Efficient Code Generation for Hardware Accelerators by Refining Partially Specified Implementations

Ulysse Beaugnon

PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École Normale Supérieure

Efficient Code Generation for Hardware Accelerators by Refining Partially Specified Implementation

Soutenue par
Ulysse BEAUGNON

Le 10 Juin 2019

Ecole doctorale n° 386
Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Composition du jury :

Francesco, ZAPPA NARDELLI Directeur de Recherche, INRIA	*Président*
Rastislav, BODIK Professeur, University of Washington	*Rapporteur*
Christophe, DUBACH Professeur, University of Edinburgh	*Rapporteur*
Anton, LOKHMOTOV CEO, Dividiti	*Examinateur*
Jacques, PIENAAR Ingenieur, Google	*Examinateur*
Albert, COHEN Chercheur, Google & ENS	*Directeur*
Marc, POUZET Professeur, École Normale Supérieure	*Directeur*

1794
ENS

Abstract

Software programmable hardware accelerators, such as Graphical Processing Units (GPUs), are specialized processors designed to perform specific tasks more efficiently than general purpose processors. They trade off generality against specialized data paths and massive parallelism, providing a raw processing power that is orders of magnitude higher than for contemporary multicore CPUs.

Unfortunately, finding an efficient implementation for a function on an hardware accelerator is a complex problem. It requires making careful decisions for mapping computations to the appropriate levels of parallelism and for expliciting data movements across the different memory spaces, in addition to choosing amongst to the many possible thread-local optimizations. While the set of possible optimizations is usually well known, complex interactions between them make it hard to find a global optimum.

Indeed, anticipating downstream transformations and deriving profitability information from intermediate compilation steps is a challenge. Transformations may not commute and some optimization opportunities may only become available after applying so-called "enabling" transformations. Conversely, some transformations may hinder further optimizations. As a result, the production of highly tuned implementations remains a critical challenge to achieve competitive performance.

This dissertation introduces the concept of *candidate* to formally define, represent and explore spaces of possible implementations. A candidate is a partially specified implementation with some decisions fixed while others are left open. It represents a whole set of possible implementations of the same function.

Candidates expose all potential decisions upfront and ensure they are commutative. Taking a decision always restricts the set of possible implementations. This defines a well-behaved optimization space; in particular, it allows search algorithms looking for the best implementations to make the most performance-impacting decisions first and to have a global knowledge of wich optimizations may feature in implementations. We provide a framework that automatically generate code to represents and manipulates candidates, from a declarative description of available choices and their interaction. This description is independent of the function to implement.

We instantiate our concept of candidate to generate efficient code for linear algebra functions on GPUs. This shows our approach is expressive enough to model interacting decisions with a fundamental impact on the structure of the generated code, including compositions of strip-mining, loop fusion, loop interchange, unrolling, vectorization, parallelization, and orchestration of data movements across the memory hierarchy.

We develop a model capable of computing a lower bound for the execution time of any implementation that derives from a candidate. We show that this model provides actionable information, even after taking only a few decisions, and that it enables pruning the implementation space, reducing its size by several orders of magnitude.

We propose a simple search algorithm to illustrate our approach. It combines the lower bound performance model and actual evaluation on the hardware with statistical exploration to drive the search towards the most efficient implementations. Our experiments show that it generates code that is competitive with hand-tuned libraries for linear algebra functions on GPUs. They also demonstrate that taking the most important decisions first helps finding better implementations faster, thus showing how the concept of candidate empowers search algorithms.

Acknowledgements

Many people helped me along the path leading to this dissertation. I am dedicating the next few paragraphs to thank them. First and foremost, I am grateful for my PhD advisors, Albert and Marc, who put their expertise to the service of my scientific and human development. In particular, I would like to honor the trust Albert put in me and the energy he gave for my ideas to succeed. And to honor Marc's thoroughness and the life he injects into the Parkas team, be it with the ski, the brioches or the rest.

This dissertation would not have been possible without Jacques, who helped me develop the initial idea and create the first prototype. Thank you for your guidance during that time. I look forward to working with you once again. The role played by Basile, Nicolas and Andi was equally crucial. They poured their ideas and their technical knowledge into my prototype to turn it into something that actually works. The careful proofreading of this dissertation by Basile and Andy was also essential. Thank you for your patience and your dedication. I would also like to thank the members of my PhD jury, especially the reviewers for reading my dissertation and providing constructive feedback.

The years I spent working in the Parkas team are full of happy memories. Thank you Tim, Francesco, Paul, Adrien, Guillaume, Nath, Guillaume, Lélio, Chandan, Adila and all the others. A special mention for Tim and his legendary helpfulness.

I made my first steps into the world of compilers during an internship with Anton. Obviously, I was convinced. Thank you for your kind welcome and your guidance.

Finally, thank you Florian, Yoann, Éric, Particia for your support along these years and Anaël for all the courage you gave me.

Contents

Chapter 1

Introduction

Hardware accelerators are specialized processors that are orders of magnitude more efficient than traditional processors for some classes of computation. This dissertation addreses the generation of optimized code for such specialized hardware. However, starting from a high level implementation of a computationally intensive algorithm, finding the right combination of code transformations to fully exploit this computing power is a complex problem. The main limitation of existing approaches is that they make it hard to anticipate downstream transformations and to extract profitability information from early compilation steps.

The question this dissertation addresses is: *Which combination of implementation decisions does lead to the fastest code, given a set of available optimizations?*. The originality of our approach is that we expose the compilation process as a progressive refinement of a *candidate implementation*, that represents a whole set of potential implementations.

Internally, a candidate implementation (candidate for short) is a partially specified implementation, with some implementation choices fixed while others are left open. It explicitly lists all potential implementation decisions, and makes sure they commute. It thus precisely defines potential downstream transformations, which in turn allows for deriving pertinent information from candidates early in the compilation process.

In this dissretation, we first introduce and formalize the concept of candidate. We then apply it to linear algebra on Graphical Processing Units (GPUs) and show how this concept helps finding better implementations faster. We develop a performance model that extracts actionable information from candidates, even when they contain several open choices. We also develop a simple search algorithm to illustrate our claims on the benefits of candidates.

Introduction Outline. Section 1.1 gives some background on hardware accelerators, Sections 1.2 and 1.3 respectively lay out our motivations and the high-level ideas behind our approach, while Sections 1.4 and 1.5 detail our key contributions and the organization of this thesis.

1.1 Hardware Accelerators

Software programmable hardware accelerators, such as GPUs are specialized processors designed to perform some specific tasks more efficiently than what is possible with general purpose processors. They trade generality against specialized data paths and massive parallelism, providing a raw processing power that is orders of magnitude higher than the performance of contemporary multicore CPUs. In some domains, adequately taking advantage of this power can be game-changing, allowing computations that would otherwise be too time or energy consuming. For instance, the recent success of deep learning methods can largely be attributed to the performance of GPU-accelerated libraries [1, 2].

Hardware accelerators perform small computation-intensive functions, called *kernels*. Unfortunately, writing or generating efficient code implementing a kernel is a complex problem. Hardware accelerators usually offer multiple levels of parallelism, with different trade-offs between the number of threads, the cost of communication and synchronization and the coherency of memory accesses. They also often feature domain-specific instructions and a complex memory hierarchy, with multiple caches and scratchpad memories of different sizes and properties.

Finding an efficient implementation requires making careful decisions for mapping computations to the appropriate levels of parallelism, for explicitly moving data movements across the different memory spaces and for exploiting domain-specific instructions, in addition to choosing amongst to the many possible thread-local optimizations. While the set of potential optimizations is usually well known, complex interactions between them make it hard to find the best global implementation. Applying one transformation can alter the low-level structure of the code and can change the pressure on constrained resources (e.g. the memory available or the number of threads that can execute in parallel) , which in turn can render other decisions invalid or inefficient. These interactions are critical for performance; for instance, parallelizing too many loops is harmful if the threads end up doing too few computations.

