1})$ for any $n \geq 1$.

We can extend this observation to the triangle $P(b)$. Using Theorem 2 when $|b| \geq 11$, the integer hull of $P(b-15n)$ is a translation of $P(b-15(n-1))$ by $(0,\vec{0}), (12,\vec{3}), (5,\vec{10})$ for $n \geq 1$.

Figure 3.4b shows the integer hulls of $P(b)$ where $-26 \leq b \leq -11$, the points in the figure are the vertices of the integer hull. We can see that the integer hull of $P(-26)$ is a translation of that of $P(-11)$.

(a) Integer hull of an angular sector

(b) Integer hull of a triangle

Figure 3.4: The periodic phenomenon in a simple example. The dots are the vertices of the integer hull.

Consider a more complicated example. In order to have a clear view, we only look at one angular sector $S(b)$ given by

$$\begin{cases} -103x + 43y & \leq & 172 \\ 59x + 83y & \leq & b \end{cases}$$

By Theorem 1, we have $T = 11086$ and the integer hull of $S(b+nT)$ is a transformation of that of $S(b)$. We pick $b = 90 \times 83$ and $n = 83$ so that the integer hull of $S(83 \times (90+11086+i))$ is a transformation of that of $S(83 \times (90+i))$. Figure 3.5 shows the first 15 iterations of the vertices of the integer hull of each sector.

(a) (b)

Figure 3.5: A more complicated example. The red dots are the vertices of the integer hull of the sector.

3.4 A new integer hull algorithm

The most natural application of our conclusions from the previous sections is to use the periodic phenomenon to study the integer hull of a parametric polyhedron. Before discussing that application, we propose an algorithm, based on the results of Section 3.1, for computing the integer hull of a rational convex polygon P.

We assume that P has at least one integer point and write $P = \{\mathbf{x} \mid A\mathbf{x} \leq \vec{b}\}$, for a matrix $A \in \mathbb{Z}^{m\times 2}$ and a vector $\vec{b} \in \mathbb{Z}^m$, where m is a positive integer. It is easy to determine:

1. the equations of the facets $F_1, \ldots, F_f$ of P, each of them having a form $a_i\, x + c_i\, y \geq b_i$. Note that if a facet has no integer point, we use Lemma 2 to replace it with a new facet that has integer points, without modifying the integer hull of P.
2. the coordinates of the vertices $V_1, \ldots, V_f$ of P, so that $[V_i, V_{i+1}] = F_i$, with the conventions $V_{f+1} = V_1$ and $F_0 = F_f$.

To compute the integer hull P_I of P, we compute its vertices. We transform $V = \{V_1, \ldots, V_f\}$ so that it becomes the vertex set of P_I. We visit each vertex V_i of V and do the following:

1. if the coordinates of V_i are integers, we keep V_i in V,
2. otherwise:
   (a) we compute the vertex set U of the integer hull of the angular sector defined by F_{i-1} and F_i with V_i as its vertex. In the current implementation of the algorithm, we first find the integer points A, B on F_i and F_{i-1} that are closest to V_i. If no such A or B exists, we pick $A = V_{i-1}$ and $B = V_{i+1}$. Then we use the *triangle rasterisation algorithm* [70] on $\triangle V_iAB$ to find the integer points that are likely to be the vertices of the integer hull of the angular sector. That is, we find all the integer points that are closest to the edges $[V_iA]$ and $[V_iB]$. Then, we compute the *convex hull* [69] of all the possible integer points plus A, B to find the vertex set U.
   (b) we replace V_i with U.

If the given P is a parametric convex polygon, where b_i is unknown, we propose the following steps to compute the vertices of P_I:

1. determine the smallest $|b_i|$ so that the constraints in Theorem 3 hold (See inequali-

ties 3.17).

2. compute the period T and the transformation vectors in Theorem 3
3. compute the integer hull of every non-parametric polyhedron in this period.
4. when the values of the parameters are available, using the corresponding solution from the previous step and the vectors from step 2 to compute the integer hull of the P with the given parameters.

Note that we can finish the first three steps "off-line". Once the parameters are given the only computation that needs to be done is the translations which could be done in linear time. This method is both time and space efficient if the period T is short.

Chapter 4

Computing the integer hull of polyhedral sets

In this chapter, we discuss a new algorithm for computing the integer hull P_I of a rational polyhedral set P, together with its implementation as a new command of MAPLE's `PolyhedralSets` library [52] as well as in the C programming language.

Our presentation of this new algorithm focuses on the two-dimensional and three-dimensional cases, see Sections 4.2 and 4.3, respectively. However, Section 4.1 highlights the core procedures of algorithm without restricting to the 2D or 3D cases. Moreover, the concluding section, namely Section 4.5 states our algorithm in arbitrary dimension.

We present benchmarks for the implementation of our algorithm in Section 4.4. Our results show that our algorithm is very efficient comparing to the well known library Normaliz [15] especially when the input polyhedral set is large in volume.

Our algorithm has three main steps:

Normalization: during this step, we construct a new polyhedral set Q from P as follows. Consider in turn each facet F of P:

1. if the hyperplane H supporting F contains an integer point, then H is a hyperplane supporting a facet of Q,
2. otherwise one slides H towards the center of P along the normal vector of F, stopping as soon as one hits a hyperplane H' containing an integer point, then making H' a hyperplane supporting a facet of Q.

The resulting polyhedral set Q clearly has the same integer hull as P; computing Q is a preparation phase for the following step.

Partitioning: during this step, we search for integer points inside Q so as to partition P into smaller polyhedral sets, the integer hulls of which can easily be computed. We observe that every vertex of Q which is an integer point is also a vertex of Q_I. Now, for every vertex v of Q which is not an integer point we look, on each facet F to which v belongs, for an integer point $C_{v,F}$ that is "close" to v (ideally as close as possible to v). All the points $C_{v,F}$ together with the vertices of Q are used to build that partition of Q. Each part of the partition is a polyhedron R which:

1. either has integer points as vertices (making the computation of the integer hull R_I trivial),
2. or has a small volume so that any algorithm (including exhaustive search) can be applied to compute R_I.

Merging: Once the integer hull of each part of the partition is computed and given by the list of its vertices, an algorithm for computing the convex hull of a set points, such as `QuickHull` [10], can be applied to deduce P_I.

4.1 Two core constructions of our algorithm

In this section, we emphasize two constructions supporting respectively the *normalization* and *partitioning* steps of our algorithm. Both constructions deal with "algebraic aspects", that is, with the fact that we are solving for the integer solutions of a system of linear inequalities. These two constructions are inspired respectively by [49] and [35].

4.1.1 Normalization

Considering the rational polyhedron $P = \{\mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} \leq \vec{b}\}$, with the notations of Section 2.1, we observe that one can compute a vector $\vec{e} \in \mathbb{Z}^m$ so that the rational polyhedron $Q = \{\mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} \leq \vec{e}\}$ satisfies:

1. $P_I = Q_I$, and
2. the supporting hyperplane of every facet of Q has at least one integer point. Notice that this does not necessarily means that the new facet has an integer point.

In the introduction, the construction of Q is referred as the *normalization* step. We construct Q from P as follows:

1. consider each facet F of P in turn; if the hyperplane H supporting F does not contain an integer point, then one "slides" H towards the center of P along the normal vector of F, stopping as soon as a hyperplane H' containing an integer point is reached, otherwise keep H unchanged;
2. the resulting polyhedron is Q, for which rational consistency must be checked, which can be done efficiently using a method based on linear programming.

The "sliding process" described above informally is performed as follows. Let the equation below define the hyperplane H supporting F:

$$a_1x_1 + \cdots + a_dx_d = b, \tag{4.1}$$

where $a_1, \ldots, a_d, b$ can be assumed to be integers. The fact that $\mathbb{Z}$ is an Euclidean domain (and thus a principal ideal domain) implies that H has integer points if and only if we have:

$$\gcd(a_1, \ldots, a_d) \mid b. \tag{4.2}$$

If the hyperplane H supporting F does not have integer points and P is included in the half-space $a_1x_1 + \cdots + a_dx_d \leq b$, then H' is given by:

$$a_1x_1 + \cdots + a_dx_d = g\,\lfloor \frac{b}{g} \rfloor, \tag{4.3}$$

with $g := \gcd(a_1, \ldots, a_d)$.

Summing things up, we denote by `Normalization`(P) a function call returning the polyhedron Q.

4.1.2 Partitioning

The other algebraic construction in our algorithm supports the *partition* step briefly explained in the introduction. The underlying question is the following: given a vertex v of P which is not an integer point and given a facet F of P to which v belongs, find on F an integer point $C_{v,F}$, if any, which is "close" to v (ideally as close to v as possible).

If P is two-dimensional, thus, if F is one-dimensional, then the question is easily answered by elementary arguments, see our previous paper [49]. If P has dimension $d \geq 3$, thus, if F has dimension $d - 1$, then we take advantage of the *Hermite normal form* of a matrix. In the sequel of this section, we review this concept. and use it to compute the integer hull of a facet of a polyhedron. Finally, we solve the question of finding an integer point $C_{v,F}$ on F (if any) as close as possible to v.

Hermite normal form.

Consider a positive integer $p \leq d$ and a linear system $C\mathbf{x} = \mathbf{s}$ where $C \in \mathbb{Z}^{p \times d}$ is a full row-rank matrix and $\mathbf{s} \in \mathbb{Z}^p$ is a vector. There exists a uni-modular matrix $U \in \mathbb{Z}^{d \times d}$ so that $CU = [\mathbf{0}H]$ where $\mathbf{0} \in \mathbb{Z}^{p \times (d-p)}$ is the null matrix and H is the column-style Hermite normal form of C. We write $U = [U_L U_R]$ where $U_L \in \mathbb{Z}^{d \times (d-p)}$ and $U_R \in \mathbb{Z}^{d \times p}$. Therefore, the matrix $H \in \mathbb{Z}^{p \times p}$ is non-singular and the following properties hold:

1. $C\mathbf{x} = \mathbf{s}$ has integer solutions if and only if $H^{-1}\mathbf{s}$ is an integer vector,
2. every integer solution of $C\mathbf{x} = \mathbf{s}$ has the form $U_R H^{-1}\mathbf{s} + U_L\mathbf{z}$, where $\mathbf{z} \in \mathbb{Z}^{d-p}$ is arbitrary.

Determining the integer hull of a facet.

Let $\vec{c}^t\mathbf{x} = s$ be the equation of the hyper-plane supporting F, thus with $\vec{c}^t \in \mathbb{Z}^d$ and $s \in \mathbb{Z}$. Let $U \in \mathbb{Z}^{d \times d}$ be a uni-modular matrix so that $\vec{c}^t U = [\mathbf{0}H]$ where $\mathbf{0} \in \mathbb{Z}^{1 \times (d-1)}$ is the null matrix and H is the column-style Hermite normal form of $\vec{c}^t$ regarded as a matrix of $\mathbb{Z}^{1 \times d}$. We write $U = [U_L U_R]$ where $U_L \in \mathbb{Z}^{d \times (d-1)}$ and $U_R \in \mathbb{Z}^{d \times 1}$. Let $\mathbf{v} := U_R H^{-1} s$. Then, from the above paragraph on Hermite Normal Form, we know that the integer points of the hyper-plane supporting F are of the form $\mathbf{x} = \mathbf{v} + U_L\mathbf{z}$ where $\mathbf{z} \in \mathbb{Z}^{d-1}$ is arbitrary. The facet F is described by a system of linear inequalities in $\mathbb{Q}^d$ with $\mathbf{x}$ as unknown vector. Substituting $\mathbf{v} + U_L\mathbf{z}$ for $\mathbf{x}$ yields a system of linear inequalities in $\mathbb{Q}^{d-1}$ (with $\mathbf{z}$ as unknown vector) representing a rational polyhedron $G \subseteq \mathbb{Q}^{d-1}$. With these notations and hypotheses, we have the following.

Theorem 4 *The vertices of the integer hull G_I of G are in one-to-one correspondence with the vertices of the integer hull F_I of F via the map*

$$R_F : \left\{ \begin{array}{ccc} \mathbb{Q}^{d-1} & \rightarrow & \mathbb{Q}^d \\ \mathbf{z} & \longmapsto & \mathbf{x} = \mathbf{v} + U_L\mathbf{z}. \end{array} \right. \tag{4.4}$$

In particular, we have $R_F(G_I) = F_I$.

PROOF ▷ The proof follows from seven claims.
Claim 1: R_F is injective. Indeed, the matrix U is uni-modular, thus the columns of U are linearly independent, and the map $\mathbf{z} \longmapsto U_L\mathbf{z}$ is injective.

Claim 2: The image of R_F is F. Since R_F is an injective affine map from $\mathbb{Q}^{d-1}$ to $\mathbb{Q}^d$, it follows that the image of R_F is an affine space of dimension $d-1$. Therefore, in order to prove the claim, it suffices to prove that for every $\mathbf{z} \in \mathbb{Z}^{d-1}$ we have $R_F(\mathbf{z}) \in F$. Since $F \cap \mathbb{Z}^d \neq \emptyset$ (as a consequence of the normalization step of our algorithm) there exists $\mathbf{z}_0 \in \mathbb{Z}^{d-1}$ so that $\mathbf{x}_0 := \mathbf{v} + U_L\mathbf{z}_0 \in F \cap \mathbb{Z}^d$ holds. Let $\mathbf{z} \in \mathbb{Z}^{d-1}$. Define $\mathbf{x} := R_F(\mathbf{z})$. We have:

$$\mathbf{x} = \mathbf{v} + U_L\mathbf{z}_0 + U_L(\mathbf{z} - \mathbf{z}_0) = \mathbf{x}_0 + U_L(\mathbf{z} - \mathbf{z}_0).$$

We deduce:

$$\vec{c}^t\mathbf{x} = \vec{c}^t\mathbf{x}_0 + \vec{c}^t U_L(\mathbf{z} - \mathbf{z}_0) = s + 0 = s,$$

which proves that $R_F(\mathbf{z}) \in F$ holds.

Claim 3: $R_F^{-1}(H)$ is a half-space of $\mathbb{Q}^{d-1}$ for any half-space H of $\mathbb{Q}^d$. Indeed, for any $\mathbf{x} \in \mathbb{Q}^d$ of the form $\mathbf{v} + U_L\mathbf{z}$, with $\mathbf{z} \in \mathbb{Q}^{d-1}$, we have

$$\vec{a}^t\mathbf{x} \geq b \quad \Longleftrightarrow \quad \vec{a}^t U_L\mathbf{z} \geq b - \vec{a}^t\mathbf{v},$$

where $H : \vec{a}^t\mathbf{x} \geq b$ is an arbitrary half-space of $\mathbb{Q}^d$.

Claim 4: The integer points of the hyper-plane supporting F are in one-to-one correspondence with the integer points of $\mathbb{Z}^{d-1}$. This claim follows directly from the properties of the Hermite Normal Form.

Claim 5: $R_F^{-1}(S)$ is a polyhedron of $\mathbb{Q}^{d-1}$ for any polyhedron S of $\mathbb{Q}^d$. Indeed, let $S := \cap_i H_i$ be a polyhedron of $\mathbb{Q}^{d-1}$ given as the intersection of finitely many half-spaces of $\mathbb{Q}^{d-1}$. We have

$$R_F^{-1}(S) = R_F^{-1}(\cap_i H_i) = \cap_i R_F^{-1}(H_i).$$

The conclusion follows with Claim 3.

Claim 6: $R_F(T)$ is a polyhedron of $\mathbb{Q}^d$ for any polyhedron T of $\mathbb{Q}^{d-1}$. The proof is similar to that of Claim 5.

Claim 7: We have: $R_F(G_I) = F_I$. Let $\mathcal{S}$ be the set of all polyhedra of $\mathbb{Q}^d$ containing $F \cap \mathbb{Z}^d$. Let $\mathcal{T}$ be the set of all polyhedra of $\mathbb{Q}^{d-1}$ containing $G \cap \mathbb{Z}^d$, where $G = R_F^{-1}(F)$. Then, by definition of F_I and G_I, we have:

$$F_I = \bigcap_{S \in \mathcal{S}} S \quad \text{and} \quad G_I = \bigcap_{T \in \mathcal{T}} T.$$

From Claim 5, we have:

$$R_F^{-1}(F_I) = \bigcap_{S \in \mathcal{S}} R_F^{-1}(S) \supseteq \bigcap_{T \in \mathcal{T}} T = G_I.$$

From Claim 6, and since R_F is injective,we have:

$$R_F(G_I) = \bigcap_{T \in \mathcal{T}} R_F(T) \supseteq \bigcap_{S \in \mathcal{S}} S = F_I.$$

Therefore, we have $R_F(G_I) = F_I$. Now we can prove the theorem. Since R_F is a bijective affine map from $\mathbb{Q}^{d-1}$ to F, it maps affine subspaces of dimension $0 \leq d' < d$ of $\mathbb{Q}^{d-1}$ to affine subspaces of dimension d' of F. Combined with Claims 5 and 6, this latter observation implies that faces of dimension $0 \leq d' < d$ of G_I are mapped to faces of dimension d' of F_I. Therefore, the vertices of G_I are in one-to-one correspondence with the vertices of F_I. $\lhd$

Theorem 4 shows that one can reduce the computation of the vertices of F_I to computing the vertices of G_I. Based on that observation, we denote by `HNFProjection`(F, d) a function call returning the ordered pair (G, R_F).

Finding an integer point $C_{v,F}$ on F (if any) close to v.

Let us return now to the question of finding an integer point $C_{v,F}$ on F (if any) as close as possible to v. A second consequence of Theorem 4 is that we can compute an integer point $C_{v,F}$ simply by choosing a point $R_F(W)$ at minimum Euclidean distance to v, where W ranges in the set of the vertices of G_I. As mentioned, such a point may not be an integer point of F at minimum Euclidean distance to v, but if F is large enough (that is, if its area is large enough) then $C_{v,F}$ is a good approximate solution to this optimization problem.

4.2 Integer hull of a 2D polyhedral set

In this section, we present our algorithm for computing the integer hull of a 2D polyhedral set. We first give a high-level overview of the algorithm, then we present its sub-routines, followed by a more precise presentation of the general algorithm together with the implementation details.