Moreover, the best combination of transformations depends on the kernel to implement, the size of the input data and the targeted accelerator. Different computations and input sizes may alter the trade-offs between bottlenecks. Implementations may not even be portable across accelerators implementing the same hardware abstraction. For example, different families of GPUs have memory spaces of different sizes and properties and the code for one GPU may not run on another [3]. It is thus difficult to reuse implementations or compiler heuristics across kernels, input sizes and targets. Writing optimized implementations by hand in infeasible when considering more that a few kernels and a few input sizes. However, automatically generating efficient implementations is also not trivial, as the following section explains.

1.2 Challenges of Code Generation for Hardware Accelerators

The problem of picking a good combination of implementation decisions in an interaction-heavy optimization space is not specific to hardware accelerators and occurs on any architecture complex enough that exhaustive enumeration is impossible. The literature already proposes multiple approaches to represent and explore the space of potential implementations, such as SPIRAL [4] for DSP codes, LGen [5] for basic linear algebra, or LIFT [6] for more general computations expressed using combinators.

These approaches all rely on rewrite rules, transforming and incrementally lowering an intermediate representation of the program. They start from an unoptimized implementation and iteratively replace expression patterns with purportedly more efficient ones, or code closer to the target instruction set architecture (ISA) and model of parallelism. A well-defined set of rewrite rules allows for the construction of a search tree, whose nodes are implementations and edges represent the application of rewrite rules. The compiler, or an expert programmer, can then navigate through the tree looking for the best implementation using domain-specific knowledge, hand-tuned heuristics or statistical exploration. Exhaustive evaluation of all possible implementations is usually infeasible. The implementation space grows exponentially with the number of decisions. Even small functions can have billions of possible implementations and just enumerating them, without evaluation, can already be too costly for most applications.

We argue that existing approaches make it hard to anticipate possible downstream transformations and to derive profitability information from intermediate compilation steps. While the list of directly applicable transformations at intermediate steps is usually well known, further transformations may only become available after applying so-called “enabling” transformations. For example, the contraction and promotion of an array into registers may only happen after fusing the producer and consumer loops of the array. Conversely, some transformations may hinder further optimizations. For example, interchanging the nesting order of two loops may forbid loop fusion. Moreover, transformations may not commute. As a result, the production and the specialization to a given target or problem size of highly tuned kernel implementations remains a critical challenge. Consequently, programmers often have to rely on proprietary libraries provided by hardware vendors to achieve competitive performance. Libraries are only available for a limited set of functions and are usually only optimized for a few problem sizes.

1.3 Our Solution: Partially Specified Implementations

This dissertation addresses the problem of anticipating downstream transformations and deriving profitability information from intermediate compilation steps. It exposes the compilation process of a kernel as the progressive refinement of a candidate. A candidate is a partially specified implementation where

some implementation choices are fixed, while others are left open. It represents a whole set of implementations of the same kernel. For example, a candidate may expose loop transformation opportunities, instruction selection, data mapping alternatives and memory transfer optimizations. This representation is compact and efficient to manipulate as it does not individually list each possible implementation. The compilation process starts with all choices open and iteratively specifies them until reaching a fully specified implementation.

Driving the Search Towards Efficient Implementations. The key benefit of candidates compared to existing approaches is that they expose all potential decisions upfront and ensure that they are commutative. Making a decision always restricts the set of possible implementations. This defines a well-behaved optimization space; in particular, it allows search algorithms looking for the best implementations to make the most performance-impacting decisions first and expert programmers to manually force any decision. It also precisely defines which transformations might be present in the final implementation from intermediate compilation steps, allowing for deriving profitability information before fixing all choices.

Our approach is not tied to a particular framework to express the objective function and to drive the search. It enables solutions combining actual evaluations on the hardware, performance models, heuristics and statistical approaches. This is critical as solvers cannot encode the full complexity of the hardware.

Candidates Representation. Not all combinations of decisions are valid. Candidates define a Constraint Satisfaction Problem (CSP) to model interactions between decisions, respect the semantics of the kernel and enforce hardware limitations. Since manually writing code to enforce constraints is a hard and error-prone process, we propose a Domain Specific Language (DSL) to describe implementation choices and express constraints between them. A compiler then automatically generates code to represent and manipulate candidates from the DSL. The DSL is generic, as it is not tied to a particular program representation. It enables us to iterate on different implementation space representations faster and makes it easier for external users to use our approach for different domains of application or hardware targets.

1.4 Key Contributions

This dissertation introduces the concept of candidate and studies how it enables search algorithms to better implementations faster. We illustrate our ideas by applying them to implement linear algebra kernels on GPUs. This last point is a contribution in itself, as it provides a new way to generate efficient code, competitive with or outperforming hand-tuned libraries.

Candidate Implementations. We propose an approach to formally define, represent and explore spaces of possible implementations using implementation candidates. We model the interactions between decisions as a constraint satisfaction problem. Candidates explicitly list open implementation choices and make sure they are commutative. This defines a well-behaved optimization space; in particular, the search algorithms can be organized to take the most performance-impacting decisions first. We provide a compiler generator that automatically generate code to represent and manipulate candidates, starting from a high-level declarative description.

Implementation Space Exploration. We study how the candidate approach empowers implementation space exploration techniques. We show that they allow precise information to be derived from sub-spaces of implementations, even before fixing all implementation choices. In particular, we develop a performance model that computes a lower bound of the execution time of a candidate. Formally, if c is a candidate, $x \in [\![X]\!]$ are the actual implementations that derive from c, $B(c)$ is the lower bound provided by the model and $T(x)$ is the execution time for $x \in [\![c]\!]$ on the target accelerator, then

$$\forall x \in [\![c]\!] \,.\, B(c) \leq T(x). \tag{1.1}$$

We combine this model with actual runs of implementations on the hardware in a branch-and-bound algorithm. This algorithm prunes entire regions of the implementation space at once without discarding the best implementation.

In order to illustrate our approach and show how it perform in practice, we propose a simple algorithm that searches for good implementations of a kernel. This algorithm combines the lower bound performance model and actual evaluation on the hardware with statistical exploration to drive the search towards the most efficient implementations. We show that both the commutativity of decisions and the ability to derive information from partially specified candidates are critical for guiding the exploration towards efficient implementations.

Candidates for Linear Algebra on GPUs. We instantiate our approach to optimize linear algebra kernels on GPUs. We provide a representation of candidates that exposes optimization decisions required to generate efficient code for this application domain and target hardware. We then provide and evaluate a lower bound performance model for this representation of candidates. This demonstrates that our approach is sufficiently expressive to represent complex decisions, with a fundamental impact on the structure of the generated code. This includes compositions of strip-mining, loop fusion, loop interchange, unrolling, vectorization, mapping to multiple levels parallelism, and orchestration of data movements across the memory hierarchy. We validate this work on the generation of optimized linear algebra kernels. We obtain code that is competitive with the performance of vendor-provided hand tuned libraries, and even outperforms them in some cases.

1.5 Organization

The rest of this thesis comprises a motivating chapter, that presents the ideas on a simple example, followed by four chapters that formalize our approach and apply it to GPUs. The last chapter presents future research directions and concludes the dissertation.

Chapter 2: Motivation: Partially Specified Schedules. The motivation chapter showcases our ideas on a simple problem: the scheduling of instructions within a basic block. It presents concrete questions and solutions that motivate following chapters. In particular, it states the problem we solve, explain how we can encode partially specified schedules and shows how we can use a performance model to prune whole classes of solutions at once when looking for the best implementation.

Chapter 3: Representing Candidate Implementations. This chapter formalizes the concept of candidate. It represents the set of potential implementations as the solutions of a constraint satisfaction problem. It then presents a language to define available implementation decisions and their interactions in a high level declarative domain specific language. Finally, it explains how to automatically generate code to represent and manipulate candidate implementations from the declarative description.

Chapter 4: Application: Linear Algebra on GPUs. This chapter instantiates the ideas of Chapter 3. It proposes an encoding of possible implementations for linear algebra kernels on GPUs. This encoding supports both thread-local and global implementation decisions, including compositions of strip-mining, loop fusion, loop interchange, unrolling, vectorization, mapping to multiple levels parallelism, and orchestrating data movements across the memory hierarchy.

Chapter 5: An Optimistic Performance Model of Candidates. This chapter presents an analytical model of a lower bound for the execution time of a candidate. It then instantiates the performance model for GPUs. The model heavily relies on the nature of candidates to provide exact information early in the compilations process.

Chapter 6: Scaling the Search with the Properties of Candidates. This chapter puts together the ideas developed in Chapters 3, 4 and 5 to efficiently explore the implementation space. The goal is to discover how to specify the open choices of a candidate in order to maximize performance. We define a hybrid approach that combines the lower bound performance model with a Monte Carlo Tree Search algorithm driving the search towards the best decisions.

Chapter 7: Conclusion. Finally, the last chapter concludes this thesis with a summary of our work and presents our research perspectives to improve the expressiveness of our implementation spaces and develop new search algorithms.

We have implemented the ideas developed in this thesis in a framework named *Temalon*. This framework, written in Rust, is available under the Apache 2.0 License at the following address.

`https://github.com/ulysseb/telamon`.

Chapter 2

Motivation: Partially Specified Schedules

The goal of this dissertation is to find combinations of implementation decisions for a kernel leading to efficient code. In the general case, this involves a wide variety of decisions, interacting with each other. This chapter introduces, on a simpler problem, the questions and the ideas developed in subsequent chapters. Its goal is to hint at the solutions and the formalism we later present, to help the reader understand the general direction of the dissertation. This chapter studies the problem of scheduling instructions within a basic block, that is, a block of straight-line code with a single entry point and a single exit point. To simplify things further, we also assume a simplistic hardware target evaluating instructions in order and on which we can easily evaluate a schedule.

Even if it only serves as an example here, the problem of scheduling basic blocks is an actual problem encountered by compilers. The number of potential schedules grows exponentially with the number of instructions, making exhaustive exploration of possible schedules impossible. We propose to reason about whole classes of schedules in order to tame the size of the search space. For this, we start from a partial order that we iteratively constrain until reaching a total order. Each intermediate step is a partially specified schedule from which we can extract information to drive the search towards the best solutions.

The idea to manipulate partially specified schedules matches the concept of candidate implementations, which is central in this dissertation. The treatment of the scheduling problem follows the ideas that we develop in the subsequent chapters of this thesis. In particular, it formulates the scheduling problem as a Constraint Satisfaction Problem (CSP). This formalism is crucial to represent partially specified solutions for more complex problems. It also builds a performance model that gives a lower bound for the execution time of any implementation derived from a partially specified schedule. This model suggests that partially specified implementations allow for the development of global heuristics that extract pertinent information from intermediate compilation steps.

This chapter is in the scope of superoptimization. This technique, pioneered by H. Massalin [7], optimizes a computation by trying every potential implementation in a class of program. In our case, we consider every permutation of the initial instructions. Superoptimization typically relies on a combination of empirical tests and a theorem prover to ensure generated implementations are correct a posteriori [8, 9]. Instead, we rely on constraint propagation to avoid considering any invalid implementation in the first place. This is crucial to later consider bigger and more complex implementation spaces, outside of the scope of superoptimization, without spending too much time finding valid implementations.

Outline. Section 2.1 gives a formal definition of the scheduling problem and discusses the difficulties encountered to find the best schedule. Section 2.2 presents some background on Constraint Satisfaction Problems. Section 2.3 leverages this background to reformulate the scheduling problem. It shows how we represent and manipulate partially specified schedules to search for a good solution. Then, Section 2.4 defines a performance model that extracts accurate information for partially specified schedules and explains how we use it to safely prune the schedule space. Finally, Section 2.5 summarizes the ideas of this chapter and explains how we develop them in the rest of this dissertation.

Note that this first chapter only serves to illustrate the problems we propose to solve and the solutions we provide in subsequent chapters. It does not claim to provide a better way to schedule instructions within a basic block nor does it provide any actual evaluation as it only considers an abstract, idealistic, hardware. Subsequent chapters develop these ideas on actual hardware with more complex implementation spaces.

2.1 Problem Statement: Scheduling a Basic Block

The principal question this dissertation answers is: *Which combination of implementation decisions lead to the fastest code, given the set of available optimizations?*. Here, we consider a simpler problem, where the kernel is composed of a straight-line sequence of instructions, without any loops or parallelism. We assume that the only way to act on performance is to reorder instructions. This corresponds to the problem of scheduling instructions within a basic block. We also assume a simplistic hardware, that executes instructions in parallel but issues them in order, at a maximum rate of one instruction per cycle.

2.1.1 Problem Formalization

Formally, let I be a set of instructions to schedule and $\prec$ be a partial order on I that indicates data dependencies. Let $l(a)$ be the latency of each instruction $a \in I$. Any total order $<$ that respects $\prec$ is a valid schedule. From a schedule $<$, we define the time $t(a) \in \mathbb{N}$ when an instruction $a \in I$ starts executing as the smallest integer such that t respects the following constraints.

- Instructions are issued in order.

$$\forall a, b \in I.\, a < b \implies t(a) \leq t(b) \tag{2.1}$$

- Only one instruction can be issued per cycle.

$$\forall a, b \in I.\, a \neq b \implies t(a) \neq t(b) \tag{2.2}$$

- Instructions wait on their data dependencies.

$$\forall a, b \in I.\, a \prec b \implies t(b) \geq t(a) + l(a) \tag{2.3}$$

Finally, we define the total execution time of the schedule $T_<$ as

$$T_< = \max_{a \in I} t(a) + l(a). \tag{2.4}$$

For a schedule, $T_<$ corresponds to the time when the last instruction of the schedule has finished executing. The goal is to find a valid schedule $<$ that minimizes $T_<$.

2.1.2 Finding the Best Schedule

The simplest approach to finding an optimal schedule is to evaluate them all. However, the number of valid schedules grows exponentially with the number of instructions. The time required to enumerate valid schedules quickly makes exhaustive enumeration intractable, even if evaluating a schedule was instantaneous.

The alternatives are either to use hand-tuned heuristics to build a single schedule or to use statistical algorithms that drive the exploration of the schedule space toward the best solutions while only evaluating a fraction of valid schedules. Hand-tuned heuristics offer acceptable performance, but are unable to adapt to the subtleties of each input program. Statistical approaches are more versatile as they learn from trials and errors but still suffer from the exponential nature of the problem. The expected number of required evaluations to find a near-optimal solution grows with the size of the search space.

To generate a good schedule, we propose to start from the partial order representing data dependencies and iteratively constrain it until reaching a total order. Each intermediate step is a partially specified schedule, from which we can extract information to drive the search towards the best solutions. A partial order represents all the total orders that derive from it. The search algorithm can thus reason about whole parts of the schedule space without enumerating them. We now introduce the concept of CSP that we later use to model partially specified schedules.

2.2 Background: Constraint Satisfaction Problems

A *Constraint Satisfaction Problem* (CSP) models a system as a finite set of variables that take values in a finite universe. Constraints between variables encode their relationships. A solution of the problem is an assignment of the variables that satisfies the constraints. CSPs are widely used to model complex combinatorial problems, for examples for scheduling, planning, vehicle routing, network management and bioinformatics [10].

Finding a solution to a CSP is an NP-hard problem. In particular, the Boolean satisfiability problem (SAT) translates into a CSP. *Constraint Satisfaction Systems* combine heuristics and combinatorial search to find solutions to CSPs. In our specific case, finding a solution to the problem is trivial. The CSP formalism allows to easily reason about and manipulate partial orders and later fully fledged partially specified implementations.

This section follows the notations and definitions presented in the thesis of G. Tack [11]. We first give a denotational semantics of CSPs. We then explain how to enforce constraints and how to search for a solution of a CSP.

2.2.1 Variables, Domains and Constraints

CSPs are defined with respect to a finite set of variables X and a finite set of values V. A solution of a CSP assigns a value to each variable. A constraint restricts the set of valid assignments. The following definition captures the concepts of assignments and constraints.

Definition 2.1 (Assignment and Constraint)**.** An *assignment* $a : X \to V$ is a mapping from variables to values. A *constraint* is a set of assignments $c \subseteq \mathcal{P}(X \to V)$. Any assignment $a \in c$ is a *solution* of c.