As mentioned in the introduction of this chapter, our main idea is to partition the input 2D-polyhedral set into several smaller polyhedral sets, compute the integer hulls of those and return the convex hull of the union of all these integer hulls. Recall from Section 2.1, that the integer hull of a polyhedron P is another polyhedron P_I whose vertices are all integer points. Therefore, given a polyhedral set P which is not an integer hull, we aim at replacing each fractional (i.e. non-integer) vertex v with some integer points in the neighborhood of v so that this replacement process allows us to deduce P_I.

To replace such a fractional vertex v, we inspect the region of P forming a "corner" around v. Such a corner region is actually given by a "small" polyhedron (often a triangle), for which the integer hull is computed by a straightforward method. Other than these corners, there is the "central" region of P. By construction, this central region is a polyhedral set with integer vertices, for which no computation is required. Our goal is, of course, to maximize the area of that central region in order to minimize the work needed in computing the integer hulls of the corner regions.

To implement the above strategy, we propose the following method to partition the input. First, we normalize the input using the procedure call `Normalization(P)` described in Section 4.1.1. Next, for each fractional vertex v, we find the closest integer point to v on each of its adjacent facets. For a 2D polyhedral set, each vertex has exactly two adjacent facets, therefore, two "closest integer points". We partition the input polyhedron by means of its integer vertices and the "closest integer points" of its non-integer vertices.

In most cases a corner region would be a triangle given by a fractional vertex v and its two closest integer points. In some special cases, when a facet contains no integer point, we combine adjacent vertices and their closest integer points to form a corner region of quadrilateral shape (or, in rare cases, of more complex shape) instead of a triangle. Finally, the central region is formed by the integer vertices of the input polyhedron P and the closest integer points of the non-integer vertices of P. The details of the sub-routines, as well as the general algorithm, are given in the following sections.

4.2.1 Algorithm

In this section, we consider an input polyhedral set P defined by a system of linear inequalities

$$\begin{cases} a_{11}x_1 + a_{21}x_2 & \leq & b_1 \\ a_{12}x_1 + a_{22}x_2 & \leq & b_2 \\ \quad \cdots & & \\ a_{1n}x_1 + a_{2n}x_2 & \leq & b_n, \end{cases}$$

where $\gcd(a_{1i}, a_{2i}, b_i) = 1$ for $i \in \{1, \ldots, n\}$. We assume that this representation of P is irredudant, that is, the defining linear inequalities of P are in one-to-one correspondence with the facets of P. In this chapter, we follow the convention of MAPLE's `PolyhedralSets` library and refer to these inequalities as the `relations` of P.

Following the informal description of the algorithm above, for each fractional vertex, we need to find the closest integer points on the facets adjacent to this vertex. But we first notice that it is possible that the supporting hyperplane of a facet, and therefore the facet itself, does not have any integer points. Therefore, the first step of our algorithm is to normalize the `relations` of the input using the "sliding process" described in Section 4.1.1.

In the next step, using the procedure `closestIntegerPoints` (Algorithm 1), we find the closest integer point to each fractional vertex on its adjacent facets. From the proof of Lemma 1 in [49] we know that, on a line $a_1x + a_2y = b$, a point (x, y) has integer coordinates if and only if its x is integer and satisfies $x \equiv \frac{b}{a_1} \mod a_2$. We can use this observation to find the closest integer point to a given point on a given line. We also deal with the case where a facet does not

contain any integer points.

Algorithm 1: Compute the closest integer points to each fractional vertex on its adjacent facets

```
1  Function closestIntegerPoints(V)
       Input: V, a list of the vertices of P
       Output: V_C, a list of pairs where V_C[i][1] and V_C[i][2] store the closest integer
               points of vertex V[i] on its two adjacent facets.
2      n ← |V|
3      for i = 1, ..., n do
4          Let V[i_1] and V[i_2] be the vertices adjacent to V[i]
5          for j = 1, 2 do
6              if there are integer points between [V[i], V[i_j]] then
7                  V_C[i][j] ← closest integer point to V[i] on [V[i], V[i_j]]
8              else
9                  V_C[i][j] ← NULL
10     return V_C
```

Next, we need to (1) construct the polyhedral sets the corner regions described informally above, and compute their integer hulls. Then, we find the convex hull of the union of all these integer hulls (see Algorithm 2). Lemma 4 in [49] shows that the vertices of this final convex hull are the vertices of P_I.

For a fractional (i.e. non-integer) vertex $V[i]$, if neither $V_C[i][1]$ nor $V_C[i][2]$ is NULL, then the corner is a triangle with vertices $[V[i], V_C[i][1], V_C[i][2]]$. If one or both of $V_C[i][1]$ and $V_C[i][2]$ are NULL, which means there is no integer point on one or both adjacent facets of $V[i]$, then we construct the corner region as follow.

1. let S_P be a stack initialized to the empty stack and let V_P be a list initialized to the empty list
2. Push $V[i]$ to S_P and add $V[i]$ to V_P
3. while S_P is not empty
   (a) pop out a vertex v from V_P,
   (b) if an adjacent facet f to v does not contain integer point then add to V_P the vertices of f and push to S_P the vertices of f.
4. for every vertex v in V_P add to V_P the "closest integer points" of v, if any.

At the end of the execution of the above procedure, the list V_P contains at least one non-integer vertex and two integer points. The convex hull of the points in V_P is the *corner region* associated with the input vertex $V[i]$.

To compute the integer hull of a corner, we use a brute-force method that searches for all the integer points within the corner polyhedral set and then compute the convex hull of all these points. In the article [19] Cook, Hartmann, Kannan and Mc Diarmid have shown that the size and shape of the integer hull (given by its vertices) of a polyhedron depends on the coefficients, a_{ij}, of the `relations` of the input but not on the constant terms b_i. This suggests that the area of the corner regions where we need to do exhaustive search on is not related to the area of the input polyhedral set P. Since the computation of the integer hull of the central region is free of

cost, this suggests that the time complexity of our algorithm is not related to the volume of the input polyhedral set.

Algorithm 2: Construct and compute the integer hulls of the corner polyhedral sets

```
1  Function cornerIntegerHulls(V)
       Input:
           • V, the list of the vertices of the input polyhedral set
           • V_C, the output from Algorithm 1
       Output: A set of all the vertices of the integer hulls of the corners
2      V_I ← {}
3      n ← |V|
4      for i = 1, ..., n do
5          if V[i] is an integer point then
6              V_I ← V_I ∪ {V[i]}
7          else
8              T ← ConstructCorner(V[i], V_C)
                   /* create a corner polyhedral set as we described above
                   */
9              A ← AllIntegerPoints(T)  /* find all the integer points in
                T  */
10             V_tmp ← ConvexHull(A)
                       /* compute the vertices of the convex hull of A */
11             V_I ← V_I ∪ {V_tmp}
12     return V_I
```

With all the sub-routines introduced above, we present our integer hull algorithm (Algorithm 3) for 2D polyhedral sets. We discuss some of the implementation details in Section 4.4.

Algorithm 3: Compute the integer hull of a given 2D polyhedral set

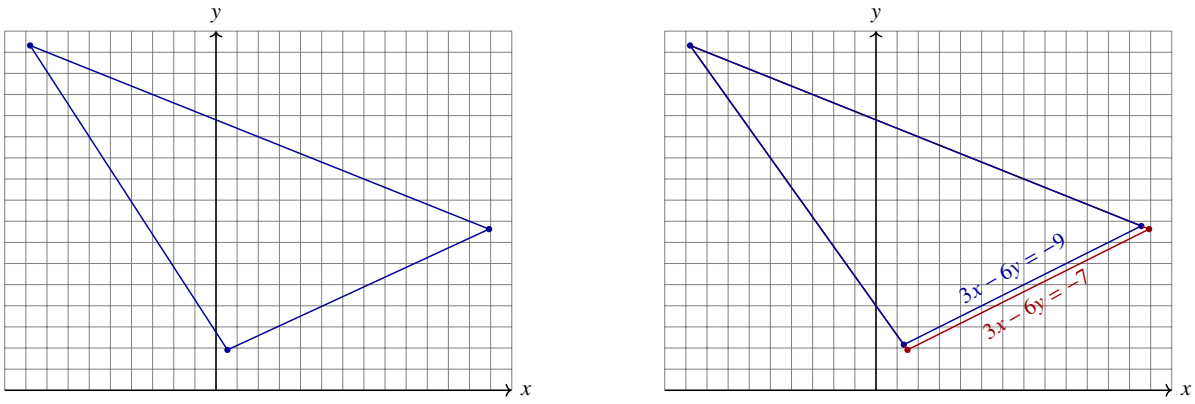
```
1  Function IntegerHull2D(P)
       Input: P, a 2D PolyhedralSet object
       Output: I, a list of the vertices of the integer hull of P
2      Process corner cases
3      Q ← Normalization(P)
4      V ← Vertices(Q)
5      V_C ← closestIntegerPoints(V)
6      V_I ← cornerIntegerHulls(V, V_C)
7      I ← ConvexHull(V_I)
8      return I
```

4.2.2 An example

In this section, we use the following example to show how our 2D algorithm works. The input is a polyhedral set defined by

$$\begin{cases} 2x + 5y & \leq & 64 \\ -7x - 5y & \leq & -20 \\ 3x - 6y & \leq & -7 \end{cases}$$

The first step we need to do is to normalize the facets. In this example, there is only one facet which is given by the `relation` $3x-6y \leq -7$. We replace it with $3x-6y \leq -9$ (See Figure 4.1).

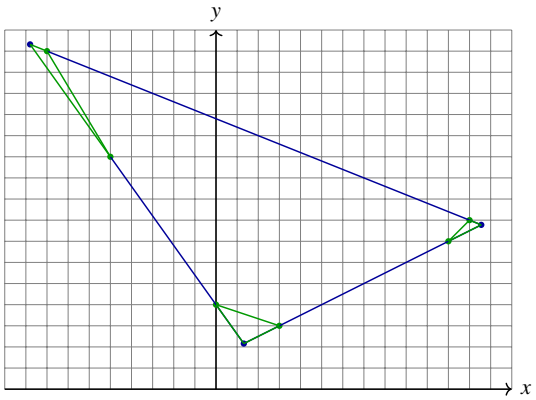


(a) Input is a polyhedral set

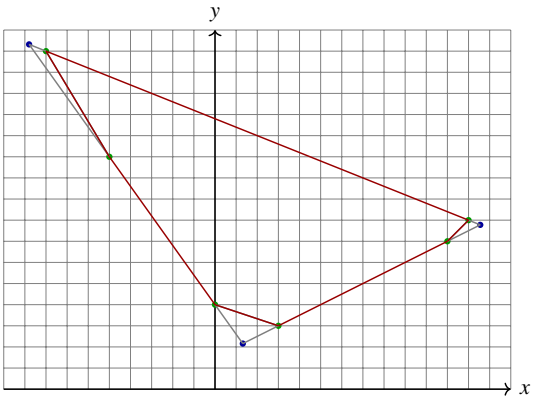
(b) Normalize the input

Figure 4.1: Input and replaceNonIntegerFacets

Next we need to find the closest integer points to each fractional vertex on its adjacent facets. In our case, all three vertices are fractional, so we need to find two integer points for each (See Figure 4.2a). And as we discussed in Section 4.2, the center part of the input is already an integer hull, so no action needed for this area. As we can see in Figure 4.2b, the center part takes most of the volume of the input, by doing so we cut down the size of the problem.



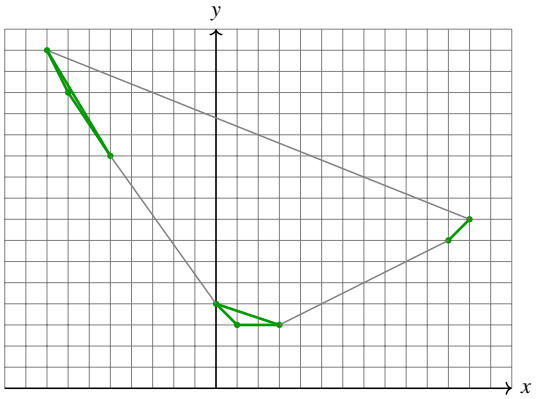
(a) For a fractional vertex, find the integer point on each adjacent facet that is closest to it and construct a triangle with the three points



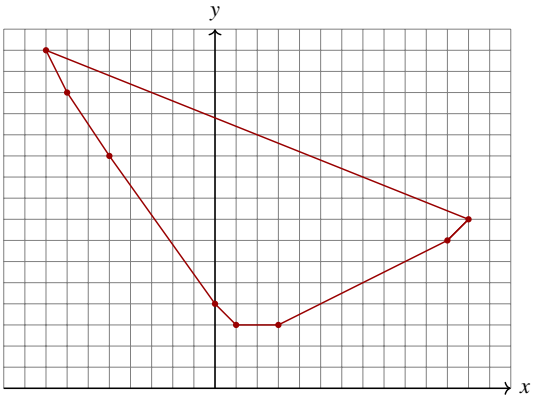
(b) The center part is already an integer hull so we don't need to do anything

Figure 4.2: Partition the input

Then we just need to compute the integer hulls of the small corner triangles and use the results to compute the final output (See Figure 4.3).



(a) Apply the previous two steps to each fractional vertex



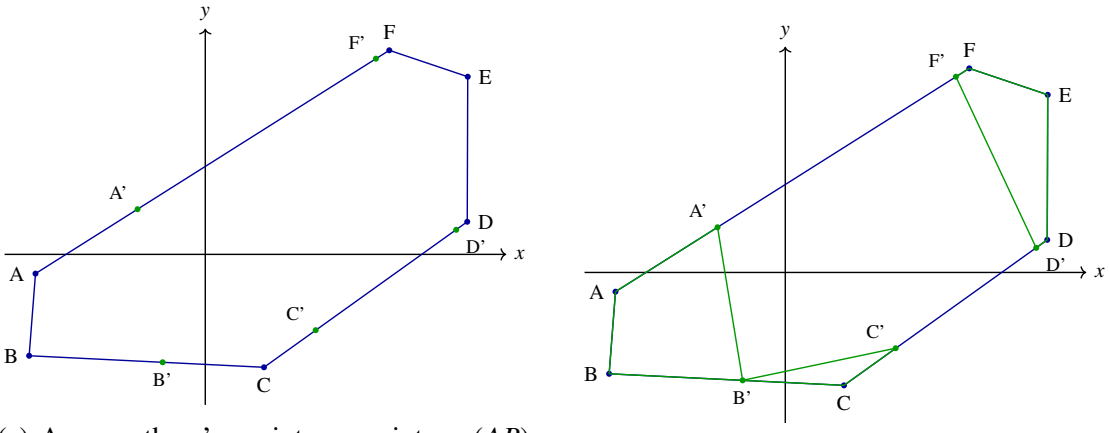
(b) Find the convex hull of all the result vertices from the previous steps

Figure 4.3: Compute the integer hulls of the parts and the final result

This is a general example such that each facet has integer points after normalization. As we mentioned in the previous section, there are corner cases where one or more facets don't have integer points although the supporting plane of them contain integer points. Here we use Figure 4.4 to indicate how we solve this problem and Figure 4.5 to show that it won't affect the efficiency of the general algorithm.

Assume $(ABCDEF)$ is the polyhedral set after Normalization and there's no integer point on facets (AB), (EF) and (ED). For facet (AB) instead of constructing a corner for each vertex A and B we construct one "corner" $(ABB'A')$ with both A and B as well as the closest integer point B' on (BC) and A' on (AF). In the scenario where several adjacent facets don't have integer points like (EF) and (DE), again we collect all the related vertices and their existing

closest integer points to form a "corner", such as $(DEFF'D')$. Therefore, the Figure 4.4b shows all the corner polyhedral sets that we need to compute the integer hull explicitly of Figure 4.4a.

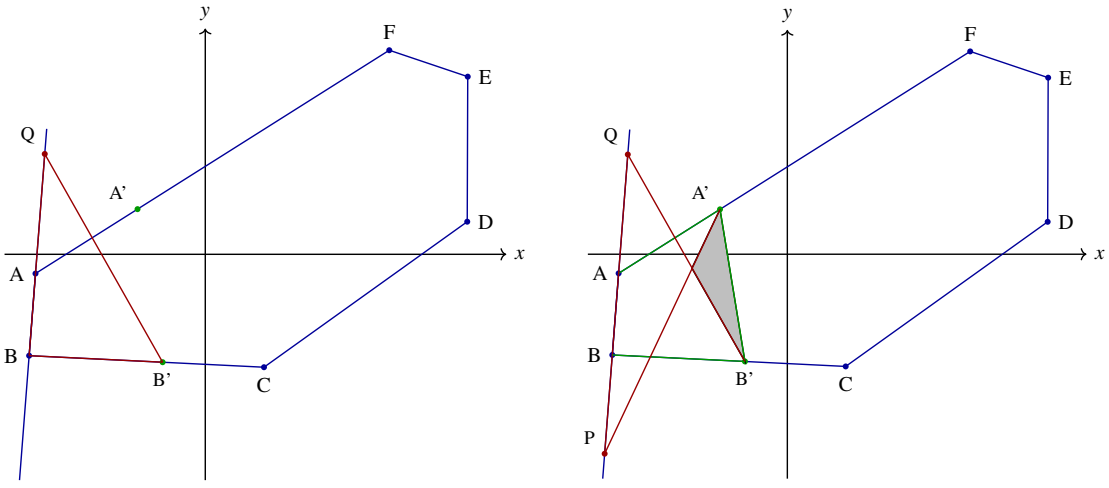


(a) Assume there's no integer point on (AB), (EF) and (ED)

(b) Corner polyhedral sets of this input

Figure 4.4: No integer point on some facets

Let's look at facet (AB) for the efficiency consideration. In Figure 4.5a, let Q be the closest integer points on the supporting plane of (AB) to B. From Section 3.1 we know the maximum length of BQ and BB', therefore, we know the maximum size of the corner polyhedral set of vertex B. Thus, we can see from Figure 4.5b, for facets (AB) we compute the integer hull of polyhedral set $ABB'A'$ instead of both $\triangle BB'Q$ and $\triangle AA'P$. In the case where some facets don't contain integer points, while we do need to consider some extra area (like the shaded area in Figure 4.5b), the actual area for integer hull computing is comparable with the general case.



(a)

(b)

Figure 4.5

4.3 Integer hull of a 3D polyhedral set

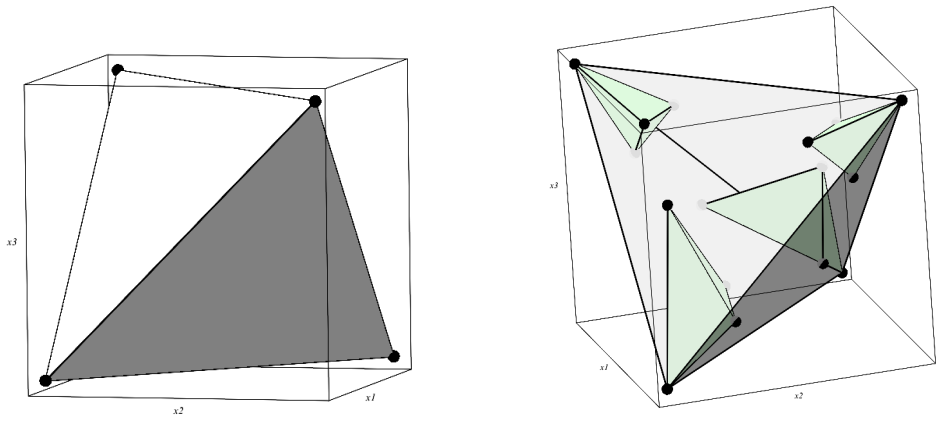
With the 2D algorithm in place, we can move on to a higher dimension. In this section, we present our integer hull algorithm for 3D polyhedral sets. The general idea behind the algorithm is the same as that of the 2D algorithm. We want to partition the input into smaller polyhedral sets and separate the parts into two categories: the ones with fractional vertices for which we need to compute the integer hulls as sub-problems and the other ones that are already integer hulls themselves. After processing all the sub-problems, we combine the results of all the parts together and compute the final result.

4.3.1 Algorithm

The first step of the 3D algorithm is the same as that in Section 4.2.1, that is, we normalize the input polyhedron as described in Section 4.1.1. Next, we find the "closest integer points" to the fractional (i.e. non-integer) vertices on their adjacent facets. This second step is done as described in Section 4.1.2. We note three important facts about the "closest integer point", w, w.r.t a given vertex v on a given facet f adjacent to v:

1. the integer point w is v itself, if v is an integer point,
2. if v is not an integer point and f contains no integer points then w is undefined and set to `NULL` in our pseudo-code below,
3. if v is not an integer point and f contains integer points, then the integer point w may not be the actual closest integer point to v on f, but it is always an integer point which is cheap to compute and expectedly close to v.

As a result, every vertex and its "closest integer points" form a "small" polyhedral set. For example, Figure 4.6a displays an input polyhedron P, actually a tetrahedron, the vertices of which are all assumed to be non-integer. Figure 4.6b displays in green the corner region of each vertex of P.



(a) Input is a polyhedral set

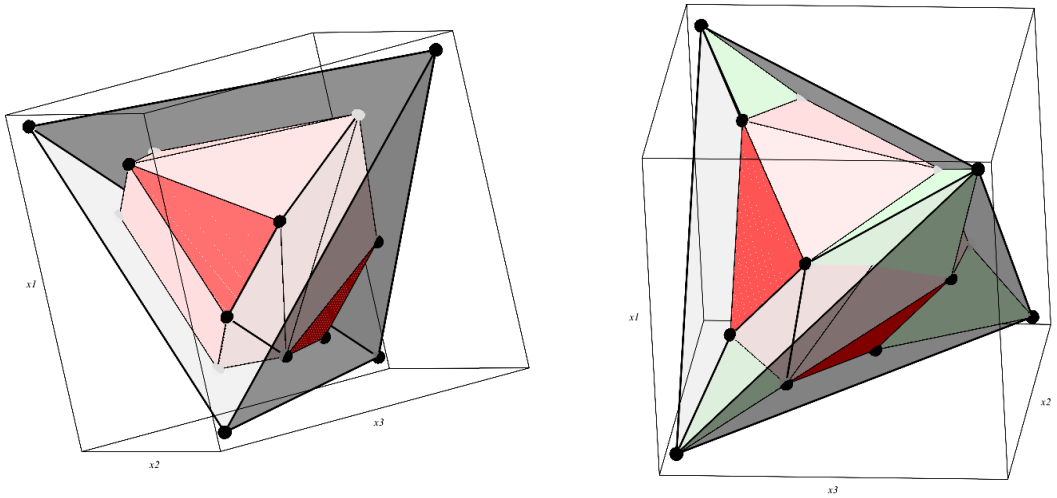
(b) Every fractional vertex and its closest integer points form a small polyhedral set

Figure 4.6: Input and fractional vertices

Figure 4.7a shows what we call the *center region* of the input polyhedron P, that is, the

convex hull of all closest integer points of all vertices of P. On our figures, we shall always color center regions in red. Since every closest integer point of every vertex is an integer point, it follows easily that center region is a polyhedron which is its own integer hull.

In the 2D case, the corner regions and the center region form a partition of the input polyhedron. But in the 3D case, there are regions that are not covered by these green and red regions, to be more precise, those missing regions are near the edges (See Figure 4.7b). It should also be noted that, in the 3D case, there could be one (or more) green regions with a non-empty intersection with the red region.



(a) The center part is already an integer hull so we don't need to do anything

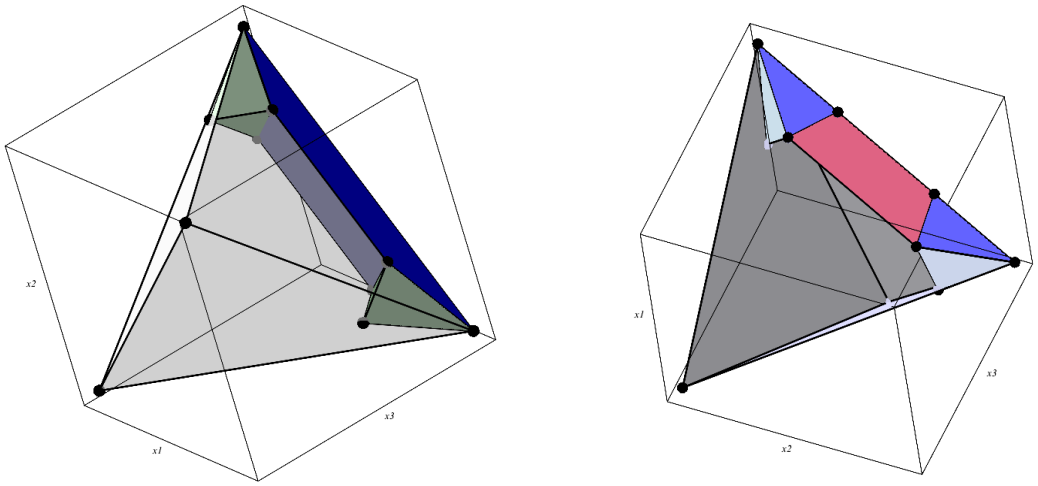
(b) There are areas that are not covered by any part

Figure 4.7: The center part and the corners

In order to cover the entire input polyhedron P, we need another set of sub-polyhedral sets of P. As shown on Figure 4.8a, for an edge e that has at least one non-integer vertex, the two vertices of that edge and their "closest integer points" on the facets adjacent to that edge form a polyhedral set that we call the *edge region* of e. We note that an edge region is either a tetrahedron or a 2D triangle. On our figures, we shall always color edge regions in blue. For convenience, we define the edge region of an edge e whose vertices are both integer points to be e itself. If we construct the edge region of each edge of P, we can cover all the missing regions, as illustrated on Figure 4.7b.

We are now ready to compute the integer hull P_I of P. Of course, we want to deduce P_I from the convex hulls of the green, red and blue regions. Recall that the red region is its own integer hull. For every green or blue region R which is not its own integer hull, we use a brute-force method to compute its integer hull, that is, we use an exhaustive search to find all the integer points within R and compute the convex hull of those points.

To cut down the cost of the exhaustive search in such a region R, we use some optimizations. For instance, if R is the edge region (blue region) of an edge e containing integer points, we use those integer points to further partition R: we we find the closest integer point of e to each non-integer vertex of e so as to split R into three parts, see Figure 4.8b for an example.



(a) The area around an edge (b) When there are integer points on the edge

Figure 4.8: Polyhedral sets that cover the edge areas

Finding, on a given segment S, the integer point closest to a given vertex of S is relatively simple in the 2D problem, but in the 3D case, we need to address the following, more complicated, question: finding, on a given bounded 3D polyhedron F, an integer point closest to a given vertex of F. A natural step towards answering this question is to represent all the integer points of F, which, itself, is an integer hull problem. Since the 3D polyhedron F is "flat", we can project it to a 2D ambient space and use our algorithm from Section 4.2.

Here we use the procedure `HNFProjection(F, d)` which is introduced in detail in Section 4.1.2. Recall that this procedure will return an ordered pair (G, R_F) where R_F mapping "flat" 3D objects to their 2D counterparts.

Having a 2D polyhedral set F_P, we use our Algorithm 3 to compute the vertices of the integer hull of F_P. Although the HNF method keeps the integer points in the projection, it can not keep the distance among the points in general, so we must find the original image of the vertices of the integer hull of F_P.

Now that we have the integer hull of a facet, we can search for the closest integer points to each of its vertices. Here we decide to use the closest vertex of the integer hull instead of the actual closest integer point. Using the closest integer vertex might slow down the later steps but only by a very small amount. Searching for the actual point would be another optimization problem and this would be less efficient looking at the whole picture.

Algorithm 4: Compute the closest integer points on a facet F to the vertices on it in a 3D polyhedral set

```
1  Function closestIntegerPoints3D(F, V)
       Input:
           • F, a facet of P in the form of a PolyhedralSet object
           • V, a list of the vertices of P
       Output: V_C, a list where V_C[i] is the integer point on F which is the "closest" to
                V[i], if V[i] is in F, and [] otherwise
2      F_P, R_F ← HNFProjection(F, 3)
                /* Make a projection F_P of the facet F onto 2D space using
       Hermite Normal Form */
3      V_tmp ← IntegerHull2D(F_P)
                        /* Find the vertices of the integer hull of F_P */
4      V_F ← R_F(V_tmp)
                /* V_F is the set of vertices of the integer hull of F, see
       Theorem 4 for more details.  */
5      n ← |V|
6      if V_F = ∅ then
7          return []
8      for i from 1 to n do
9          if V[i] in F then
10             V_C[i] ← closest point to V[i] in V_F
11         else
12             V_C[i] ← NULL
13     return V_C
```

As mentioned above, in order to form a complete "partition" (actually coverage) of the input polyhedral set, we need to carefully consider every edge that has at least one fractional vertex. To this end, we use Algorithm 1 to find the closest integer points to a fractional vertex on its adjacent edges. Now that we have all the "closest integer points" we need, we can construct the parts that are the "blue", "red" and "green" regions in Figure 4.6, 4.7 and 4.8. Algorithm 5 shows the procedure for the partition. Since all the vertices of the "red" polyhedral set are

integer points, work remains to be done only in the "green" and "blue" polyhedral sets.

Algorithm 5: Partition of a 3D polyhedral set

```
1  Function Partition3D()
       Input: V, A list of all the vertices of P
       E, a list of all the edges of P
       F, a list of all the facets of P
       V_C, a list of lists where V_C[i][j] is the "closest" integer point to V[j] on F[i]
       V_E, , a list of lists where V_E[i][j] is the "closest" integer point to V[j] on E[i]
       Output: V_set, a set where each element contains the vertices of one part of the
               partation
2      V_set ← ∅
3      for each V_C[i] in V_C do
4          if V_C[i] = [] then
5              V_tmp ← {Vertices(F[i])}
                                   /* F[i] does not contain integer point */
6              V_tmp ← V_tmp ∪ {V_C[j][k] | V_C[j][k] ≠ NULL, V[k] ∈ F[i]}
7              V_set ← V_set ∪ {V_tmp}
8      for each V[j] in V do
9          V_tmp ← {V[j]}
10         V_tmp ← V_tmp ∪ {V_C[i][j] | V[j] ∈ F[i]}
11         V_set ← V_set ∪ {V_tmp}
12     for each E[i] in E do
13         if V_E[i] = [] then
14             V_tmp ← {Vertices(E[i])}
                                   /* E[i] does not contain integer point */
15             V_tmp ← V_tmp ∪ {V_C[j][k] | E[i] ∈ F[j], V[k] ∈ E[i]}
16             V_set ← V_set ∪ {V_tmp}
17         else
18             for each vertex v of E[i] do
19                 V_tmp ← {v}
20                 V_tmp ← V_tmp ∪ {V_C[j][k] | E[i] ∈ F[j], V[k] = v}
21                 V_tmp ← V_tmp ∪ {V_E[i][k] | V[k] = v}
22                 V_set ← V_set ∪ {V_tmp}
23     return V_set
```

Before we present the complete algorithm, there are some corner cases that need to be considered. Similar to our 2D problem, the input polyhedral set could be not fully dimensional. Again we use Hermite Normal Form (HNF) to project the input to 2D space, and deal with it as a 2D problem. Another corner case would be after applying `Normalization`: no facets have integer points, in this case we use the brute-force approach for the whole input.

With all the sub-routines in order, here is our algorithm, Algorithm 6, for computing the

integer hull of a bounded 3D polyhedral set.

Algorithm 6: Compute the integer hull of a given 3D polyhedral set

Function `IntegerHull3D`(P)
- **Input:** P, a 3D PolyhedralSet object
- **Output:** I, a list of the vertices of the integer hull of P
- Process corner case: P is not fully dimensional
- $P \leftarrow$ `Normalization`(P)
- $V \leftarrow$ `Vertices`(P)
- $F \leftarrow$ `Facets`(P)
- **for** *each* $F[i]$ *in* F **do**
  - $V_C[i] \leftarrow$ `closestIntegerPoints3D(F[i], V)`
  - `/* V_C is a 2D list where V_C[i][j] contains the closest integer point to V[j] on F[i] */`
- $E \leftarrow$ `Edges`(P)
- **for** *each* $E[i]$ *in* E **do**
  - $V_E \leftarrow$ `closestIntegerPointsOnEdge`($E[i], V$)
  - `/* V_E is a 2D list where V_E[i][j] contains the closest integer point to V[j] on E[i] */`
- $V_{\text{set}} \leftarrow$ `Partition3D`(V, E, F, V_C, V_E)
  `/* V_set = {V_1, ..., V_n} where V_i contains the vertices of one part */`
- $I \leftarrow$ {`ElementsOf(V_C)`, `ElementsOf(V_E)`}
- **for** *each* V_i *in* V_{set} **do**
  - $P_{\text{list}} \leftarrow$ `PolyhedralSet`(V_i)
  - $A_I \leftarrow$ `AllIntegerPoints`(P_{list})
  - $I \leftarrow I \bigcup$ `ConvexHull`(A_I)
- **return** `ConvexHull`(I)

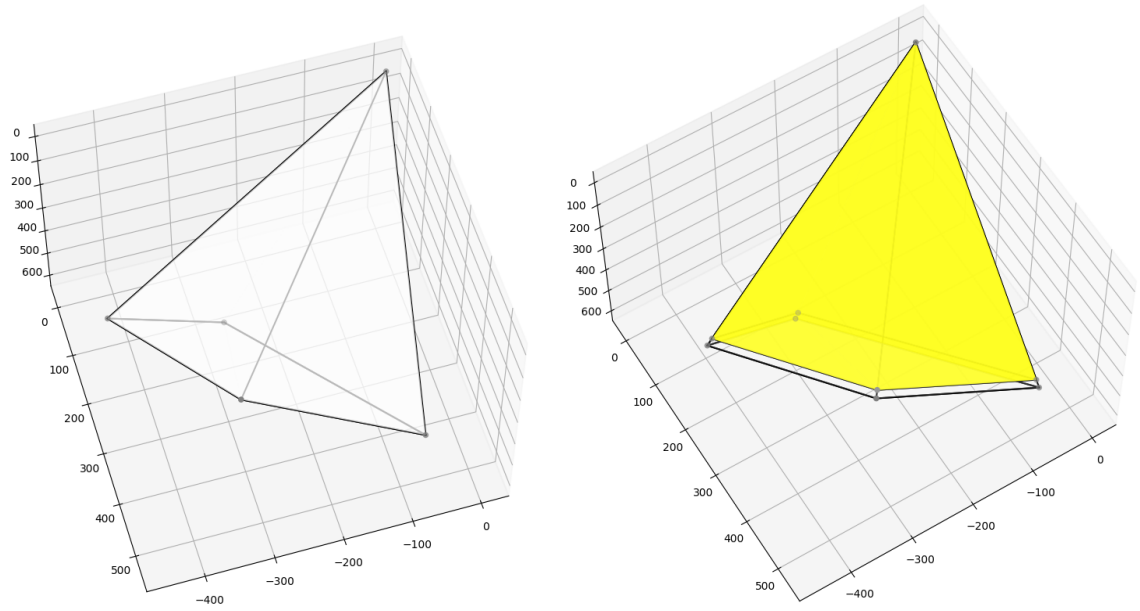
4.3.2 An example

In this section we use a concrete 3D example to demonstrate how Algorithm 5 and Algorithm 6 work. Consider the polyhedral set P given by the Equations 4.5 and Figure 4.9a. In the `Normalization` step, only one facet is updated as is shown in Figure 4.9b and the normalized input is given by Equations 4.6. This new P has vertices:

$$V = [[7, -\frac{22}{3}, -4], [-\frac{855412}{3319}, \frac{2489899}{19914}, \frac{7854701}{13276}], [-\frac{23351687}{53104}, \frac{11786923147}{52732272}, \frac{11209969}{53104}],$$
$$[-\frac{4021497}{53104}, \frac{27434329}{53104}, \frac{18239129}{53104}], [-\frac{4300293}{13276}, \frac{6638921}{13276}, \frac{719333}{39828}]]$$

$$\begin{cases} -98877x_1 - 189663x_2 - 1798x_3 \leq 705915 \\ -10109x_1 - 5958x_2 - 14601x_3 \leq 31333 \\ -5405x_1 + 4965x_2 + 3870x_3 \leq 4303504 \\ 729x_1 - 117x_2 + 350x_3 \leq 4561 \\ 677x_1 + 465x_2 - 540x_3 \leq 3489 \end{cases} \tag{4.5}$$

$$\begin{cases} -98877x_1 - 189663x_2 - 1798x_3 & \leq & 705915 \\ -10109x_1 - 5958x_2 - 14601x_3 & \leq & 31333 \\ -1081x_1 + 993x_2 + 774x_3 & \leq & 860700 \\ 729x_1 - 117x_2 + 350x_3 & \leq & 4561 \\ 677x_1 + 465x_2 - 540x_3 & \leq & 3489 \end{cases} \tag{4.6}$$