In practice, constraints are usually expressed as logic formulae that only involve a few variables. For $a : X \to V$, $x \in X$ and $v \in V$ we denote $a[v/x]$ the assignment a' where $a'(x) = v$ and $a'(y) = a(y)$ for all $y \neq x$. We define the *significant* variables of a constraint c, $\text{vars}(c)$, as the variables that have an impact on whether an assignment is in c.

$$\text{vars}(c) := \{x \in X \mid \exists v \in V.\, a \in c.\, a[v/x] \notin c\} \tag{2.5}$$

Next, we define *domains* that indicate the set of values each variable can take.

Definition 2.2 (Domain)**.** A *domain* is a function $d : X \to \mathcal{P}(V)$ that maps variables to set of values. The set of values $d(x)$ for a particular variable $x \in X$ is the *variable domain* of x. A domain represents a set of assignments (i.e. a constraint) defined as

$$\text{con}(d) := \{a \in X \to V \mid \forall x \in X.\, a(x) \in d(x)\}. \tag{2.6}$$

We say that an assignment $a \in \text{con}(d)$ is *licensed* by d. We call $\text{Dom} := X \to \mathcal{P}(V)$ the set of all domains.

While domains each represent a constraint $\text{con}(d)$, the converse is not true. Domains are a Cartesian representation that is limited to conjunction of unary constraints, that is constraints with a single significant variable. We define a subset order on domains as the subset order on the induced constraints:

$$\forall d_1, d_2 \in \text{Dom}.\, d_1 \subset d_2 \iff \text{con}(d_1) \subset \text{con}(d2). \tag{2.7}$$

When any variable domain $d(x)$ is empty, $\text{con}(d) = \emptyset$. In that case, we say the domain is *failed*. We say that a domain corresponding to a single assignment or a variable domain with a single value is *fixed*.

A CSP is composed of a domain that restricts the initial values variables can take and of constraints that specify the relations between variables.

Definition 2.3 (Constraint Satisfaction Problem)**.** A *Constraint Satisfaction Problem* is a pair $\langle d, C\rangle$ of a domain $d \in X \to \mathcal{P}(V)$ and a set of constraints $C \in \mathcal{P}(X \to V)$. A *solution* of $\langle d, C\rangle$ is an assignment licensed by d that respects all the constraints in C.

$$\text{sol}(\langle d, C\rangle) := \{a \in \text{con}(a) \mid \forall c \in C.\, a \in c\} \tag{2.8}$$

2.2.2 Constraint Propagation and Search

The basis of a constraint satisfaction system is a search procedure that instantiates variables with values from their domain. Enforcing constraints directly from their extensional definition is infeasible, as it would require to enumerate all the assignments licensed by the domain. Instead, constraint satisfaction systems employ *propagators* that decide if an assignment is valid and prune the domain to rule out invalid assignments.

Definition 2.4 (Propagator)**.** A *propagator* is a function $p : Dom \to Dom$ such that:

- p is contracting: $p(d) \subseteq d$ for any domain $d \in Dom$, and
- p is sound: for any domain $d \in Dom$ and assignment $a \in d$, $p(\{a\}) \subseteq p(d)$.

We say that a propagator $p : Dom \to Dom$ *prunes* a domain $d \in Dom$ if $p(d)$ is strictly included in d, that is $p(d) \subset d$. The propagator p induces a constraint c_p defined by the set of assignment not pruned by p.

$$c_p := \{a \in X \to V \mid p(\{a\}) = \{a\}\} \tag{2.9}$$

Soundness is a weak form of monotonicity. It expresses that the pruning decisions realized by a propagator are consistent. A propagator never prunes assignments that satisfy its induced constraint. Propagators are the operational equivalent of constraints. Each propagator induces a single constraint, that is exactly realized by its pruning procedure. We now define *propagation problems* to realize CSPs using propagators.

Definition 2.5 (Constraint Propagation Problem)**.** A *propagation problem* is a pair $\langle d, P\rangle$ of a domain $d \in Dom$ and a set of propagators $P \subseteq Dom \to Dom$. The *induced constraint satisfaction problem* of a propagation problem $\langle d, P\rangle$, is the CSP $\langle d, \{c_p \mid p \in P\}\rangle$. The *solutions* of a propagation problem are the solutions of the induced CSP:

$$\mathrm{sol}(\langle d, P\rangle) := \mathrm{sol}(\langle d, \{c_p \mid p \in P\}\rangle) \tag{2.10}$$

The solutions of a propagation problem $\langle d, P\rangle$, are the assignments licensed by d that are not pruned by any propagator. This is obtained by just applying the definition of the induced constraints to the definition of the solution of a propagation problem.

$$\mathrm{sol}(\langle d, P\rangle) = \{a \in \mathrm{con}(d) \mid \forall p \in P.\, p(\{a\})) = \{a\}\} \tag{2.11}$$

In practice, the constraint satisfaction system automatically generates the propagators from the constraints definition.

The search procedure for a propagation problem $\langle d, P\rangle$ works as follows. It starts from the initial domain and instantiates the value of a variable by restricting its domain to a single element. It then prunes the domain with propagators until reaching a fix point. The resulting domain falls into three categories:

- If each variable domain contains a single value, then the search is finished and the domain represents a single valid assignment.
- If any variable domain is empty, then the domain is failed and the search procedure backtracks.
- Otherwise, some variable domain still contains multiple values. In that case, the search procedure calls itself recursively on the pruned domain.

Running all propagators at each step of the search procedure would be too costly. Therefore, constraint satisfaction systems try to avoid running a propagator when they know it will not prune the domain. In particular, they only run a propagator if the domain of any significant variable of the induced constraint has changed.

Two propagators inducing the same constraint may prune domains differently. Definition 2.4 only requires that they enforce the induced constraints on fixed domains that represent a single assignment. Propagators may keep invalid values in other domains as long as they are sound. An optimal propagator

p_{opt} inducing a constraint c only keeps the values of a domain d that appear in at least one solution of $\langle d, \{c\}\rangle$.

$$\forall x \in X.\ p_{opt}(d)(x) = \{v \in d(x) \mid \exists a \in c \cap \mathrm{con}(d).\ a(x) = (v)\} \tag{2.12}$$

Using optimal propagators is impractical in the general case. They may require to enumerate all possible assignments. Moreover, the fix point of two optimal propagators inducing constraints c_1 and c_2 might be weaker than the optimal propagator for $c_1 \cap c_2$. Considering constraints independently instead of as a single conjunction alters propagation strength but makes it possible to generate faster propagators and reduces the number of significant variables. Constraint satisfaction systems must thus consider the trade-off between strong and fast propagators. Since propagators are exact on fixed domains, using weaker propagators does not alter correctness. Backtracking handles the cases that constraint propagation misses.

We now go back to the problem of scheduling a basic block and leverage the CSP formalism we just presented to model partially specified schedules.

2.3 Partially Specified Schedule

One way to specify a valid schedule is to start from the partial order representing data dependencies and to iteratively refine it by specifying the order between pairs of instruction, until the order is total. With this approach, a partial order is a partially specified schedule that represents all the total orders that derive from it. We model this problem as a CSP whose variables are the pairwise ordering between instructions.

2.3.1 Formalization as a Constraint Satisfaction Problem

Formally, a valid schedule is an assignment of a function *order* that maps pairs of distinct instructions to the values *before* and *after*,

$$order : I \times I \to \{before, after\} \tag{2.13}$$

and that respects the following constraints.