(a) Input polyhedral set P (b) After Normalization

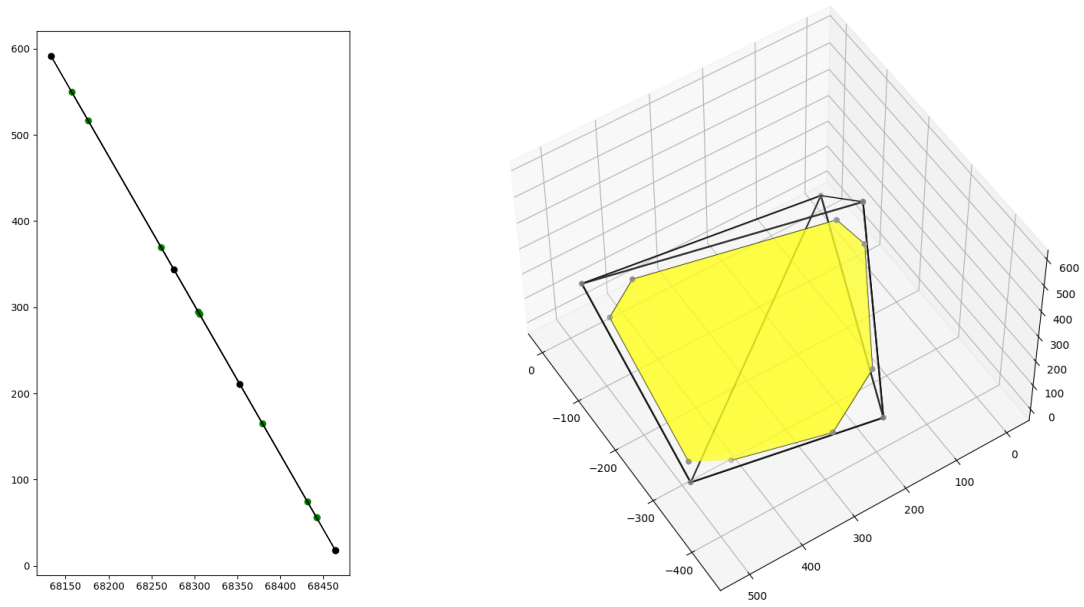
Figure 4.9: Normalized input

Next, we compute lists V_C and V_E. We use the facet F that defined by the vertices $V[1]$, $V[2]$, $V[3]$, $V[4]$ as an example to show this process. We first call procedure `HNFProjection` on F. It returns a projection F_P of F and a map R_F as follow:

$$F_P = \texttt{PolyhedralSet}([[\frac{901093936819}{13183068}, \frac{11209969}{53104}], [\frac{681694796}{9957}, \frac{719333}{39828}], [\frac{2713599509}{39828}, \frac{7854701}{13276}], [\frac{1812873801}{26552}, \frac{18239129}{53104}]], [x'_1, x'_2])$$

$$R_F : \begin{cases} x_1 &= 993x'_1 + 573x'_2 - 67995300 \\ x_2 &= 1081x'_1 + 623x'_2 - 74020200 \\ x_3 &= x'_2 \end{cases}$$

We call Algorithm 3 to compute the integer hull of F_P, then we use R_F to find the integer hull of F. Then for each vertex v of F, we will pick the closest vertices of the integer hull to v as the "closest" integer point. Figure 4.10a shows the projection F_P and its integer hull (although it may look like a line, all the points are not on a line) and Figure 4.10b shows the integer hull of the facet F.



(a) The projection F_P and its integer hull

(b) Integer hull of facet F

Figure 4.10: "Closest" integer points on facets

Now we have the full list of V_C and V_E we can proceed to Algorithm 5 to find the "corners".

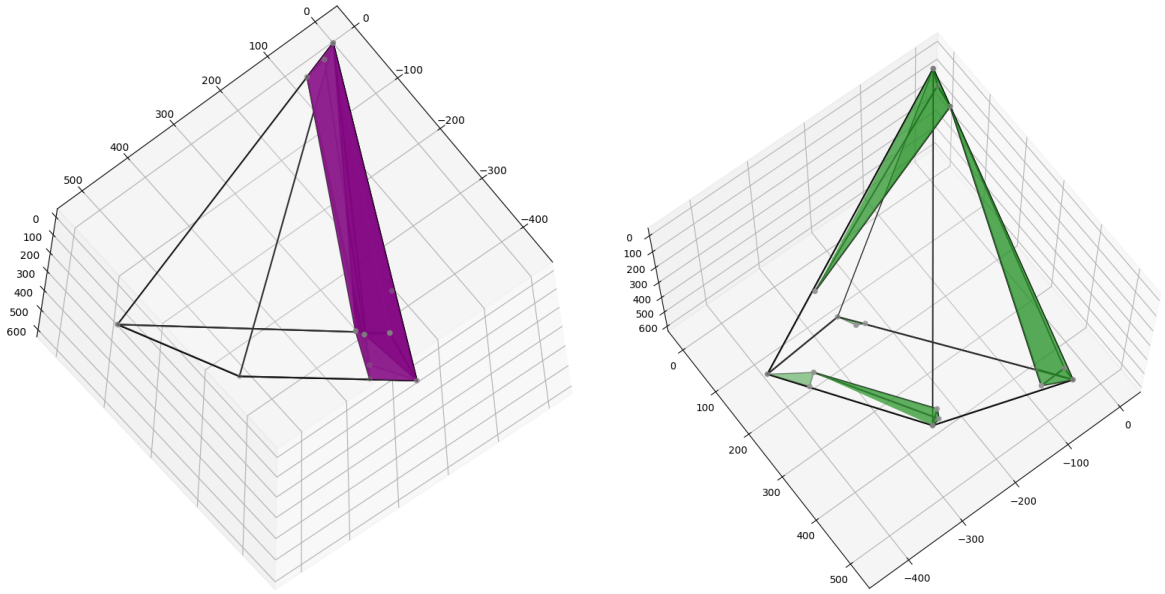
$$\begin{aligned} V_C = &[[], [[-3, 20, 7], [], [], [-78, 495, 322], [-303, 480, 27]], \\ &[[-317, 165, 150], [], [-392, 280, 155], [], [-392, 280, 155]], \\ &[[-3, 56, 38], [-235, 172, 560], [], [-103, 456, 380], []], \\ &[[], [-249, 167, 550], [-408, 294, 165], [-126, 502, 292], [-306, 490, 56]]] \end{aligned}$$

$$V_E = [[], [], [], [[-3, 56, 38], [], [], [-63, 436, 290], []], [], [], [], []]$$

For each facet that does not contain integer point we process it with line 3-7 of Algorithm 5, and create one corner polyhedral set for each of such facet. In our example only one such facet exists, and we create the purple polyhedral set in Figure 4.11a with its vertices $V[0]$, $V[1]$, $V[2]$ and all the $V_C[i][j]$ that exists for $j = 0, 1, 2$. The list of the points to form the purple polyhedral set is as follow:

$$\begin{aligned} &[[7, -22/3, -4], [-855412/3319, 2489899/19914, 7854701/13276], \\ &[-23351687/53104, 11786923147/52732272, 11209969/53104], [-3, 20, 7], \\ &[-317, 165, 150], [-392, 280, 155], [-3, 56, 38], [-235, 172, 560], \\ &[-249, 167, 550], [-408, 294, 165]] \end{aligned}$$

Now we move on to line 8-11 of Algorithm 5. For each vertex, we form a polyhedral set of the vertex and all the "closest" integer points on its adjacent facets. These "corners" are shown in green in Figure 4.11b.



(a) We make one "corner" for each facet that does not contain any integer point

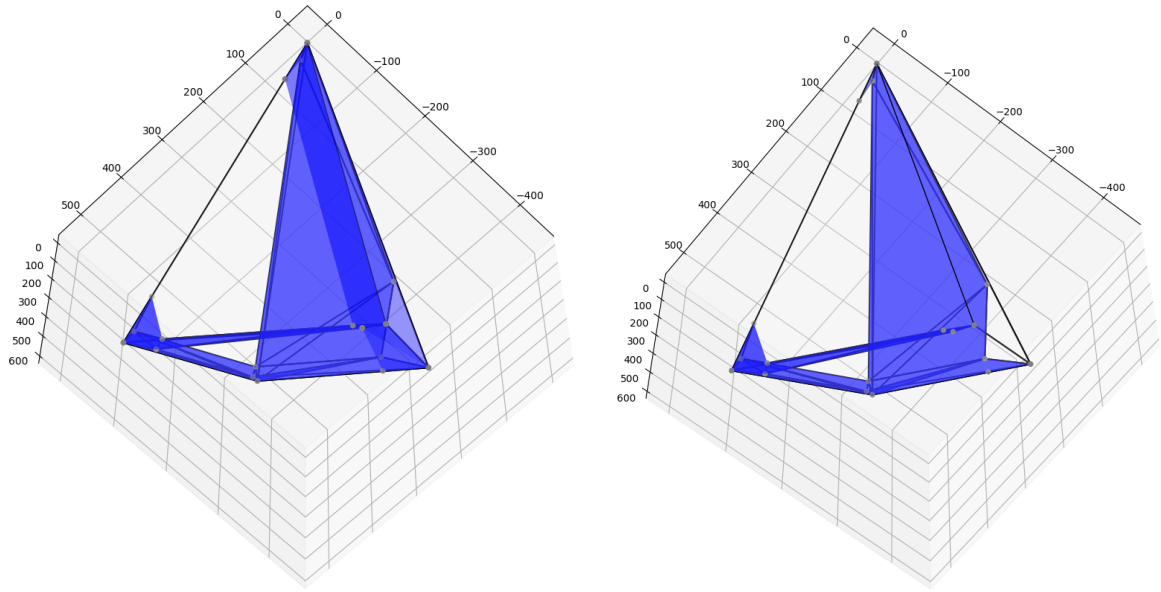
(b) Every fractional vertex and its closest integer points form a green "corner" polyhedral set

Figure 4.11: Empty facets and vertices

In the last step of Algorithm 5 (line 12-22) we look at each edge. For each edge that does not contain integer point, we make one "corner". And for the edges that have integer points, we make two "corners" for each such edge, see the blue parts in Figure 4.12a. In our example with 8 edges, there is one that contains integer points so in total we make 9 blue polyhedral sets. But in reality, some of them overlap with the purple one in Figure 4.11a, we discard them in our implementation. The blue "corners" that we actually need to process are as shown in Figure 4.12b. A similar discarding process can be applied to the green corners as well.

Other than the purple, green and blue "corners" we formed, the majority of the volume of the input fall into the parts that are already integer hulls. They are the red ones in Figure 4.13a all their vertices comes from V_C and V_E therefore guaranteed integer points.

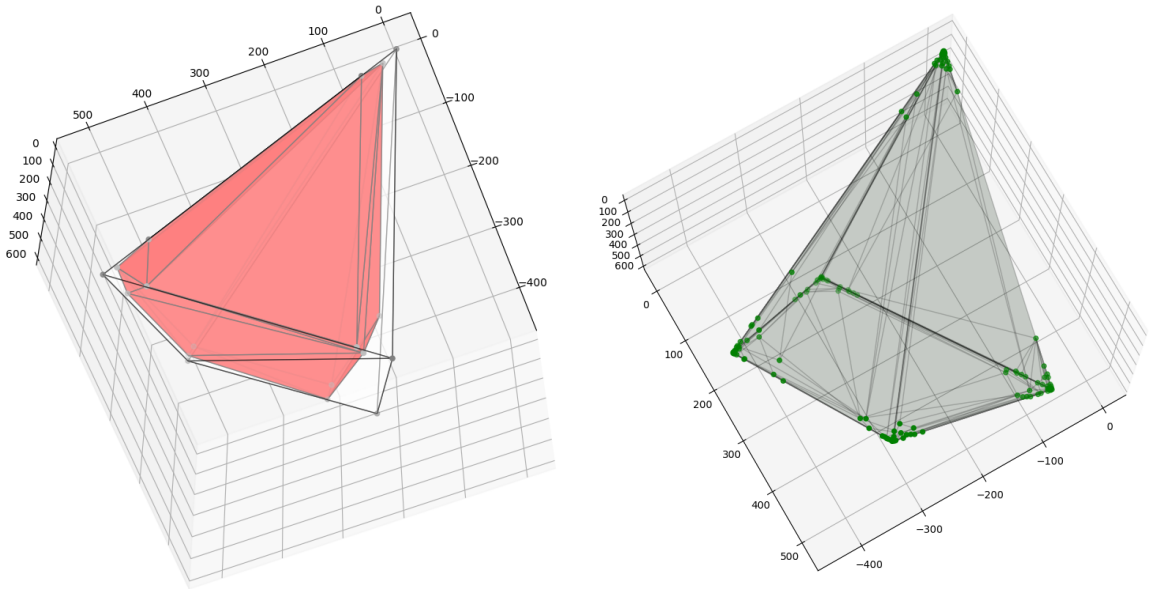
In Figure 4.13b we show the integer hull of this example, it contains 139 vertices and as we can see most of them are clustered around the vertices of the input. Recall the results in [19], [65] and [13] the number of vertices of the integer hull is related to the "size" of the inequalities that define the input. So even for input with small number of vertices (for example, 5 vertices in our example) the integer hull can has many more vertices.



(a) We make one or two blue "corners" for each edge

(b) Some blue "corners" can be discarded

Figure 4.12: How we deal with edges



(a) The parts that have only integer vertices

(b) Integer hull of the input

Figure 4.13

4.4 Implementation and experimentation

We have implemented the bounded 2D and 3D algorithms in both MAPLE and the C/C++ programming language. The MAPLE version is available in 2022 release of MAPLE as the `IntegerHull` command of the `PolyhedralSets` library. In addition, an implementation for arbitrary dimensional inputs is done in MAPLE (not in the released version) as well. In this section, we discuss implementation details and the experimentation with our implementations. All the benchmarks are done on an Intel i5-8300H CPU at 2.30GHz with 16GB of memory. As we discussed in Chapter 1, there are studies (such as [33] and [13]) developing approaches to enumerate the vertices of P_I using their relations with the vertices of P but to our knowledge no implementation of such methods exist. So in the following sections we compare our implementation with the existing implementation of the naive method (enumeration of all integer points, followed by the computation of their convex hull) for verification and proof of concept.

4.4.1 The MAPLE implementation

For the MAPLE version, we use the functions provided by the `PolyhedralSets` library for polyhedral set manipulation such as construction, getting the vertices and faces. To obtain the adjacency information among the faces we need to compute the face lattice of the input polyhedral set; the `PolyhedralSets` library provides the command `Graph` for that task. We compare our MAPLE implementation with another MAPLE package. In the 2019 Maple Conference, Jing and Moreno-Maza introduced the `ZPolyhedralSets` package, presented in [36]. A `ZPolyhedralSet` is the intersection of a polyhedral set and a lattice. The integer hull of a polyhedral set is equal to a `ZPolyhedralSet` when the `ZPolyhedralSet` is defined using the standard integer lattice (which represents all the points with integer coordinates).

The `ZPolyhedralSet` package provides a command, `EnumerateIntegerPoints`, which would find and output all the integer points within a `ZPolyhedralSet` object. Given a polyhedral set, to obtain the same result that our algorithm computes, which is the list of the vertices of the integer hull, we use the command `EnumerateIntegerPoints` to find all the integer points within the input, then we the command use `ConvexHull` from `ComputationalGeometry` to compute the vertices.

Table 4.1 shows the time spent in our algorithm (`IntegerHull`) and the above two-step method (EIP+CH) to obtain the same result. The inputs are triangles with different volumes. As we discussed in Chapter 1, the cost for finding all the integer points is related to the volume of the input and we can see the trend in the "EIP+CH" columns. Time spent by our algorithm does not seem to depend on the volume of the input.

From Algorithm 3, we can see that the complexity of our algorithm depends on the number of facets and the number of fractional vertices in the input. Table 4.2 shows the running time of both algorithms (IntegerHull and EIP+CH) when the inputs are hexagons. The running time for `IntegerHull` is roughly double the time for triangle inputs.

Volume	27.95		111.79		11179.32	
Algorithm	IntegerHull	EIP+CH	IntegerHull	EIP+CH	IntegerHull	EIP+CH
Time(s)	0.172	0.410	0.244	0.890	0.159	58.083

Table 4.1: Integer hulls of triangles

Volume	58.21		5820.95		23283.82	
Algorithm	IntegerHull	EIP+CH	IntegerHull	EIP+CH	IntegerHull	EIP+CH
Time(s)	0.303	0.752	0.275	31.357	0.304	123.159

Table 4.2: Integer hulls of hexagons

Tables 4.3 and 4.4 show the running times of the same algorithms when the input is a tetrahedron and a bipyramid respectively. The result is similar to that of the 2D algorithm where the running time increases if there are more facets and vertices. One thing that we need to notice is that the running time of our algorithm grows as the volume increases, this is due to the way we deal with the parts that are around the edges. As we discussed in Section 4.3.1, if there is no integer point on an edge, the sub-polyhedral set would include the whole edge and its volume depends on the length of the edge. Recall that we use exhaustive search for the sub-polyhedral sets thus the running time depends on the volume of the input polyhedral set.

Volume	447.48		6991.89		55935.2	
Algorithm	IntegerHull	EIP+CH	IntegerHull	EIP+CH	IntegerHull	EIP+CH
Time(s)	1.202	6.892	1.498	67.814	1.517	453.577

Table 4.3: Integer hulls of tetrahedrons (4 facets, 4 vertices and 6 edges)

Volume	412.58		7050.81		60417.63	
Algorithm	IntegerHull	EIP+CH	IntegerHull	EIP+CH	IntegerHull	EIP+CH
Time(s)	1.476	5.711	1.573	60.233	1.728	512.101

Table 4.4: Integer hulls of triangular bipyramids (6 facets, 5 vertices and 9 edges)

4.4.2 The C/C++ implementation

For the C/C++ implementation, we follow the representations in the C library `cddlib` by Komei Fukuda[24] for the polyhedral set computations. GMP rational arithmetic is used until

the integer coordinates are obtained to ensure correctness. Our implementation can take polyhedral sets in either the V-representation or the H-representation as input; `cddlib` is used for representation conversion and some redundancy removal.

As we have discussed in Section 4.1.2 we use part of the algorithm in [35] to partition the polyhedral sets and we follow that same article for the enumeration of the integer points in the corners. We implemented Algorithm 2.4.10 in [18] and Algorithm 3 in [35] for the procedure `HNFProjection`. We also implemented the algorithm introduced by Kaibel and Pfetsch in [37] for the computations of the face lattice.

To verify our implementation, we compare our results with that of the Normaliz library [15]. We also implemented a naive procedure based on enumeration and convex hull computation to obtain the integer hull. Note that Algorithm 3 in [35] only enumerates the integer points inside the given polyhedral set while for Normaliz, if the input is not homogeneous Normaliz homogenizes it by raising the input to a higher dimension, therefore, Normaliz enumerates more points than we do for the same input.

Tables 4.5 and 4.6 show the time spent in these three different approaches for computing the integer hulls of the same inputs. Since the I/O formats are different for Normaliz and `cddlib`, we only measured the timings for the integer hull computation part but not the I/O parts of the programs. Especially, for Normaliz we only timed the function call "MyCone.compute(ConeProperty::IntegerHull)".

The examples are named as xdy_z, where x is the dimension of the input (all the examples are full dimensional). Each y represents a set of examples that are of the same shape which means these polyhedral sets $A\mathbf{x} \leq \mathbf{b}$ share the same coefficient matrix A while the vector $\mathbf{b}$ varies. xdy_0 is the smallest (volume wise) example in a set, for $z = 1, 2, 3$, vector $\mathbf{b}$ gets multiplied by $2, 5, 10$ respectively. For the 2D examples, $2d1$ has 6 vertices, $2d2$ has 4 vertices and $2d3$ has 3 vertices. And for the 3D examples, $3d1$ has 12 facets, 8 vertices and 18 edges, $3d2$ has 4 facets, 4 vertices and 6 edges and $3d3$ has 6 facets, 5 vertices and 9 edges.

The result is consistent with our observation in [49]. For the same family of input, the time spent by our algorithm is relatively stable while for both our naive implementation and Normaliz, the larger the volume of the input is, the more time they need to do the computation since often time larger polyhedral sets contain more integer points for enumeration.

4.5 Conclusion and future work

In this chapter, we introduced a new algorithm for computing the integer hull of a convex polyhedral set. This algorithm is essentially driven by the following simple idea: if P is close to be P_I (that is, up to a few vertices) then the computation of P_I from P should be cheap. We implement the algorithm for two-dimensional and three-dimensional input in both MAPLE and C/C++. The efficiency of this algorithm depends mainly on the shape of the input while the size of the input has little impact. We show in Section 4.4 that our algorithm can deal with inputs of very large volumes which are out of reach for algorithms that depend on enumeration.

The main steps of our algorithm are normalization, partition and merging. Our algorithm can be stated for polyhedral sets of arbitrary dimension and an MAPLE implementation is already in place. Our on-going development is an algebraic complexity analysis of our algorithm.

example	IntegerHull	Naive	Normaliz
2d1_0	0.451	0.565	2.837
2d1_1	0.478	0.657	1216.238
2d1_2	0.396	0.682	740.559
2d1_3	0.443	1.134	472.447
2d2_0	0.413	1.128	1258.422
2d2_1	0.411	2.714	1242.081
2d2_2	0.393	16.079	2622.995
2d2_3	0.449	47.145	10218.368
2d3_0	0.284	0.768	835.730
2d3_1	0.339	1.676	462.116
2d3_2	0.286	6.883	1559.401
2d3_3	0.324	25.637	5072.894

Table 4.5: Timing (ms) for computing integer hull of 2D examples.

We sketch below and in Algorithm 7 our algorithm for computing the integer hull of a D-dimensional convex polyhedral set P:

1. The input to the algorithm can be either the H-representation or the V-representation of P (or both). If only one is provided, the other representation will be calculated during the process, since both the vertices and the expressions of the facets are needed during the computation.
2. On lines 2 to 6, if the input is not full dimensional, we use `HNFProjection` to project to a lower dimension where it is full dimensional, compute the integer hull in that lower dimension and then lift the result back to the input dimension.
3. normalize the input using the procedure introduced in Section 4.1.1
4. for each vertex, find the "closest integer points" to it on each of its adjacent faces. We do that by computing the integer hull of the each face and then find the closest integer vertex of the integer hull to each of the faces' fractional vertex (Lines 10 - 14). On line 11 when we compute the integer hull of the projected facets, we can either recursively call our algorithm or we can use other approaches such as brute-force.
5. for each face of dimension from 0 to $D-2$, construct a "corner polyhedral set" using the integer points we obtained from Step 4.
6. compute the integer hull of each corner
7. compute the convex hull of all the integer hulls from Step 6
8. this convex hull is the integer hull of P

The procedures used in Algorithm 7 are explained as follow,

1. `HNFProjection(P)`: the procedure introduced in Section 4.1.2 that returns a projection G of P using Hermite normal form where G is full dimensional and the map $R_F(G) = F$.
2. `Normalization(P)`: the procedure introduced in Section 4.1.1 that returns the normalized P where the supporting plane of each facet contains integer points. Note that this procedure does change the integer hull of P.
3. `FaceLattice(P)`: returns the lattice of all the faces of P and their containment relationships.

example	IntegerHull	Naive	Normaliz
3d1_0	51.727	11.396	274.364
3d1_1	52.034	13.483	1018.449
3d1_2	60.821	21.106	2330.534
3d1_3	54.350	79.219	15346.996
3d2_0	4.488	0.826	851.495
3d2_1	4.615	0.923	956.666
3d2_2	4.624	1.527	793.192
3d2_3	5.522	4.394	1318.150
3d3_0	11.049	21.235	7862.109
3d3_1	16.001	145.068	N/A
3d3_2	23.822	2082.559	N/A
3d3_3	24.162	N/A	N/A

Table 4.6: Timing (ms) for computing integer hull of 3D examples.

4. `Faces(L,i)`: L is a face lattice and i is an integer number, this procedure returns the faces in L that have dimension i.
5. `CornerPolySet(P)`: this procedure is a generalization of Algorithm 5. For a face f, let F be a set of all the facets that intersect at f. If there exist integer points on f (which means the closest integer points on f to its vertices exist), for each vertex v of f, the "corner" polyhedral set has vertices consisting of: v, all the closest integer points on F to v, the closest integer point on f to v. If there is no integer point on f, the "corner" polyhedral set has vertices consisting of: all the vertices V of f, all the closest integer points on F to V.
6. `Enumeration(P)`: enumerate all the integer points in P.

Algorithm 7: Compute the integer hull of a polyhedralset

```
1  Function IntegerHull(P)
       Input: P, a PolyhedralSet
       Output: I, a list of the vertices of the integer hull of P
2      if P is not fully dimensional then
3          R_F, G ← HNFProjection(P)
               /* make projection G of P to a dimension where G is full
           dimensional */
4          V_G ← IntegerHull(G)
5          V_P ← R_F(V_G)
6          return V_P
7      P ← Normalization(P)
8      D ← Dimension(P)
9      L ← FaceLattice(P)
10     for each f in L do
11         V_f ← IntegerHull(f)
12         V ← Vertices(f)
13         for each v in V do
14             find the closest point to v in V_f
15     V_set ← {}
16     for i from 0 to D − 2 do
17         F ← Faces(L, i)
18         for each f in F do
19             V ← Vertices(f)
20             if there are integer points on f then
21                 for each v in V do
22                     C ← CornerPolySet(v)
23                     P_T ← Enumeration(C)
24                     V_T ← ConvexHull(P_T)
25                     V_set ← V_set ∪ V_T
26             else
27                 C ← CornerPolySet(f)
28                 P_T ← Enumeration(C)
29                 V_T ← ConvexHull(P_T)
30                 V_set ← V_set ∪ V_T
31     return ConvexHull(V_set)
```

Chapter 5

KLARAPTOR: A Tool for Dynamically Finding Optimal Kernel Launch Parameters Targeting CUDA Programs

In this chapter we present KLARAPTOR (Kernel LAunch parameters RAtional Program estimaTOR), a freely available tool built on top of the LLVM Pass Framework and NVIDIA CUPTI API to dynamically determine the optimal values of kernel launch parameters of a CUDA kernel. We describe a technique to build at the compile-time of a CUDA program a so-called rational program. The rational program, based on some performance prediction model, and knowing particular data and hardware parameters at runtime, can be executed to automatically and dynamically determine the values of launch parameters for the CUDA program that will yield nearly optimal performance. Our underlying technique could be applied to parallel programs in general, given a performance prediction model which accounts for program and hardware parameters. We have implemented and successfully tested our technique in the context of GPU kernels written in CUDA.

The remainder of this chapter is organized as follows. Section 5.2 formalizes and exemplifies the notion of rational programs and their relation to piece-wise rational functions and performance prediction. Section 5.3 gives on overview of the KLARAPTOR tool which applies our technique to CUDA kernels. The general algorithm underlying our tool, that is, building and using a rational program to predict program performance, is given in Section 5.4. Our implementation is detailed in Section 5.5, while our implementation is evaluated in Section 5.6. Lastly, we draw conclusions and explore future work in Section 5.7.

5.1 Introduction

Programming for high-performance parallel computing is a notoriously difficult task. Programmers must be conscious of many factors impacting performance including scheduling, synchronization, and data locality. Of course, program code itself impacts the program's performance, however, there are still further *parameters* which are independent from the code and greatly influence performance. For parallel programs three types of parameters influence performance:

1. *data parameters*, such as input data and its size;
2. *hardware parameters*, such as cache capacity and number of available registers; and
3. *program parameters*, such as granularity of tasks and the quantities that characterize how tasks are mapped to processors (e.g. dimension sizes of a thread block for a CUDA kernel).

Data and hardware parameters are independent from program parameters and are determined by the needs of the user and available hardware resources. Program parameters, however, are intimately related to data and hardware parameters. The choice of program parameters can largely influence the performance of a parallel program, resulting in orders of magnitude difference in timings (see Section 5.6). Therefore, it is crucial to determine values of program parameters that yield the best program performance for a given set of hardware and data parameter values.

In the CUDA programming model the kernel launch parameters, and thus the size and shape of thread blocks, greatly impact performance. This should be obvious considering that the memory accesses pattern of threads in a thread block can depend on the block's dimension sizes. The same could be said about multithreaded programs on CPU where parallel performance depends on task granularity and number of threads. Our general technique (see Section 5.4) is applied on top of some performance model to estimate program parameters which optimize performance. This could be applied to parallel programs in general, where performance models using program parameters exist. However, we dedicate this chapter to the discussion of GPU programs written in CUDA.

An important consequence of the impact of kernel launch parameters on performance is that an optimal thread block format (that is, dimension sizes) for one GPU architecture may not be optimal for another, as illustrated in [63]. This emphasizes not only the impact of hardware parameters on program parameters, but also the need for performance portability. That is to say, enabling users to efficiently execute the same parallel program on different architectures that belong to the same hardware platform.

In this chapter, we describe the development of KLARAPTOR (Kernel LAunch parameters RAtional Program estimaTOR), a tool for automatically and dynamically determining the values of CUDA kernel launch parameters which optimize the kernel's performance, for each kernel invocation independently. That is to say, based on the actual data and target device of a kernel invocation. The accuracy of KLARAPTOR's prediction is illustrated in Figure 5.1 where execution times are given for each kernel in the the PolyBench/GPU benchmark suite [29] on two different architectures. For each kernel, execution times are shown for three different thread block configurations: one chosen by KLARAPTOR, one resulting in the minimum time, and one resulting in the maximum time. The latter two are decided by an exhaustive search. In most cases, KLARAPTOR's prediction is very close to optimal; notice that the y-axis is log scaled. Further experimental results are reported in Section 5.6.

KLARAPTOR applies to CUDA a generic technique, also described herein in Section 5.4, to statically build a so-called *rational program* which is then used dynamically at runtime to determine optimal program parameters for a given multithreaded program on specific data and hardware parameters. The key principle is based on an observation of most performance metrics. In most performance prediction models, *high-level performance metrics*, such as execution time, memory consumption, and hardware occupancy, can be seen as decision trees or flowcharts based on *low-level performance metrics*, such as memory bandwidth and cache

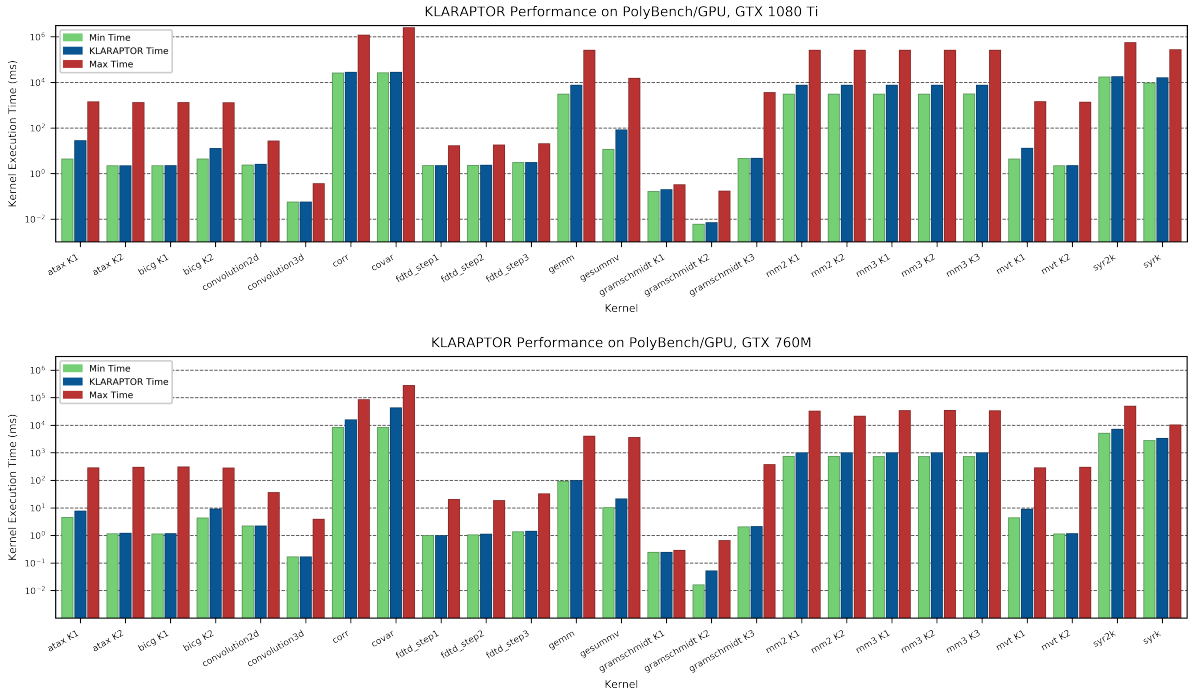


Figure 5.1: Comparing kernel execution time (log-scaled) for the thread block configuration chosen by KLARAPTOR versus the minimum and maximum times as determined by an exhaustive search over all possible configurations. Kernels are part of the `PolyBench/GPU` benchmark suite and executed on (1) a GTX 1080Ti with a data size of $N = 8192$ (except convolution3d with $N = 1024$), and (2) a GTX 760M with a data size of $N = 2048$ (except convolution3d with $N = 512$ and gemm with $N = 1024$).

miss rate. These low-level metrics are themselves piece-wise rational functions (PRFs) of program, data, and hardware parameters. This construction could be applied recursively to obtain a PRF for the high-level metric. We regard a computer program that computes such a PRF as a *rational program*, a technical notion defined in Section 5.2.

If one could determine these PRFs, then it would be possible to estimate, for example, the running time of a program based on its program, data, and hardware parameters. Unfortunately, exact formulas for low-level metrics are often not known, instead estimated through empirical measures or assumptions, or collected by profiling. This is a key challenge our technique addresses.

In most cases the values of the data parameters are only given at runtime, making it difficult to determine optimal values of the program parameters at an earlier stage. On another hand, a bad choice of program parameters can have drastic consequences. Hence, it is crucial to be able to determine the optimal program parameters at runtime without much overhead added to the program execution. This is precisely the intention of the approach proposed here.

5.1.1 Contributions

The goal of this work is to determine values of program parameters which optimize a multithreaded program's performance. Towards that goal, the method by which such values are found must be receptive to changing data and changing hardware parameters. Our contributions encapsulate this requirement through the dynamic use of a rational program. Our specific contributions include:

1. a technique for devising a mathematical expression in the form of a *rational program* to evaluate a performance metric from a set of program and data parameters;
2. KLARAPTOR, a tool implementing the rational program technique to dynamically optimize CUDA kernels by choosing optimal launch parameters; and
3. an empirical and comprehensive evaluation of our tool on kernels from the `Polybench/GPU` benchmark suite.

5.1.2 Related Works

The *Parallel Random Access Machine* (PRAM) model [60, 27], including PRAM models tailored to GPU code analysis such as TMM [45] and MCM [31] analyze the performance of parallel programs at an abstract level. More detailed GPU performance models are proposed such as MWP-CWP [34, 58], which estimates the execution time of GPU kernels based on the profiling information of the kernels.