- The order enforces data dependencies.

$$\forall a, b \in I.\ a \prec b \implies order\,(a, b) = before \tag{2.14}$$

- The order is antisymmetric.

$$\forall a, b \in I.\ order\,(a, b) = before \iff order\,(b, a) = after \tag{2.15}$$

- The order is transitive.

$$\forall a, b, c \in I.\ order\,(a, b) = before \wedge order\,(b, c) = before \implies order\,(a, c) = before \tag{2.16}$$

This specifies a constraint satisfaction problem, with *order* being an uninterpreted function that defines the CSP variables.

$$X := \{order(a, b) \mid a, b \in I\} \quad V := \{before, after\} \tag{2.17}$$

Formulae (2.15) and (2.16) respectively translate into one constraint for each pair and each triplet of instructions, while (2.14) defines the initial domain d.

$$\begin{array}{rccl} d: & X & \to & \mathcal{P}\left(\{before, after\}\right) \\ & order(a, b) & \mapsto & \begin{cases} \{before\} & \text{if } a \prec b \\ \{after\} & \text{if } b \prec a \\ \{before, after\} & \text{otherwise} \end{cases} \end{array} \tag{2.18}$$

Following the search procedure sketched in Section 2.2.2, the search for a schedule starts from the initial domain and iteratively specifies the value of the order between pairs of instructions. In this specific case, we can generate propagators that are strong enough to prevent backtracking. After propagation, *order* represents a partial order.

2.3.2 Schedule Search Tree

The traditional goal of CSPs is to find solutions to a problem. In our case, finding a solution is trivial. A topological sort of the instructions following data dependencies would suffice. Instead we are looking for the best solution we can obtain in a limited amount of time. Ideally, we would generate all possible schedules and return the best one. For this, we would build a binary tree where each node is a partial order and the child of a node are the possible orderings between a pair of unordered instructions. This tree corresponds to the possible instantiations of the pairwise orderings by the CSP search procedure, with variables instantiated in a fixed order.

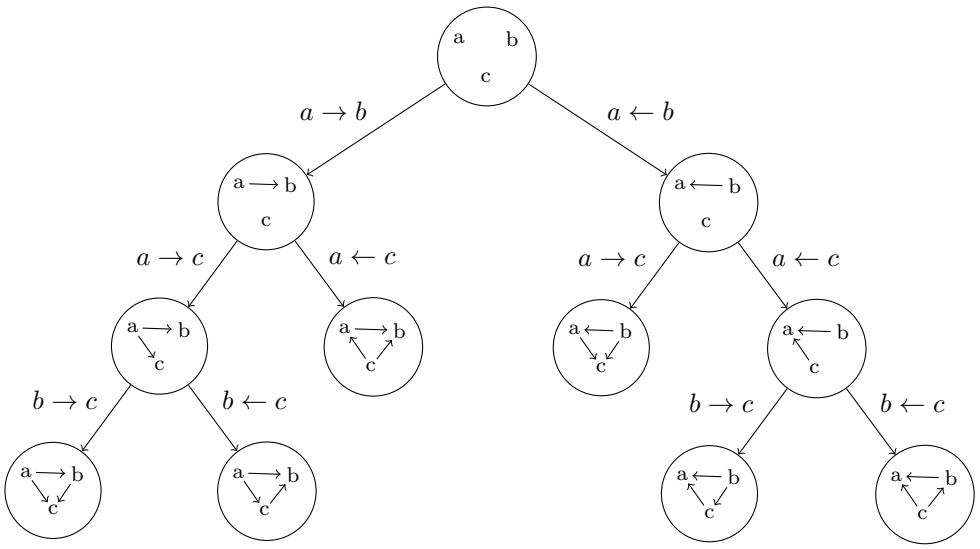


Figure 2.1: A Schedule Search Tree for Basic Blocks With Three Instructions

To build the tree, we start from the root representing the full domain. The two children of a node are the domains obtained by respectively assigning *before* and *after* to the next unspecified variable and propagating constraints. The leaves of the tree are fully specified schedules for which we can compute an execution time, while the nodes of the tree represent the set of schedules in their sub-tree. Figure 2.1 presents a schedule search tree for a basic block with three instructions (a, b and c) and no data dependencies. Each level of the tree specifies the order between two instructions, with two possible outcomes.

Building the entire tree would, of course, be infeasible when the number of instructions grows. Thus, we only build the parts of the tree in which we expect to find the best implementations. The idea is that we can reason on partially specified schedules to detect uninteresting nodes early in the search tree, thus avoiding building large parts of the tree.

2.4 Optimistic Model of Partially Specified Schedules

We now present a performance model of partially specified schedules to prune the search tree. The performance model gives a lower bound $B(d)$ on the execution time of any schedule licensed by the domain $d \in Dom$. If S is the set of all valid schedules, then:

$$\forall s \in S.\ s \in \text{con}(d) \implies T_s \geq B(d) \tag{2.19}$$

From the point of view of the search tree, the bound obtained for an internal node is a lower bound for the execution time of all the schedules in the subtree rooted in the node.

The performance model relies on a graph that represents timestamp constraints between instructions. Given a domain d, the vertices are the set of instructions I and the edges between any two instructions $a, b \in I$ are defined as follows.

- If a is scheduled before b, that is $d(order(a, b)) = \{before\}$, then E contains an edge from a to b with weight 1. This corresponds to the constraint imposed by equation (2.1)

- If b depends on a ($a \prec b$), then E contains an edge from a to b with weight $l(a)$. This corresponds to the constraint imposed by equation (2.3).

We compute $B(d)$ as the weight of the longest path in the graph plus the latency of the last instruction. This corresponds to equation (2.4). Constraints (2.15–2.16) ensure there are no loops in the graph so the longest path weight is well-defined. When the domain is fixed, the bound is equal to the execution time. Otherwise, the graph only contains edges that appear in the graph of every schedule licensed by the domain and the longest path in the graph is a lower bound of possible execution times.

This bound allows us to prune the search tree using a branch and bound strategy. The search algorithm discards branches for which the bound B is bigger than the current best candidate. Because this model works on partially specified implementations, we can prune whole parts of the schedule space at once. We also ensure than we never prune any optimal schedules.

2.5 Summary

This chapter presents a solution to the problem of scheduling instructions within a basic block. For this it first formalized the problem and gave some background on CSPs. It them reformulated the problem of scheduling instructions as a CSP. Intermediate steps of the CSP resolution represent partial orders, while solution of the CSP represent fully specifed implementations of the basic block. In practice, scheduling instructions in straight-line code is only one of the many aspects of choosing an implementation for a kernel.

Chapter 3 builds on the ideas of representing sets of potential implementation using the CSP formalism. It presents a generic framework that defines a CSP for each kernel from a declarative description of implementation choices and constraints. Chapter 4 then apply this framework to represent implementation spaces of linear algebra kernels on GPUs, supporting complex transformations. It shows that while the formalism of this chapter may seem overly complex, it is critical when considering several classes of optimizations, interacting with each other.

This chapter also proposes to build a performance model to prune parts of the schedule space. Chapter 5 extends this idea to compute a lower bound for the execution time of any implementation that derives from a partially specified implementation. We show that such models can extract pertinent information early in the compilation process, greatly improving the performance of the search for efficient implementations.

Chapter 3

Representing Candidate Implementations

This chapter introduces and formalizes the concept of *candidates*, that are partially specified implementations, with some implementation decisions left open while others are fixed. A candidate thus represents a whole set of potential implementations of the same kernel, up to semantics-preserving implementation decisions. The compilation process starts with all choices open and iteratively specifies them until reaching a fully specified implementation.

Not all combinations of decisions are valid. Candidates define a Constraint Satisfaction Problem (CSP) to model interactions between decisions, respect the semantics of the kernel and enforce hardware limitations. Because manually writing the code to enforce constraints is a hard and error-prone process, we propose a domain specific language (DSL) to describe implementation choices and express constraints between them. This defines a *decision space* that we can instantiate on different kernel to represent candidates.

A dedicated compiler transforms a specification using this DSL into code that we then use to construct the search tree. This consists in generating a data structure that holds the list of available decisions, the code that builds this data structure from the kernel description and the code that propagates constraints on decisions. The DSL and its compiler make our approach independent of a particular encoding of transformations or a of a particular kernel representation. It enables us to easily try out different decision spaces and allows external users to define their own decision space for different application domains.

Contributions. This chapter focuses on the formalism and the generic framework to define candidate implementations, while Chapter 4 explains how we use our framework to build a decision space for linear algebra kernels. Specifically, this chapter makes the following contributions.