In the context of improving CUDA program performance, other research groups have used techniques such as loop transformation [12], auto-tuning [30, 38, 54, 40], dynamic instrumentation [39], or a combination of the latter two [59]. Auto-tuning techniques have achieved great results in projects such as ATLAS [68], FFTW [23], and SPIRAL [53] in which multiple kernel versions are generated *off-line* and then applied and refined *on-line* once the runtime parameters are known. In contrast, our technique does not optimize the parallel code itself, only the program parameters controlling it.

Although much research has been devoted to compiler optimizations for kernel source code or PTX code, previous works such as [16] and [63] suggest that kernel launch parameters (i.e. thread block configurations) have a large impact on performance and must be considered as a target for optimization. In [43], the authors present an input-adaptive GPU code optimization framework G-ADAPT, which uses statistical learning to find a relation between the input sizes and the thread block sizes. At linking time, the framework predicts the best block size for a given input size using the linear model obtained from compile time. This approach only considers the total size of the thread blocks and not their configuration. Meanwhile, the authors of [54] use a linear regression model to predict optimal thread block configurations (that is, dimension sizes and not just the total size). However, they assume kernel execution time scales linearly with data size. The authors in [42] have also developed a method determining the best thread block configuration, but similarly, they assume execution time scales linearly with data size. In [25], machine learning techniques are used in combination with auto-tuning to search for optimal configurations of OpenCL kernels, but their examples are limited to stencil computations.

5.2 Theoretical Foundations

Let $\mathcal{P}$ be a multithreaded program to be executed on a targeted multiprocessor. Parameters influencing its performance include

1. *data parameters*, describing size and structure of the data;
2. *hardware parameters*, describing hardware resources and their capabilities; and
3. *program parameters*, characterizing parallel aspects of the program (e.g. how tasks are mapped to hardware resources).

By fixing the target architecture, the hardware parameters, say, $\boldsymbol{H} = (H_1, \ldots, H_h)$ become fixed and we can assume that the performance of $\mathcal{P}$ depends only on data parameters $\boldsymbol{D} = (D_1, \ldots, D_d)$ and program parameters $\boldsymbol{P} = \left(P_1, \ldots, P_p\right)$. Moreover, an optimal choice of $\boldsymbol{P}$ naturally depends on a specific choice of $\boldsymbol{D}$. For example, in programs targeting GPUs the parameters $\boldsymbol{D}$ are typically dimension sizes of data structures, like arrays, while $\boldsymbol{P}$ typically specifies the formats of thread blocks.

Let $\mathcal{E}$ be a high-level performance metric for $\mathcal{P}$ that we want to optimize. More precisely, given the values of the data parameters $\boldsymbol{D}$, the goal is to find values of the program parameters $\boldsymbol{P}$ such that the execution of $\mathcal{P}$ optimizes $\mathcal{E}$. Performance prediction models attempt to estimate $\mathcal{E}$ from a combination of $\boldsymbol{P}$, $\boldsymbol{D}$, $\boldsymbol{H}$, and some model- or platform-specific low-level metrics $\boldsymbol{L} = (L_1, \ldots, L_\ell)$. It is natural to assume that these low-level performance metrics are themselves combinations of $\boldsymbol{P}$, $\boldsymbol{D}$, $\boldsymbol{H}$. This is an obvious observation from models based on PRAM such as TMM [45] and MCM [31].

Therefore, we look to obtain values for these low- and high-level metrics given values for program, and data parameters. To address our goal, we compute a mathematical expression for each metric, parameterized by data and program parameters, in the format of a *rational program* at the compile-time of $\mathcal{P}$. At the runtime of $\mathcal{P}$, given the specific values of $\boldsymbol{D}$ and a choice of $\boldsymbol{P}$, we can evaluate the rational programs to obtain a value for each metric and thus $\mathcal{E}$. These values can be used to determine which choice of $\boldsymbol{P}$ optimizes overall program performance. This method is detailed in Section 5.4. We take this section to define the rational program itself.

One could view a rational program as simply a computer program evaluating some performance-predicting model. However, as we will see in the following sections, it is more than that. Specifically, the encoding of some model as a flow chart whose nodes can then be approximated as a rational function is a powerful idea which can be used to simplify models and extrapolate results.

5.2.1 Rational Programs

Let $X_1, \ldots, X_n, Y$ be pairwise different variables[1]. Let $\mathcal{S}$ be a sequence of three-address code (TAC [5]) instructions such that the set of the variables that occur in $\mathcal{S}$ and are never assigned a value by an instruction of $\mathcal{S}$ is exactly $\{X_1, \ldots, X_n\}$.

Definition 2 *We say that the sequence $\mathcal{S}$ is* rational *if every arithmetic operation used in $\mathcal{S}$ is either an addition, a subtraction, a multiplication, or a comparison of integer numbers in either*

[1] Variables refer to both its mathematical meaning and programming language concept.

fixed or arbitrary precision. We say that the sequence $\mathcal{S}$ is a rational program *in $X_1, \ldots, X_n$ evaluating Y if the following two conditions hold:*

1. *$\mathcal{S}$ is rational, and*
2. *after specializing each of $X_1, \ldots, X_n$ to some integer value in $\mathcal{S}$, the execution of the specialized sequence always terminates and the last executed instruction assigns an integer value to Y.*

It is worth noting that the above definition can easily be extended to include Euclidean division, the integer part operations floor and ceiling, and arithmetic over rational numbers. For Euclidean division one can write a rational program evaluating the quotient q of integer a by b, leaving the remainder r to be simply calculated as $a - qb$. Then, floor and ceiling can be computed via Euclidean division. Rational numbers and their associated arithmetic are easily implemented using only integer arithmetic. Therefore, by adding these operations to Definition 2, the class of rational programs does not change. We regard rational programs as such henceforth.

5.2.2 Rational Programs as Flowcharts

For any sequence $\mathcal{S}$ of computer program instructions, one can associate $\mathcal{S}$ with a *control flow graph* (CFG). In the CFG of $\mathcal{S}$, the nodes are the *basic blocks* of $\mathcal{S}$. Recall that a *flowchart* is another graphic representation of a sequence of computer program instructions. In fact, CFGs can be seen as particular flowcharts.

If, in a given flowchart C, every arithmetic operation occurring in every (process or decision) node is either an addition, subtraction, multiplication, or comparison of integers in either fixed or arbitrary precision then C is the flowchart of a rational sequence of computer program instructions. Therefore, it is meaningful to depict rational programs using flowcharts, and vice versa, flowcharts as rational programs. For example, one could consider the metric of theoretical *hardware occupancy* as defined by NVIDIA. The following example details its definition, its depiction as a flowchart, and its dependency on program, data, and hardware parameters.

Example 1 *Hardware occupancy*, as defined in the CUDA programming model, is a measure of a program's effectiveness in using the Streaming Multiprocessors (SMs) of a GPU. Theoretical occupancy is calculated from a number of hardware parameters, namely:

- the maximum number $R_{\max}$ of registers per thread block,
- the maximum number $Z_{\max}$ of shared memory words per thread block,
- the maximum number $T_{\max}$ of threads per thread block,
- the maximum number $B_{\max}$ of thread blocks per SM and
- the maximum number $W_{\max}$ of warps per SM,

as well as low-level kernel-dependent performance metrics, namely:

- the number R of registers used per thread and
- the number Z of shared memory words used per thread block,

and a program parameter, namely the number T of threads per thread block. The occupancy of a CUDA kernel is defined as the ratio between the number of active warps per SM and the maximum number of warps per SM, namely $W_{\text{active}}/W_{\max}$, where

$$W_{\text{active}} \leq \min\left(\lfloor B_{\text{active}} T/32 \rfloor, W_{\max}\right) \tag{5.1}$$

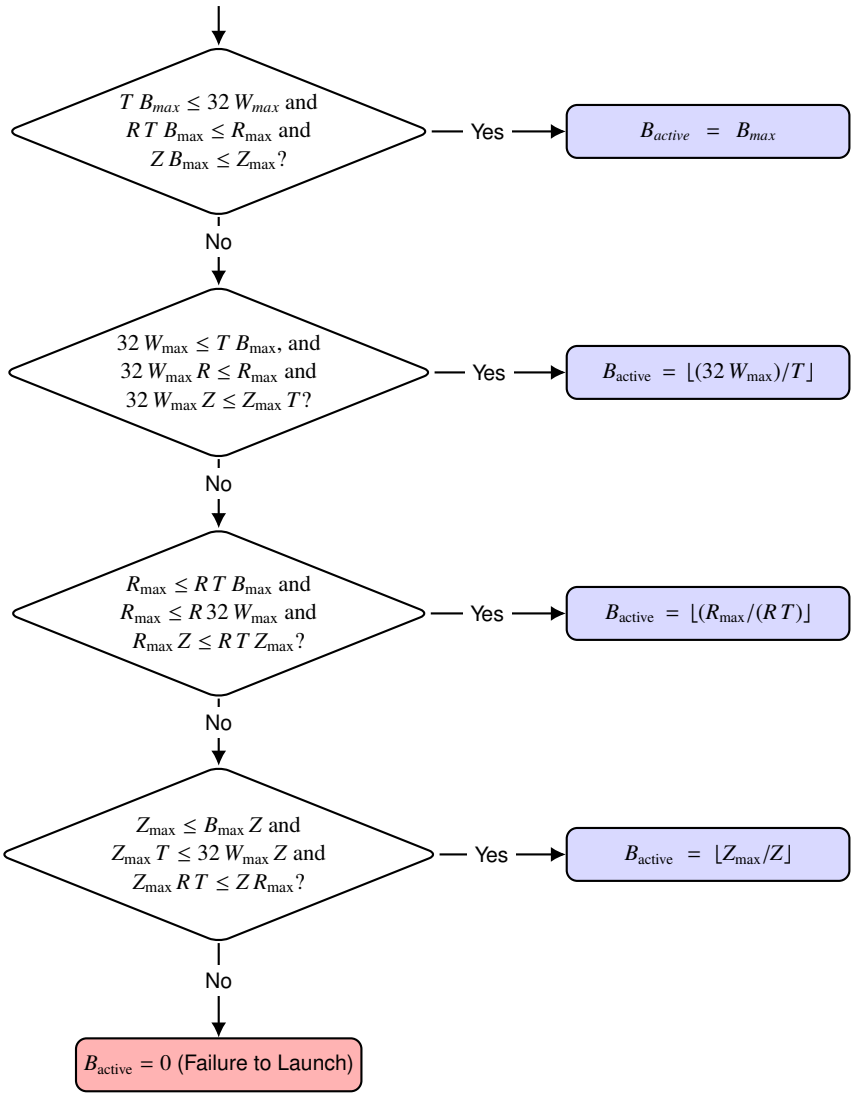


Figure 5.2: Rational program (presented as a flow chart) for the calculation of active blocks in CUDA.

and B_{active} is given by the flowchart in Figure 5.2. This flowchart shows how one can derive a rational program computing B_{active} from $R_{\max}$, $Z_{\max}$, $T_{\max}$, $B_{\max}$, $W_{\max}$, R, Z, T. It follows from formula (5.1) that W_{active} can also be computed by a rational program from $R_{\max}$, $Z_{\max}$, $T_{\max}$, $B_{\max}$, $W_{\max}$, R, Z, T. Finally, the same is true for theoretical occupancy of a CUDA kernel using W_{active} and $W_{\max}$.

5.2.3 Piece-Wise Rational Functions in Rational Programs

We begin with an observation describing the fact that a rational program can be viewed as a piece-wise rational function [2] .

[2]Here, rational function is in the sense of algebra, see Section 5.5.4.

Observation 1 Let $\mathcal{S}$ be a rational program in $X_1, \ldots, X_n$ evaluating Y. Let s be any instruction of $\mathcal{S}$ other than a branch or an integer part instruction. Hence, this instruction can be of the form $C = -A$, $C = A + B$, $C = A - B$, $C = A \times B$, where A and B can be any rational number. Let $V_1, \ldots, V_v$ be the variables that are defined at the entry point of the basic block of the instruction s. An elementary proof by induction yields the following fact. There exists a rational function in $V_1, \ldots, V_v$ that we denote by $f_s(V_1, \ldots, V_v)$ such that $C = f_s(V_1, \ldots, V_v)$ for all possible values of $V_1, \ldots, V_v$. From there, one derives the following observation. There exists a partition $\mathcal{T} = \{T_1, T_2, \ldots\}$ of $\mathbb{Q}^n$ (where $\mathbb{Q}$ denotes the field of rational numbers) and rational functions $f_1(X_1, \ldots, X_n)$, $f_2(X_1, \ldots, X_n)$, ... such that, if $X_1, \ldots, X_n$ receive respectively the values $x_1, \ldots, x_n$, then the value of Y returned by $\mathcal{S}$ is one of $f_i(x_1, \ldots, x_n)$ where i is such that $(x_1, \ldots, x_n) \in T_i$ holds. In other words, $\mathcal{S}$ computes Y as a *piece-wise rational function* (PRF). Notice that, trivially, if $\mathcal{S}$ contains only one basic block, then $\mathcal{S}$ can be given by a single rational function.

Example 1 shows that the hardware occupancy of a CUDA kernel is given as a piece-wise rational function in the variables $R_{\max}$, $Z_{\max}$, $T_{\max}$, $B_{\max}$, $W_{\max}$, R, Z, T. Hence, in this example, we have $n = 8$, and, as shown by Figure 5.2, its partition of $\mathbb{Q}^n$ contains 5 parts as there are 5 terminating nodes in the flowchart.

Suppose that a flowchart C representing the rational program $\mathcal{R}$ is partially known; to be precise, suppose that the decision nodes are known (that is, mathematical expressions defining them are known) while the process nodes are not. Then, from Observation 1, each process node can be given by a series of one or more rational functions. Trivially, a single formula can also be seen as a flowchart with a single process node. Determining each of those rational functions can be achieved by solving an *interpolation* or *curve fitting* problem. More generally, if the sequence of instructions in a process node involves non-rational functions (e.g. log) we can apply Stone-Weierstrass Theorem [61] to approximate each of those by a PRF.

It then follows that any performance metric, which can be depicted as a flow chart or a formula, can also be represented as a piece-wise rational function, and thus a rational program. For high-level performance metrics, which rely on low-level metrics, one could work recursively, first determining rational programs for the low-level metrics which depend on $\boldsymbol{P}$, $\boldsymbol{D}$, and $\boldsymbol{H}$, and then constructing a rational program for the high-level metric from those rational programs. Hence, by this recursive construction, we can fully determine a rational program for a high-level metric depending only on $\boldsymbol{P}$, $\boldsymbol{D}$, and $\boldsymbol{H}$. Of course, hardware parameters could be fixed given a target architecture to yield a rational program which depends only on $\boldsymbol{P}$ and $\boldsymbol{D}$. Again, notice that even where formulas for low-level metrics are not known, it is still possible to estimate them as PRFs, and thus rational programs, via a curve fitting.

As an example, consider occupancy (Example 1). One could first determine PRFs for the number of registers user per thread and the amount of shared memory used per thread block. Then, a PRF is determined for the number of active blocks (Figure 5.2) from these two low-level metrics, and a few more hardware and program parameters. Thus, by recursive construction, we have a PRF depending only on program and hardware parameters.

Lastly, we make one final remark. We assumed that the decision nodes in the flowchart of the rational program were known, however, we could relax this assumption. Indeed, each decision node is given by a series of rational functions. Hence, those could also be determined by solving curve fitting problems. However, we do not discuss this further since it is not needed

in our proposed technique or implementation presented in the remainder of this chapter.

5.3 KLARAPTOR: A Dynamic Optimization tool for CUDA

The theory of rational programs is put into practice for the CUDA programming model by our tool KLARAPTOR. KLARAPTOR is a compile-time tool implemented using the LLVM Pass Framework and the MWP-CWP performance model to dynamically choose a CUDA kernel's launch parameters (thread block configuration) which optimize its performance. Most high-performance computing applications require computations be as fast as possible and so kernel performance is simply measured as its execution time.

As mentioned in Section 5.1, thread block configurations drastically affect the running time of a kernel. Determining optimal thread block configurations typically follows some heuristics, for example, constraining block size to be a multiple of 32 [2]. However, it is known that the dimension sizes of a thread block, not only its total size, affect performance [63, 16]. Moreover, since thread block configurations are intimately tied to the size of data being operated on, it is very unlikely that a static thread block configuration optimizes the performance of all data sizes. Our tool effectively uses rational programs to dynamically determine the thread block configuration which minimizes the execution time of a particular kernel invocation, considering the invocation's particular data size and target architecture. This is achieved in two main steps.

1. At the compile-time of a CUDA program, its kernels are analyzed in order to build rational programs estimating the performance metrics for each individual kernel under the MWP-CWP model. Each rational program, written as code in the C language, is inserted into the code of the CUDA program so that it is called before the execution of the corresponding kernel.
2. At runtime, immediately preceding the launch of a kernel, where data parameters have specific values, the rational program is evaluated to determine the thread block configuration which optimizes the performance of the kernel. The kernel is then launched using this thread block configuration.

Not only are we concerned with kernel performance, but also programmer performance. By that, we mean the efficiency of a programmer to produce optimal code. When a programmer is attempting to optimize a kernel, choosing optimal launch parameters can either be completely ignored, performed heuristically, determined by trial and error, or determined by an exhaustive search. The latter two options quickly become infeasible as data sizes grow large. Regardless, any choice of optimal thread block configuration is likely to optimize only a single data size.

For KLARAPTOR to be practical, not only does the choice of optimal kernel launch parameters need to be correct, but its two main steps must also be performed in an manner more efficient than trial and error or exhaustive search. Namely, the compile-time analysis cannot add too much overhead to the the compilation time and the runtime decision of the kernel launch parameters cannot overwhelm the program execution time. For the former, our analysis is performed quickly by analyzing kernel performance on only small data sizes, and then results are extrapolated. The time for this process typically ranges between 30 seconds and 2 minutes (see Section 5.6). For the later, the rational program evaluation is quick and simple, being only the evaluation of a few rational functions. Moreover, we maintain a runtime invocation history to instantly provide results for future kernel launches. Our implementation is

detailed in Section 5.5.