- It introduces the concept of candidate implementation.
- It presents a formalism to define candidates independently of a particular kernel.
- It presents a declarative language and an associated compiler that implements the formalism. Together, they make it easier to extend the implementation space with new decisions or to apply our approach to other domains of applications.

Outline. Section 3.1 gives a definition of candidate implementations and explains how we expose potential decisions. Section 3.2 then explains how we can specify the decisions and constraints that make up candidates independently of kernel instances. This section introduces the concept of *decision space*, that lists generic choices and constraints that can be instantiated on any kernel. It also introduces the concept of *lowering*, that allows us to lower some high-level constructs in the kernel definition when certain conditions are met. Section 3.3 defines a declarative language that defines decision spaces. Section 3.4 illustrate our approach by applying our formalism to the problem of Chapter 2. Then, Section 3.5 explains how we generate code from our domain specific language to represent and manipulate candidates. Finally, Sections 3.6 and 3.7 compares our work to existing approaches and discuss how our framework helps finding efficient implementations.

3.1 Candidate Implementations

We represent the problem of choosing among possible implementations of a computational kernel as a CSP. Each variable of the CSP materializes an implementation choice. The initial domain of such a variable contains one value for each possible outcome of the choice. Not all combinations of decisions are valid, hence constraints ensure decisions are coherent. An implementation is a solution of the CSP, that is, an assignment from choices to outcomes that satisfies the constraints. This section assumes that the encoding of the implementation space into a CSP $\langle d_K, C_K \rangle$ for any kernel K is given. Sections 3.2 and 3.4 and Chapter 4 later explain how we build this encoding from a kernel-independent description.

3.1.1 Representation of Potential Implementations.

Following the motivating example of Chapter 2, we use domains to represent and manipulate sets of possible implementations.

Definition 3.1 (Candidate and Implementation)**.** A *candidate* is a pair (K, d), composed of a kernel K to implement and a domain d for the CSP $\langle d_K, C_K \rangle$. The domain d is thus a more constrained version of the initial domain d_K of the CSP: $d \subseteq d_K$. An *implementation* of (K, d) is a solution of $\langle d, C_K \rangle$. The *implementation space* $[\![(K, d)]\!]$ of a (K, d) is the set of all solutions of $\langle d, C_K \rangle$.

The domain of a candidate lists possible decisions for each choice. A search algorithm enforces a decision by restricting the domain of the corresponding choice. Constraints may forbid some combinations of decisions. Following the formalism of CSPs, we use propagators that implement the constraints to prune the domains from invalid values. To maximize the information about the validity of decisions, we only consider candidates with domains on which propagators have reached a fixed point. We thus run propagators at the creation and after each modification of the domain. Even then, some invalid value may remain undetected until further choices are specified. In fact, it might be possible that a candidate does not represent any valid implementation at all if one of the variable domains only contains invalid values. CSP solvers solve this problem by backtracking.

In our case, Section 4.5.2 shows that the constraints we use are simple enough constraints to generate accurate propagators that almost never miss invalid values, and thus accurately represent possible decisions. Our search strategy, which we present in Chapter 6, already backtracks when it encounters some candidates that only lead to inefficient implementations. The impact of backtracking due to suboptimal propagators is negligible compared to the number of backtracking due to the search strategy. The definition of propagators (Section 2.2) ensures no invalid value remains in fixed domains, thus ensuring that fully specified candidates are valid.

3.1.2 Interface With Search Algorithms

The objective of a search algorithm is to find an assignment of variables that satisfies the constraints and minimizes the execution time of the kernel on a particular input. Algorithms operate on the domain of variables while the kernel representation remains mostly untouched. From the point of view of a search algorithm, a candidate is thus a partially instantiated decisions vector, with the domain of each choice. The only actions the algorithm can perform are the following.

1. It can restrict the domain of a variable in the decision vector. This automatically triggers propagators to recursively prune incompatible values from other variable domains.

2. It can copy the decision vector, to branch the search and try different decisions. It does not need to copy the constraints that define the CSP, as they solely depend on the kernel instance which is unchanged by this operation.

A search algorithm typically combines these actions to build a tree whose nodes are candidates. The root is the fully unrestricted candidate (K, d_K) and each node is a strictly more constrained version of its parent. The leaves of the tree are fixed candidates, corresponding to actual implementations, for which the algorithm can generate code and measure its execution time.

3.1.3 Key Benefits of Candidates

This approach based on candidates offers three advantages. First, domains define a well-behaved set of actions that search algorithms can use to generate implementations. Since it can only restrict the domain of choices, decisions commute and the set of potential decisions decreases after each action. This means that the search algorithm is aware of all potential actions upfront and can make the most performance-impacting decisions first.

Second, it allows for manipulating intermediate steps of the decision process. This is in opposition to black-box solvers, such as ILP or SMT solvers, that optimize for a single objective function. Such solvers cannot fully encode the complexity of the target architecture. On the opposite, our approach allows to mix actual evaluations on the hardware, performance models, heuristics and statistical approaches.

Finally, it allows for the definition of functions from candidates, that extract information from the domains and guide the search. For instance, a function could estimate the execution time of the best implementation that derives from a candidate, or compute a probability distribution on the decisions. Theses function reason about entire sets of candidates at once and have a global knowledge of upcoming decisions, giving them access to the potential characteristics of final implementations. A limitation is that these functions operate on the domain, which is a Cartesian over-approximation of valid assignments.

The definition of candidates we gave in this section assumes we have a CSP $\langle d_K, C_K \rangle$ for each kernel K. The next section introduces the concept of *decision space*, that describes d_K and C_K independently of the kernel K.

3.2 Decision Space Definition

As stated in Definition 3.1, candidates are composed of two parts: a representation of the kernel and a domain. The kernel representation is mostly constant through the search, although some decisions – such as the introduction of temporary variables – can interact with the representation in a limited way. This section explains how we derive the list of decisions that must appear in the domain and the associated constraints from a kernel instance. It presents a formalism to define a generic CSP that can be instantiated for any kernel instance.

This formalism interprets the kernel representation as a set of objects that respect different properties, such as *x is an instruction* or *x depends on y*. While objects vary from one kernel to the other, the list of properties remains the same, allowing for the description of the implementation space independently from the kernel instance. Our formalism describes generic decisions and constraints that we instantiate for every tuple of objects verifying given properties. For instance, the generic CSP may induce a choice, indicating if a loop is unrolled or not, for each object x respecting the property *x is a loop*.

We first explain how we specify the choices and constraints independently of the kernel representation in Section 3.2.1 and then how we allow extending the implementation space with new choices and constraints when lowering high level constructs Section 3.2.2.

3.2.1 Formalization of the CSP Definition

This section explains how we define the CSP $\langle d_K, C_K \rangle$ relative to a kernel K. For that, we first formalize the concept of *kernel*. We then define *generic choices* as choices that we instantiate for each kernel and a *decision space* as a collection of generic choices and constraints. Finally, we define the CSP associated to a kernel K as the instantiation of the generic choices and constraints for K.

Kernel Representation

Let Ω be a set of identifiers representing objects let Π be a set of identifiers called *properties* and let $a : \Pi \to \mathbb{N}$ denote their arity. Let Ω^* be the set of tuples of objects of any size. We define a *kernel instance* as a mapping of properties to tuples of objects.

Definition 3.2 (Kernel Instance)**.** A *kernel instance* is a function K mapping properties to the objects satisfying it:

$$\begin{aligned} K : \quad & \Pi \to \mathcal{P}(\Omega^*) \\ & p \mapsto K(p) \subseteq \Omega^{a(p)} \end{aligned}$$

Generic Choices

Let $\mathcal{D}$ be a set of *domains*, where each domain is a (distinct) set of identifiers. We define a *generic choice* as a choice in a given domain that is instantiated for all tuples of objects respecting a set of properties.

Definition 3.3 (Generic Choice)**.** A generic choice with arity k is a tuple $c : ((p_0, \dots p_{k-1}), D) \in \Pi^k \times \mathcal{D}$.