We have made use of the `Polybench/GPU` benchmark suite as an empirical evaluation of the correctness of our tool on a range of CUDA programs. In Figure 5.1 we have already seen that KLARAPTOR accurately predicts the optimal or near-optimal thread block configuration. Before presenting more detailed results and experimentation in Section 5.6, we describe the steps followed by our tool to build and use rational programs for determining a thread block configuration which optimizes performance.

5.4 An Algorithm to Build and Deploy Rational Programs to Select Program Parameters

In this section the notations and hypotheses are the same as in Section 5.2. Namely, $\mathcal{E}$ is a high-level performance metric for the multithreaded program $\mathcal{P}$, $\boldsymbol{L}$ is a set of low-level metrics of size ℓ, and $\boldsymbol{P}$, $\boldsymbol{D}$, $\boldsymbol{H}$ are sets of program, data, and hardware parameters, respectively. Recall $\boldsymbol{P}$ has size p. Let us assume that the values of $\boldsymbol{H}$ are known at the compile-time of $\mathcal{P}$ while the values of $\boldsymbol{D}$ are known at runtime. Further, let us assume that $\boldsymbol{P}$ and $\boldsymbol{D}$ take integer values. Hence the values of $\boldsymbol{P}$ belong to a finite set $F \subset \mathbb{Z}^p$. That is to say, the possible values of $\boldsymbol{P}$ are tuples of the form $(\pi_1, \ldots, \pi_p) \in F$, with each π_i being an integer. Let us call such a tuple a *configuration* of the program parameters. Due to the nature of program parameters, those are not necessarily all independent variables For example, in CUDA, the product of the dimension sizes of a thread block is usually a multiple of the warp size (32).

Given a performance-prediction model for $\mathcal{E}$, one could work recursively to determine a single rational program $\mathcal{R}$, depending on only $\boldsymbol{D}$ and $\boldsymbol{P}$, evaluating $\mathcal{E}$, from a combination of rational programs constructed for each low-level metric in $\boldsymbol{L}$ and values of $\boldsymbol{D}$ and $\boldsymbol{P}$. Following Section 5.2.3, each of these rational programs are constructed by computing rational functions. Without loss of generality, let us assume each low-level metric is given by a single formula and thus a single rational function. Hence, we look to determine $g_1(\boldsymbol{D}, \boldsymbol{P})$, …, $g_\ell(\boldsymbol{D}, \boldsymbol{P})$ for the ℓ low-level metrics. Finally, at runtime, given particular values of $\boldsymbol{D}$, the rational program for $\mathcal{E}$ can be evaluated for various values of $\boldsymbol{P}$ to determine the optimal. In the context of CUDA where we look to optimize execution time, the selection of program parameters leading to optimal performance is a complex task, currently we relies on the MWP-CWP model to do the selection. In the remainder of this section we describe the general process to build and use rational programs to determine optimal configurations. The entire process is decomposed into six steps: the first three occur at compile-time and the next three at runtime.

1. **Data collection**: To perform a curve fitting of the rational functions $g_1(\boldsymbol{D}, \boldsymbol{P})$, …, $g_\ell(\boldsymbol{D}, \boldsymbol{P})$ we require data points to fit. These are collected by
   (a) selecting a subset of K points from the space of possible values of $(\boldsymbol{D}, \boldsymbol{P})$; and
   (b) executing the program $\mathcal{P}$, recording the values of the low-level performance metrics $\boldsymbol{L}$ as $\boldsymbol{V} = (V_1, \ldots, V_\ell)$, at each point in K.

   Data used for executing the programs is generated randomly, but could follow some scheme provided by the user.
2. **Rational function approximation**: For each low-level metric L_i we use the set of points K and the corresponding value V_i measured for each point to approximate the rational

function $g_i(\boldsymbol{D}, \boldsymbol{P})$. We observe that if these values were known exactly the rational function $g_i(\boldsymbol{D}, \boldsymbol{P})$ could be determined exactly. In practice, however, these empirical values are likely to be noisy from profiling, and/or numerical approximations. Consequently, we actually determine a rational function $\widehat{g_i}(\boldsymbol{D}, \boldsymbol{P})$ which approximates $g_i(\boldsymbol{D}, \boldsymbol{P})$.

3. **Code generation**: In order to generate the rational program $\mathcal{R}$, we proceed as follows:
   (i) we convert the rational program representing $\mathcal{E}$ into code, essentially encoding the CFG for computing $\mathcal{E}$;
   (ii) we convert each $\widehat{g_i}(\boldsymbol{D}, \boldsymbol{P})$ into code, specifically sub-routines, estimating L_i; and
   (iii) we include those sub-routines into the code computing $\mathcal{E}$, which yields the desired rational program $\mathcal{R}$, fully constructed, and depending only on $\boldsymbol{D}$ and $\boldsymbol{P}$.
4. **Rational program evaluation**: At the runtime of $\mathcal{P}$, the data parameters $\boldsymbol{D}$ are given particular values. For those specified values of $\boldsymbol{D}$ and for all practically meaningful values of $\boldsymbol{P}$ from the set F,[3] we compute an estimate of $\mathcal{E}$ using $\mathcal{R}$. The evaluation of $\mathcal{E}$ over so many different possible program parameters is feasible for three reasons:
   (i) the number of program parameters is small, typically $p \leq 3$, see Section 5.5;
   (ii) the set of meaningful values for $\boldsymbol{P}$ is small (consider that in CUDA the product of thread block dimension sizes should be a multiple of 32 less than 1024); and
   (iii) the program $\mathcal{R}$ simply evaluates a few polynomial formulae and thus runs almost instantaneously.
5. **Selection of optimal values of program parameters**:
   When the search space of values of program parameters $\boldsymbol{P}$ is large, a numerical optimization technique is required for this step. But, as just explained, the total number of evaluations is quite small and thus an exhaustive search is feasible to determine an optimal configuration. However, it is possible that due to inaccuracies in the performance prediction model being used, and in the approximation of the rational functions $g_i(\boldsymbol{D}, \boldsymbol{P})$ several configurations, up to some margin, optimize $\mathcal{E}$. Then, a secondary performance metric or some heuristic specific to the platform of $\mathcal{P}$ may be used to refine the choice of optimal configuration.
6. **Program execution**: Once an optimal configuration is selected, the program $\mathcal{P}$ can finally be executed using this configuration along with the values of $\boldsymbol{D}$.

5.5 The Implementation of KLARAPTOR

In this section we give an overview of the implementation of our previously presented technique (Sections 5.4) specialized to CUDA using the MWP-CWP model in the KLARAPTOR tool. Our tool is built in the C language, making use of the LLVM Pass Framework (see Section 5.5.2) and the NVIDIA CUPTI API (see Section 5.5.3). KLARAPTOR is freely available in source at https://github.com/orcca-uwo/KLARAPTOR.

In the case of a CUDA kernel, the data parameters specify the input data size. In many examples this is a single parameter, say N, describing the size of an array (or the order of a multi-dimensional array), the values of which are usually powers of 2. Program parameters

[3]The values for $\boldsymbol{P}$ are likely to be constrained by the values $\boldsymbol{D}$. For example, if P_1, P_2 are the two dimension sizes of a two-dimensional thread block of a CUDA kernel operating on a square matrix of order D_1, then $P_1 P_2 \leq D_1^2$ is meaningful.

describe the kernel launch parameters, i.e. grid and thread block dimension sizes, and are also typically powers of 2. For example, a possible thread block configuration may be $1024 \times 1 \times 1$ (a one-dimensional thread block), or $16 \times 16 \times 2$ (a three-dimensional thread block). Lastly, the hardware parameters are values specific to the target GPU, for example, memory bandwidth, the number of SMs available, and their clock frequency.

We organize this section as follows. Sections 5.5.1 and 5.5.2 are specific to our implementation and do not correspond to any step of Section 5.4. The compile time steps 1 (data collection) and 2 (rational function estimation) are reflected in Sections 5.5.3 and 5.5.4, respectively, while step 3 requires no explanation. The runtime steps 4 (rational program evaluation), 5 (selection of optimal configuration) and 6 (program execution) are trivial to perform, Throughout this section we refer to the notion of a *driver program* as the code, for each individual kernel, using a rational program to select a configuration.

5.5.1 Annotating and Preprocessing Source Code

Beginning with a CUDA program written in C/C++, we minimally annotate the host code to make it compatible with our *pre-processor*. We take into account the following points:

(i) the code targets at least CUDA Compute Capability (CC) 3.x;
(ii) there should be no CUDA runtime API calls as such calls will interfere with later CUDA driver API calls used by our tool, for example, `cudaSetDevice`;
(iii) the block dimensions and grid dimensions must be declared as the typical CUDA `dim3` structs.

For each kernel in the CUDA code, we add two pragmas, one specifying the dimension of the kernel (1, 2, or 3), and one defining the index of the kernel input arguments corresponding to the data size N. As an example, consider the code segment below of a CUDA kernel and the associated pragmas. This kernel operates of a two-dimensional array of order N, making it a two-dimensional kernel.

```
#pragma kernel_info_size_param_idx_Sample = 1;
#pragma kernel_info_dim_sample_kernel = 2;
__global__ void Sample (int *A, int N) {
    int tid_x = threadIdx.x + blockIdx.x*blockDim.x;
    int tid_y = threadIdx.y + blockIdx.y*blockDim.y;
    ...
}
```

Lastly, for each kernel, the user must fill two formatted configuration files which follow `Python` syntax. One specifies the constraints on the thread block configuration while the other specifies the grid dimensions. For example, for the 2D kernel `Sample` above, one could specify that its thread block configuration (bx, by, bz) must satisfy $bx < by^2$, $bx < N$ and $by < N$. Since the kernel dimension is given as 2, we assume $bz = 1$. Similarly, the grid dimensions (gx, gy, gz), could be specified as $gx = \lceil \frac{N}{bx} \rceil$, $gy = \lceil \frac{N}{by} \rceil$, $gz = 1$.

Now, a preprocessor processes the annotated source code, replacing CUDA runtime API calls with driver API kernel launches. This step includes source code analysis in order to extract a list of kernels, a list of kernel calls in the host code, and finally, the body of each kernel to be used for further analysis. A so-called "PTX lookup table" is built to store kernel information and static parameters. This table will be inserted into the "instrumented binary", the compiled

CUDA program augmented by the driver programs.

5.5.2 Input/Output Builder

The Input/Output builder Pass, or IO-builder, is a compiler pass written in the LLVM Pass Framework to build the previously mentioned "instrumented binary". This pass embeds an IO mechanism (i.e. a function call) to communicate with the driver program of a kernel for each of its invocations. Thus, at the runtime of the CUDA program being analyzed (step 6 of Section 5.4), an IO function is called before each kernel invocation to return six integers, (gx, gy, gz, bx, by, bz), the optimal kernel launch parameters.

The IO-builder pass goes through the following steps:

1. obtain the LLVM intermediate representation of the instrumented source code and find all CUDA driver API kernel calls;
2. relying on the annotated information for each kernel, determine which variables in the IR contain the value of N for a corresponding kernel call; and
3. insert a call to an IO function immediately before each kernel call in order to pass the runtime value of N to the corresponding driver program and retrieve the optimal kernel launch parameters.

5.5.3 Building a Driver Program: Data Collection

For the MWP-CWP model as well as our eventual rational function estimation, we must collect data and statistics regarding certain performance counters and runtime metrics (these metrics are thoroughly defined in [34] and [1]). These metrics can be partitioned into three categories.

Firstly, *architecture-specific performance counters* of a kernel, characteristics influenced by the CC of the target device. These can be obtained at compile-time, since the target CC is specified at this time. These characteristics include the number of registers used per thread, the amount of static shared memory per thread block, and the number of (arithmetic and memory) instructions per thread.

Secondly, *runtime-specific performance counters* that depend on the behavior of the kernel at runtime. This includes values impacted by memory access patterns, namely, the number of memory accesses per warp, the number of memory instructions of each thread, and the total number of warps that are being executed. We have developed a customized profiler using NVIDIA's EVENT API within the CUPTI API to collect the required runtime performance counters. The profiler is customized to collect only the required information, allowing it to be very lightweight and avoid the huge overheads of a typical profiler (e.g. NVIDIA's NVPROF [4]).

Thirdly, *device-specific parameters*, which describe an actual GPU card, allow us to compute a more precise performance estimate. A subset of such parameters can be determined by microbenchmarking the device (see [47] and [71]), this includes the memory bandwidth, and the departure delay for memory accesses. The remaining parameters can easily obtained by consulting the vendor's guide [3], or by querying the device itself via the CUDA driver API. Such parameters include the number of SMs on the card, the clock frequency of SM cores, and the instruction delay.

5.5.4 Building a Driver Program: Rational Function Approximation

Using the runtime data collected in the previous step, we look to determine the rational functions $\widehat{g_i}(\boldsymbol{D}, \boldsymbol{P})$ (see Section 5.4) which estimate the low-level metrics or other intermediate values in the rational program. For ease of discussion, we replace the parameters $\boldsymbol{D}$ and $\boldsymbol{P}$ with the generic variables $X_1, \ldots, X_n$.

A rational function is simply a fraction of two polynomials:

$$f(X_1, \ldots, X_n) = \frac{\alpha_1 \cdot (X_1^0 \cdots X_n^0) \ + \ \ldots \ + \ \alpha_i \cdot (X_1^{u_1} \cdots X_n^{u_n})}{\beta_1 \cdot (X_1^0 \cdots X_n^0) \ + \ \ldots \ + \ \beta_j \cdot (X_1^{v_1} \cdots X_n^{v_n})} \tag{5.2}$$

With a *degree bound* (an upper limit on the exponent) on each variable X_k in the numerator and the denominator, u_k and v_k, respectively, these polynomials can be defined up to some *parameters* (using the language of parameter estimation), namely the coefficients of the polynomials, $\alpha_1, \ldots, \alpha_i$ and $\beta_1, \ldots, \beta_j$. Through algebraic analysis of performance models like the MWP-CWP model, and empirical evidence, these degree bounds are relatively small.

We perform a parameter estimation (for each rational function) on the coefficients $\alpha_1, \ldots, \alpha_i$, $\beta_1, \ldots, \beta_j$ to determine the rational function precisely. This is a simple linear regression which can be solved by an over-determined system of linear equations, say by the method of linear least squares. However, the system suffers from *multicollinearity* (see [26, Chapter 23]) and can become rank-deficient. Solving using the typical QR-factorization is impossible; hence we use the computationally more intensive yet more numerically stable method of *singular value decomposition* (SVD; for details see [20, Chapter 4]).

Our implementation uses LAPACK [6] for SVD and the Basic Polynomial Algebra Subprograms (BPAS) library [7] for efficient rational function and polynomial implementations.

5.6 Experimentation

In this section we examine the performance of KLARAPTOR by applying it to the CUDA programs of the `Polybench/GPU` benchmark suite [30]. We note here that many of the kernels in this suite perform relatively low amounts of work; they are best suited to being executed many times from a loop in the host code. Data in this section was collected using a GTX 1080Ti.

Table 5.1 provides experimental data for the main kernels in the benchmark suite `Polybench/GPU`. Namely, this table compares the execution times of the thread block configuration chosen by KLARAPTOR against the optimal thread block configuration found though exhaustive search. The table shows a couple of data sizes in order to highlight that the best configuration can change for different input sizes. While it may appear for some examples that there are large variations between timings of the KLARAPTOR-chosen configuration and the optimal, these should be considered within the full range of possible configurations. Recall from Figure 5.1 that compared to the worst possible timings, the KLARAPTOR-chosen configuration and the optimal result in very similar in timings.

In Figure 5.3 we compare the time it takes KLARAPTOR to perform its compile-time analysis and build the rational programs for each example in the `PolyBench/GPU` suite. This is compared against determining the optimal thread block configuration by an exhaustive search.

Table 5.1: KLARAPTOR vs. exhaustive search for thread block configuration choice for kernels in `Polybench/GPU`.

Kernel	N	KLARAPTOR Time (ms)	Chosen Config.	Optimal Time (ms)	Optimal Config.
atax K1	4096	2.35	32, 4	0.85	32, 1
	8192	27.83	1, 64	4.33	16, 2
atax K2	4096	1.09	16, 2	1.04	32, 1
	8192	2.20	32, 1	2.19	64, 1
bicg K1	4096	1.05	256, 1	1.05	32, 1
	8192	2.23	256, 1	2.21	64, 1
bicg K2	4096	1.15	8, 4	0.85	32, 1
	8192	12.58	256, 4	4.35	512, 1
convolution2d	4096	0.79	256, 1	0.77	32, 4
	8192	2.54	256, 4	2.35	32, 4
corr	4096	5700.65	256, 1	5075.77	32, 1
	8192	27846.91	256, 1	26024.94	32, 1
covar	4096	5682.96	256, 1	5076.77	32, 1
	8192	27865.89	256, 1	26182.65	32, 1
fdtd_step1	4096	0.56	256, 1	0.56	32, 2
	8192	2.22	256, 4	2.22	32, 4
fdtd_step2	4096	0.58	256, 1	0.58	512, 1
	8192	2.33	32, 16	2.30	512, 1
fdtd_step3	4096	0.77	256, 1	0.77	512, 2
	8192	3.06	256, 4	3.05	1024, 1
gemm	4096	723.29	256, 1	386.76	32, 32
	8192	7481.13	256, 1	3069.66	32, 16
gesummv	4096	8.19	2, 16	1.62	32, 1
	8192	82.21	32, 16	11.58	64, 1
gramschmidt K1	4096	0.09	4, 32	0.09	256, 1
	8192	0.20	8, 32	0.17	64, 1
gramschmidt K2	4096	0.01	32, 2	0.01	256, 1
	8192	0.01	512, 2	0.01	256, 1
gramschmidt K3	4096	2.15	256, 1	2.11	32, 1
	8192	4.68	256, 1	4.61	32, 1
mm2 K1	4096	695.23	256, 1	384.93	32, 32
	8192	7531.13	256, 1	3062.26	32, 16
mm2 K2	4096	761.49	256, 1	386.61	32, 32
	8192	7533.08	256, 1	3077.75	32, 16
mm3 K1	4096	749.27	256, 1	388.40	32, 32
	8192	7531.56	256, 1	3065.34	32, 16
mm3 K2	4096	816.08	256, 1	389.13	32, 16
	8192	7532.66	256, 1	3067.87	32, 16
mm3 K3	4096	737.21	256, 1	392.81	32, 16
	8192	7530.24	256, 1	3085.43	32, 16
mvt K1	4096	1.15	8, 4	0.86	32, 1
	8192	12.90	256, 4	4.35	16, 2
mvt K2	4096	1.05	256, 1	1.05	32, 1
	8192	2.23	256, 1	2.21	128, 1
syr2k	4096	7050.62	1, 64	2097.15	4, 32
	8192	18013.51	16, 64	17398.88	4, 8
syrk	4096	2973.88	2, 16	1165.24	16, 16
	8192	15936.21	32, 16	9368.56	16, 16

Since KLARAPTOR's compile-time analysis is a one-time occurrence which optimizes for all data sizes, exhaustive search times are given as a sum for data sizes up to $N = 8192$. The best and worst execution times for the main kernel in each example (for $N = 8192$) is also given to highlight the fact that our optimization step is sometimes faster than even a single execution of a kernel with a poor choice of thread block configuration. We note that for some kernels, with very quick running times, exhaustive search is not a bad option. However, some examples such as `GRAMSCHMIDT`, take an exorbitant amount of time for exhaustive search. This figure also shows that the one-time compile-time cost of optimization can often be amortized by only a few executions of the kernel.