A generic choice $c = ((p_0, \dots, p_{k-1}), D) \in \Pi^* \times \mathcal{D}$ indicates a choice among the values of D for each tuple of distinct objects verifying properties $(p_0, \dots, p_{k-1})$. For instance, the choice could be how each loop should be implemented, with $k = 1$, $p_0 =$ *is a loop* and $D = \{\texttt{regular}, \texttt{unrolled}, \texttt{parallel}\}$. Similarly, the choice could be the pairwise ordering between pairs of instructions, with $k = 2$, $p_0 = p_1 =$ *is an instruction* and $D = \{\texttt{before}, \texttt{after}\}$.

Instantiated for a kernel K, a generic choice $c = ((p_0, \dots, p_{k-1}), D) \in \Pi^* \times \mathcal{D}$ defines an actual choice – that is a CSP variable – for each combination of object that respect the properties. That is for each tuple of distinct objects in $K(p_0) \times \cdots \times K(p_{k-1})$. In our examples, that would respectively be for each loop and for each pair of distinct instructions.

Decision Space

Next we define a *decision space* in order to represent sets of possible implementations. A decision space is the combination of a set of generic choices defining the list of open decisions and a set of formulae expressing constraints on the decisions. Constraints are universally quantified over objects so that they are independent of the kernel instance, similarly to generic choices.

Definition 3.4 (Decision Space)**.** A *Decision space* is the pair $\langle \mathcal{C}, F \rangle$ of a set $\mathcal{C} \subseteq \Pi^* \times \mathcal{D}$ of generic choices, and a set of first-order formulae F called *constraint formulae*. The alphabet for the constraint formulae contains one predicate symbol with arity $a(p)$ for each property $p \in \Pi$, and one function symbol with arity $\sum_{i<k} a(p_i)$ for each choice $c = ((p_0, \dots, p_{k-1}), D) \in \mathcal{C}$.

Note that a *decision space* represents the generic decisions and constraints that are independent of any kernel, while an *implementation space* represents the set of potential implementations of a kernel. An implementation space is thus a decision space instantiated for a kernel.

CSP Associated to a Kernel

A decision space $\langle \mathcal{C}, F \rangle$ defines a CSP for each kernel K. The initial domain is the domain that maps each variable defined by a choice $c = ((p_0, \dots, p_{k-1}), D)$ to the full domain D. Solutions are the assignments licensed by the decision vector that respect the constraints specified by F. The definition of *decision vector* below captures the domains of this CSP.

Definition 3.5 (Partially Instantiated Decision and Decision Vector)**.** A *partially instantiated decision* for a choice $c = ((p_0, \dots, p_{k-1}), D)$ and a kernel K is a function χ_c that assigns sets of possible values to a generic choice:

$$\chi_c : K(p_0) \times \cdots \times K(p_{k-1}) \mapsto \mathcal{P}(D).$$

A *decision vector* for a decision space $\langle \mathcal{C}, F \rangle$ and a kernel is vector with one partially instantiated decision χ_c for each choice $c \in \mathcal{C}$.

Partially instantiated decisions are partial in the sense that it restricts the possible values, but does not necessarily select one. They are the domains associated to each generic choice instantiated for a kernel while a decision vector represents the full domain. In practice, their representation stores the value of the applications of the χ_c to each of their input.

We are now ready to define the constraint satisfaction problem $\langle d_K, C_K \rangle$ associated to a kernel K that we mention in Definition 3.1.

Definition 3.6 (CSP Associated to a Kernel)**.** The CSP associated to a kernel K for the decision space $\langle \mathcal{C}, F \rangle$ is the pair $\langle d_K, C_K \rangle$ where:

- d_K is the decision vector that maps each variable defined by the generic choice $c = ((p_0, \dots, p_{k-1}), D) \in \mathcal{C}$ to the full domain D, and
- C_K is the assignments licensed by d_K for which all constraint formulae hold when interpreting the predicate symbol for property p with $K(p)$ and the function symbol for choice c with χ_c.

3.2.2 Lowering of High-Level Constructs

This formalism allows for adding objects to the kernel representation during the search. Indeed, extending the set of objects that respects properties adds decisions and constraints without invalidating pre-existing ones. This is useful to lower high-level constructs upon reaching specific decisions. For example, this allows for inserting the appropriate load and store instructions after deciding to allocate a temporary array in memory. Lowering a construct may only add new objects or add properties to existing ones.

It is important to note that it is always possible to express directly in the initial CSP the choices and constraints that the constructs lowering introduce. Indeed, if a construct lowering triggers when a condition p is satisfied, and introduce a choice c, we can add a value `ABSENT` to the initial domain of c and a constraint $\neg p \vee c = \texttt{ABSENT}$ so that we can expose c directly in the initial CSP. Similarly, we can add the term $\vee \neg p$ to the constraints the lowering introduces. This ensures that our claim that all decisions can be available upfront is respected.

The reason to rely on the lowering of high-level constructs to define the decision space are:

- to avoid cluttering the decision vector with decisions that will not affect the generated code, making constraint propagation more efficient;
- and intentionally delay some decisions to focus search algorithms on more impactful choices first.

An hybrid approach is to use construct lowering but to pre-compute the transitive closure of choices that we can add to the CSP and to expose them to search heuristics only.

To lower a high-level construct, we operate in four steps.

1. First, the CSP solver detects the lowering condition is met. We describe such conditions using first-order logic formulae that are independent of the kernel instance, similar to the ones in F.
2. Then, the solver copies the kernel representation and adds objects to the sets representing properties.
3. Then, the solver extends the decision vector to accommodate the new variables induced by the new objects.
4. Finally, the solver enforces the new constraints obtained by evaluating the properties $p \in \Pi$ to the new objects in F.

This section gave of formal definition of the concept of *decision space*, that describes the choices and constraints that make up a candidate, independently of kernels. The following section explains how we specify decision spaces in practice.

3.3 Candidate Representation Description Language

This section presents a domain specific language (DSL) that implements the formalism presented in Section 3.2. It declaratively defines a decision space. In particular, it specifies how to interact with the kernel and defines generic choices, constraints and lowering conditions.

This language serves two distinct purposes. First, it precisely define the decision space in a concise way. This is essential to understand and debug the constraints. Second, it simplifies changing the encoding of the decision space. This makes it possible to try out different approaches or to build an decision space for a different domain with low effort. Indeed, we have implemented a compiler, hereafter referred to as the *DSL compiler*, that generates code to create, represent and manipulate candidates from the DSL. In particular, it outputs code to store the decision vector, compute the initial domain, propagate constraints, and perform construct lowering.

Outline. The specification of a decision may contain five kinds of statements, each detailed in a dedicated section:

- $\langle set \rangle$ statements that specify the properties the kernel expose, and how to access the set of objects that respect each property, defined in Section 3.3.1,
- $\langle choice \rangle$ statements to define generic choices, defined in Section 3.3.2,

- ⟨*constraint*⟩ statements to specify the constraints formulae, defined in Section 3.3.3,
- ⟨*lowering*⟩ statements to indicate when and how to lower high-level constructs, defined in Section 3.3.4 and
- ⟨*quotient*⟩ statements to represent classes of equivalent objects, for example fused dimensions, defined in Section 3.3.5.

3.3.1 Kernel Interface Definition

Our DSL defines a decision space on top of an existing kernel representation. Following our formalism, a kernel representation exposes a set of properties Π, and the sets of objects that respect each property $p \in \Pi$. A ⟨*set*⟩ statement defines a single property $p \in \Pi$ and specifies how to access the corresponding set $K(p)$ for each kernel K.