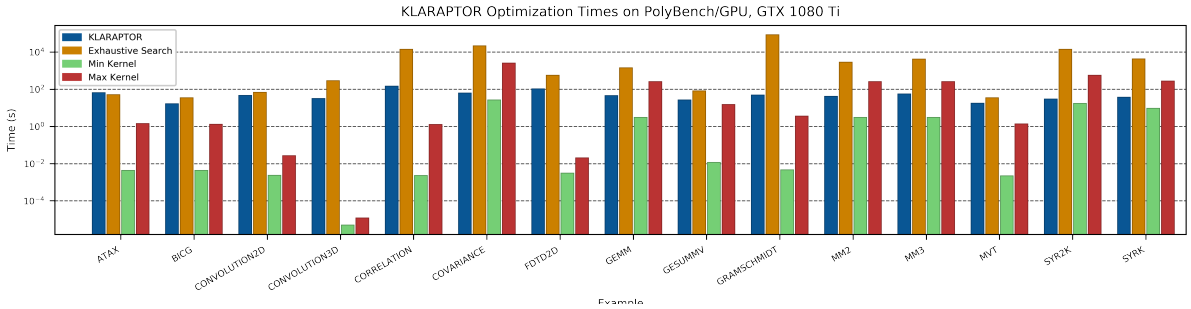


Figure 5.3: Comparing times (log-scaled) for (1) compile-time optimization steps of KLARAPTOR, (2) exhaustive search over all thread block configurations, the execution time for a kernel given (3) the best thread block configuration, and (4) the worst thread block configuration. Exhaustive search is given as a sum for values up to $N = 8192$ (except convolution3d with $N = 1024$).

5.7 Conclusions and Future Work

The performance of a single CUDA program can vary wildly depending on the target GPU device, the input data size, and the kernel launch parameters. Moreover, a thread block configuration yielding optimal performance for a particular data size or a particular target device will not necessarily be optimal for a different data size or different target device. In this chapter we have presented the KLARAPTOR tool for determining optimal CUDA thread block configurations for a target architecture, in a way which is adaptive to each kernel invocation and input data, allowing for dynamic data-dependent performance and portable performance. This tool is based upon our technique of encoding a performance prediction model as a rational program. The process of constructing such a rational program is a fast and automatic compile-time process which occurs simultaneously to compiling the CUDA program by use of the LLVM Pass framework. Our tool was tested using the kernels of the `Polybench/GPU` benchmark suite with great results.

That same experimentation has lead us to consider the limitations our chosen performance prediction model, MWP-CWP. Recently, the author of [67] and [66] has suggested a GPU performance model relying on Little's law; it measures concurrency as a product of latency and throughput. This model considers both warp and instruction concurrency while previous models [3, 34, 58, 9] consider only warp concurrency. The author's analysis of those models

suggests their limitation is the significant underestimation of occupancy when arithmetic intensity (the number of arithmetic instructions per memory access) is intermediate. This is exactly the type of kernels on which KLARAPTOR underperforms. In future work we look to apply an improved performance prediction model in order to achieve even better results.

Bibliography

[1] CUDA runtime API: v10.0. NVIDIA Corporation, September 2018. http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf.

[2] CUDA C Best Practices Guide, v10.1.105, March 2019. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[3] CUDA C Programming Guide, v10.1.105, March 2019. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[4] Profiler User's Guide, v10.1.105, March 2019. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[5] Alfred V. Aho, Ravi. Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 3rd edition, 1999.

[7] Mohammadali Asadi, Alex Brandt, Changbo Chen, Svyatoslav Covanov, Farnam Mansouri, Davood Mohajerani, Robert Moir, Marc Moreno Maza, Linxiao Wang, Ning Xie, and Yuzhen Xie. Basic Polynomial Algebra Subprograms (BPAS), 2019. http://www.bpaslib.org.

[8] Benjamin Assarf, Ewgenij Gawrilow, Katrin Herr, Michael Joswig, Benjamin Lorenz, Andreas Paffenholz, and Thomas Rehn. Computing convex hulls and counting integer points with polymake. *Math. Program. Comput.*, 9(1):1–38, 2017.

[9] Sara Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for GPU architectures. PPoPP '10, pages 105–114. ACM, 2010.

[10] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

[11] Alexander I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.*, 19(4):769–779, 1994.

[12] Muthu Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *ICS 2008, Island of Kos, Greece, June 7-12, 2008*, pages 225–234, 2008.

[13] Sebastian Berndt, Klaus Jansen, and Kim-Manuel Klein. New bounds for the vertices of the integer hull. In Hung Viet Le and Valerie King, editors, *4th Symposium on Simplicity in Algorithms, SOSA 2021, Virtual Conference, January 11-12, 2021*, pages 25–36. SIAM, 2021.

[14] Alexander Brandt, Davood Mohajerani, Marc Moreno Maza, Jeeva Paudel, and Lin-Xiao Wang. KLARAPTOR: A tool for dynamically finding optimal kernel launch parameters targeting CUDA programs. *CoRR*, abs/1911.02373, 2019.

[15] Winfried Bruns, Bogdan Ichim, Tim Römer, Richard Sieg, and Christof Söger. Normaliz. algorithms for rational cones and affine monoids, 2010.

[16] Changbo Chen, Xiaohui Chen, Abdoul-Kader Keita, Marc Moreno Maza, and Ning Xie. MetaFork: A compilation framework for concurrency models targeting hardware accelerators and its application to the generation of parametric CUDA kernels. In *Proceedings of CASCON 2015*, pages 70–79, 2015.

[17] Vasek Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discret. Math.*, 4(4):305–337, 1973.

[18] Henri Cohen, Henry Cohen, and Henri Cohen. *A course in computational algebraic number theory*, volume 8. Springer-Verlag Berlin, 1993.

[19] William J. Cook, Mark Hartmann, Ravi Kannan, and Colin McDiarmid. On integer points in polyhedra. *Comb.*, 12(1):27–37, 1992.

[20] Robert Corless and Nicolas Fillion. *A graduate introduction to numerical methods*. Springer, 2013.

[21] Eugène Ehrhart. Sur un problème de géométrie diophantienne linéaire. i. *Journal für die reine und angewandte Mathematik*, 226:1–29, 1967.

[22] Eugène Ehrhart. Polynômes arithmétiques et méthode des polyèdres en combinatoire. *International Series of Numerical Mathematics, Birkhäuser Verlag, Basel*, 35, 1977.

[23] Matteo Frigo and Steven G. Johnson. FFTW: an adaptive software architecture for the FFT. In *IEEE,ICASSP '98*, pages 1381–1384, 1998.

[24] Keisuke Fukuda. cdd. c: C-implementation of the double description method for computing all vertices and extremal rays of a convex polyhedron given by a system of linear inequalities. *Department of Mathematics, Swiss Federal Institute of Technology, Lausanne, Switzerland*, 1993.

[25] Joseph D. Garvey and Tarek S. Abdelrahman. Automatic performance tuning of stencil computations on gpus. In *ICPP 2015*, pages 300–309, 2015.

[26] James E. Gentle, Wolfgang K. Härdle, and Yuichi Mori. *Handbook of computational statistics: concepts and methods*. Springer, 2012.

[27] Phillip B. Gibbons. A more practical PRAM model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168. ACM, 1989.

[28] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs *and* an algorithm for the mixed integer problem. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, pages 77–103. Springer, 2010.

[29] Scott Grauer-Gray and Louis-Noel Pouchet. Implementation of polybench codes GPU processing, 2012. http:://web.cse.ohio-state.edu/ pouchet.2/polybench.

[30] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Proc. InPar*, pages 1–10. IEEE, 2012.

[31] Sardar Anisul Haque, Marc Moreno Maza, and Ning Xie. A many-core machine model for designing algorithms with minimum parallelism overheads. In *Parallel Computing: On the Road to Exascale, Proc. of ParCo*, volume 27, pages 35–44. IOS Press, 2015.

[32] Warwick Harvey. Computing two-dimensional integer hulls. *SIAM J. Comput.*, 28(6):2285–2299, 1999.

[33] A. C. Hayes and David G. Larman. The vertices of the knapsack polytope. *Discret. Appl. Math.*, 6(2):135–138, 1983.

[34] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA 2009*, pages 152–163, 2009.

[35] Rui-Juan Jing and Marc Moreno Maza. Computing the integer points of a polyhedron, I: algorithm. In Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing - 19th International Workshop, CASC 2017, Beijing, China, September 18-22, 2017, Proceedings*, volume 10490 of *Lecture Notes in Computer Science*, pages 225–241. Springer, 2017.

[36] Rui-Juan Jing and Marc Moreno Maza. The z_polyhedra library in maple. In Jürgen Gerhard and Ilias S. Kotsireas, editors, *Maple in Mathematics Education and Research - Third Maple Conference, MC 2019, Waterloo, Ontario, Canada, October 15-17, 2019, Proceedings*, volume 1125 of *Communications in Computer and Information Science*, pages 132–144. Springer, 2019.

[37] Volker Kaibel and Marc E. Pfetsch. Computing the face lattice of a polytope from its vertex-facet incidences. *Comput. Geom.*, 23(3):281–290, 2002.

[38] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Trans. Archit. Code Optim.*, 9(4), 2013.

[39] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *(TOPLAS)*, 25(4):500–548, 2003.

[40] Jakub Kurzak, Yaohung Tsai, Mark Gates, Ahmad Abdelfattah, and Jack J. Dongarra. Massively parallel automated software tuning. In *ICPP 2019, Kyoto, Japan, 2019*, pages 92:1–92:10, 2019.

[41] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[42] Robert V. Lim, Boyana Norris, and Allen D. Malony. Autotuning GPU kernels via static and predictive analysis. In *ICPP 2017*, pages 523–532, 2017.

[43] Yixun Liu, Eddy Z Zhang, and Xipeng Shen. A cross-input adaptive framework for GPU program optimizations. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009.

[44] Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. *J. Symb. Comput.*, 38(4):1273–1302, 2004.

[45] Lin Ma, Kunal Agrawal, and Roger D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Comp. Syst.*, 30:202–215, 2014.

[46] Marc Moreno Maza and Linxiao Wang. Computing the integer hull of convex polyhedral sets. In Franccois Boulier, Matthew England, Timur M. Sadykov, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing - 24th International Workshop, CASC 2022, Gebze, Turkey, August 22-26, 2022, Proceedings*, volume 13366 of *Lecture Notes in Computer Science*, pages 246–267. Springer, 2022.

[47] Xinxin Mei and Xiaowen Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans. Parallel Distrib. Syst.*, 28(1):72–86, 2017.

[48] Benoit Meister. *Stating and manipulating periodicity in the polytope model: Applications to program analysis and optimization*. PhD thesis, Strasbourg 1, 2004.

[49] Marc Moreno Maza and Linxiao Wang. On the pseudo-periodicity of the integer hull of parametric convex polygons. In Franccois Boulier, Matthew England, Timur M. Sadykov, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing - 23rd International Workshop, CASC 2021, Sochi, Russia, September 13-17, 2021, Proceedings*, volume 12865 of *Lecture Notes in Computer Science*, pages 252–271. Springer, 2021.

[50] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[51] Georg Pick. Geometrisches zur zahlenlehre. *Sitzenber. Lotos (Prague)*, 19:311–319, 1899.

[52] Maple polyhedralsets package, 2021.

[53] Markus Püschel, Jose M.F. Moura, Bryan Singer, Jianxin Xiong, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1):21–45, 2004.

[54] Katsuto Sato, Hiroyuki Takizawa, Kazuhiko Komatsu, and Hiroaki Kobayashi. Automatic tuning of CUDA execution parameters for stencil processing. In *Software Automatic Tuning, From Concepts to State-of-the-Art Results*. 2010.

[55] Alexander Schrijver. On cutting planes. *Combinatorics*, 79:291–296, 1980.

[56] Alexander Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.

[57] Rachid Seghir, Vincent Loechner, and Benoit Meister. Integer affine transformations of parametric $\mathbb{Z}$-polytopes and applications to loop nest optimization. *ACM Trans. Archit. Code Optim.*, 9(2):8:1–8:27, 2012.

[58] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard W. Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, 2012.

[59] Chenchen Song, Lee-Ping Wang, and Todd J. Martinez. Automated code engine for graphical processing units: Application to the effective core potential integrals and gradients. *J. Chem. Theory Comput.*, 12(1):92–106, 2015.

[60] Larry J. Stockmeyer and Uzi Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984.

[61] Marshall H. Stone. The generalized weierstrass approximation theorem. *Mathematics Magazine*, 21(5):237–254, 1948.

[62] Rekha R. Thomas. Integer programming: Algebraic methods. In Christodoulos A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization, Second Edition*, pages 1624–1634. Springer, 2009.

[63] Yuri Torres, Arturo González-Escribano, and Diego R. Llanos. uBench: exposing the impact of CUDA block geometry in terms of performance. *The Journal of Supercomputing*, 65(3):1150–1163, 2013.

[64] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007.

[65] SI Veselov and A Yu Chirkov. Some estimates for the number of vertices of integer polyhedra. *Journal of Applied and Industrial Mathematics*, 2(4):591–604, 2008.

[66] Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2016.

[67] Vasily Volkov. A microbenchmark to study GPU performance models. In *Proc. PPoPP*, pages 421–422, 2018.

[68] R. Clinton Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 IEEE/ACM Conference on Supercomputing*, 1998.

[69] Wikipedia contributors. Convex hull — Wikipedia, the free encyclopedia, 2021. [Online; accessed 1-July-2021].

[70] Wikipedia contributors. Rasterisation — Wikipedia, the free encyclopedia, 2021. [Online; accessed 1-July-2021].

[71] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE ISPASS*, pages 235–246, 2010.

[72] Hiroki Yanagisawa. A simple algorithm for lattice point counting in rational polygons, 2005.

Curriculum Vitae

Name: Linxiao Wang

Post-Secondary Education and Degrees:

University of Western Ontario
London, ON
2018 - 2022 Ph.D.

University of Western Ontario
London, ON
2016 - 2018 M.Sc.

Renmin University of China
Beijing, China
2011 - 2015 B.Eng.

Related Work Experience:

Teaching Assistant
University of Western Ontario
2017 - 2022

Research Assistant
Ontario Research Center for Computer Algebra
University of Western Ontario
2017 - 2022

Intern
Mitacs Accelerate Fellowship with Maplesoft Inc.
Nov. 2021 - Feb.2022, May. 2022 - Sept. 2022

Intern
IBM Center for Advanced Studies (CAS)
Summer 2018, Summer 2019

Research Assistant
Information Security Lab
Renmin University of China
2014 - 2015

Publications:

- Maza, M. M., & Wang, L. (2022). Computing the Integer Hull of Convex Polyhedral Sets. In International Workshop on Computer Algebra in Scientific Computing (pp. 246-267). Springer, Cham.
- Moreno Maza, M., & Wang, L. (2021, September). On the Pseudo-Periodicity of the Integer Hull of Parametric Convex Polygons. In International Workshop on Computer Algebra in Scientific Computing (pp. 252-271). Springer, Cham.
- Covanov, S., Mohajerani, D., Moreno Maza, M., & Wang, L. (2019, July). Big prime field FFT on multi-core processors. In Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation (pp. 106-113).
- Qin, B., Wang, L., Wang, Y., Wu, Q., Shi, W., & Liang, B. (2016). Versatile lightweight key distribution for big data privacy in vehicular ad hoc networks. Concurrency and Computation: Practice and Experience, 28(10), 2920-2939.
- Qin, B., Wang, L., Wang, Y., Wu, Q., Shi, W., & Liang, B. (2014, October). Efficient sub-/inter-group key distribution for ad hoc networks. In International Conference on Network and System Security (pp. 448-461). Springer, Cham.

Software

- Basic Polynomial Algebra Subprograms (BPAS). M. Asadi, A. Brandt, C. Chen, S. Covanov, M. Kazaemi, F. Mansouri, D. Mohajerani, R.H.C. Moir, M. Moreno Maza, D. Talaashrafi, L. Wang, N. Xie, Y. Xie. 2022. http://bpaslib.org.
- The PolyhedralSets:-IntegerHull command. J. Gerhard, M. Moreno Maza, L. Wang. Shipped with Maple (Maplesoft), since Maple 2022.
- KLARAPTOR: A Tool for Dynamically Finding Optimal Kernel Launch Parameters Targeting CUDA Programs. A. Brandt, D. Mohajerani, M. Moreno Maza, L. Wang. 2019. http://github.com/orcca-uwo/KLARAPTOR.