⟨*specification*⟩ ::= (⟨*set*⟩ | ⟨*choice*⟩ | require ⟨*constraint*⟩ | ⟨*lowering*⟩ | ⟨*quotient*⟩)*

⟨*set*⟩ ::= 'set' ⟨*set-ident*⟩ ⟨*set-arg*⟩? ⟨*superset*⟩? ':' ⟨*set-disjoint*⟩? (⟨*entry-name*⟩ '=' ⟨*code-snippet*⟩)* 'end'

⟨*set-arg*⟩ ::= '(' ⟨*ident*⟩ 'in' ⟨*set-ident*⟩ ')'

⟨*superset*⟩ ::= 'subsetof' ⟨*set-ident*⟩

⟨*set-disjoint*⟩ ::= 'disjoint' ':' list(⟨*set-ident*⟩, ',')

⟨*entry-name*⟩ ::= 'type' | 'id_type' | 'id_getter' | 'iterator' | 'getter' | 'from_superset' | 'var_prefix'

Figure 3.1: Top-Level and Set Statements Syntax

Sets Definition Syntax. Figure 3.1 presents the syntax of set definitions in Backus-Naur form. The `list(A, B)` operator indicates a list of `A`s separated by `B`s. A set definition includes the set name, an optional argument, an optional superset, a list of other disjoint sets and several entries that indicate how to generate code that manipulates the set. Each ⟨*entry-name*⟩ may appear only once in a set definition. When generating code from snippets, the CSP compiler substitutes variables starting with `$`. It replaces `$fun` by a pointer to the kernel, `$item` by a pointer to the current object and `$id` by the id of the current object where it makes sense.

```
set Instructions:
  type = "ir::Instruction"
  iterator = "$fun.instructions()"
  id_type = "ir::InstId"
  id_getter = "$obj.id()"
  item_getter = "$fun.instruction($id)"
  var_prefix = "inst"
end
```

Listing 3.1: Instructions Set Definitions

For example, Listing 3.1 defines a property `Instructions` that lists the instructions of the kernel. Beside the name of the set, this definition has a list of entries that map attributes to snippets of code. The CSP compiler embed these snippets in the generated code to manipulate the set and its objects. Mandatory entries are:

`type` : the type of the set objects,

`iterator` : an expression that returns an iterator on the set,

`id_getter` : an expression that returns the unique identifier of an object,

`id_type` : the type of the unique identifier, and

`getter` : an expression that returns the object associated to a unique identifier.

Additionally, the optional `var_prefix` entry specifies a prefix to use when naming a variable holding objects of the set. It makes it easier to debug the code generated by the CSP compiler.

Relations Between Sets. Listing 3.2 shows how we define relations between sets with subset and disjointedness directives. The DSL compiler uses these directives to type-check constraints and to determine when to run propagators. It automatically assumes sets with no common superset disjoint. While limited in expressiveness, these relations nicely match the concepts of type and subtype in programming languages. Listing 3.2 defines two sets `Loads` and `Stores` that correspond to the `ir::LoadInst` and `ir::StoreInst` types in the kernel representation. Both are subtypes of `ir::Instruction` and thus the `LoadInsts` and `StoreInsts` sets are subsets of `Instructions`. Subsets have an additional entry, `from_superset` to test if an object of the superset is in the subset. This entry maps to an expression that returns the object downcast to the correct type if it is part of the set, using the `Option` Rust type.

```
set Loads subsetof Instructions:
  type = "ir::LoadInst"
  from_superset = "$item.as_load()"
  ... // Elided for the sake of brevity
end

set Stores subsetof Instructions:
  disjoint: Loads
  type = "ir::StoreInst"
  from_superset = "$item.as_store()"
  ... // Elided for the sake of brevity
end
```

Listing 3.2: Subset Definition

Parametric Sets. Finally, our DSL supports parametric sets. Parametric sets define a set for each object respecting a property. For example, Listing 3.3 defines a set containing the arrays accessed for each load instruction. In our formalism, this translates to a property on tuples of objects: the parameters and the objects of the set.

```
set AccessedArray($inst in LoadInsts) subsetof Arrays:
  ... // Elided for the sake of brevity
end
```

Listing 3.3: Parametric Set Definition

Note that we specify the set parameter as a variable binding: `$inst in LoadInsts`. This pattern occurs each time we quantify a definition for all objects in a set. The variable may appear in code snippets inside the definition. The DSL compiler replaces it with an expression pointing to the object of the currently considered set.

3.3.2 Choices Definition

Next, we present the syntax to define generic choices in Figure 3.2. We distinguish three kinds of choices declaration, depending on the type of domain they use (`enum`, `integer` or `counter`). Each declares the name of the choice and the objects it applies to in ⟨*choice-decl*⟩. We first present the ⟨*choice-decl*⟩ syntax and then discuss the specificities of each type of domain.

⟨*choice*⟩ ::= 'choice' 'enum' ⟨*choice-decl*⟩ ':' ⟨*enum-stmt*⟩* 'end'
| 'choice' 'integer' ⟨*choice-decl*⟩ ':' ⟨*code snippet*⟩ 'end'
| 'choice' ('half')? 'counter' ⟨*choice-decl*⟩ ':' ⟨*counter-def*⟩ 'end'

⟨*choice-decl*⟩ ::= ⟨*choice-ident*⟩ '(' list(⟨*ident*⟩ 'in' ⟨*set*⟩, ',') ')'

⟨*set*⟩ ::= ⟨*set-ident*⟩ ('(' ⟨*ident*⟩ ')')

⟨*enum-stmt*⟩ ::= 'value' ⟨*variant-name*⟩ ':' ('requires' ⟨*constraint*⟩)*
| 'alias' ⟨*variant-name*⟩ '=' list(⟨*variant-name*⟩, '|') ':' ('requires' ⟨*constraint*⟩)*
| 'symmetric
| 'antisymmetric' ':' (⟨*variant-ident*⟩ '⟨-⟩' ⟨*variant-ident*⟩)*

⟨*counter-def*⟩ ::= 'sum' ⟨*counter-val*⟩
| 'mul' ⟨*counter-val*⟩
| 'forall' ⟨*ident*⟩ 'in' ⟨*set-ident*⟩ ':' ⟨*counter-def*⟩

⟨*counter-val*⟩ ::= 'forall' ⟨*ident*⟩ 'in' ⟨*set*⟩ ':' ⟨*counter-val*⟩
| ⟨*code snippet*⟩ 'when' ':' ⟨*condition*⟩*

Figure 3.2: Domain Specification Syntax

A ⟨*choice-decl*⟩ statement defines the name of the choice and a list of sets corresponding to properties $p_0, \dots p_{k-1} \in \Pi$. Following the formalism of Section 3.2, this defines a CSP variable for each combination of objects respecting the properties, that is, for each combination of objects in the sets. For example, Listing 3.4 defines a generic choice `cache_level` that applies to each instruction in the `LoadInsts` set. In the generated code, it defines a partially specified decision from load instructions to the level of cache to use:

$$\mathit{cache\_level} : \texttt{LoadInsts} \to \mathcal{P}\left(\{\texttt{L1}, \texttt{L2}, \texttt{READ\_ONLY}, \texttt{NONE}\}\right) \tag{3.1}$$

```
choice enum cache_level($inst in LoadInsts):
  value L1:         // Use L1+L2 caches
  value L2:         // Use L2 cache
  value READ_ONLY:  // Use read-only cache
  value NONE:       // Do not use caches
end
```

Listing 3.4: Choice Definition for Cache Directives

Similarly to set parameters, ⟨*choice-decl*⟩ declare sets with a variable binding. This variable may appear in code snippets in domain-specific directives and is replaced by the actual object of the set being considered by the DSL compiler in the generated code.

Enum Choices

Enum choices have a domain D composed of a small set of predefined values. In fixed implementations, they could be represented with C enums. The enum definition specifies the list of values the choice can

take, with optional constraints for each of the values. For example, Listing 3.4 defines one value for each level of cache. The domain D_{cache} of the `cache` choice is defined as follows.

$$D_{cache} := \{\texttt{L1}, \texttt{L2}, \texttt{READ\_ONLY}, \texttt{NONE}\} \tag{3.2}$$

Symmetry. Our DSL supports the specification of symmetry and antisymmetry constraints for enum choices defined with two similar set arguments. In that case the decision vector only stores half of the partially instantiated decision for the choice; the other half is deduced from the first on demand. Symmetry information also helps generate stronger propagators. In the case of antisymmetric choices, a directive specifies the mapping of values to their inverse.

```
choice enum order($lhs in Instructions, $rhs in Instructions):
  value BEFORE:
  value AFTER:
  antisymmetric:
    BEFORE <-> AFTER
end
```

Listing 3.5: Enum Choice for Ordering Instructions

Listing 3.5 provides an example of an antisymmetric choice order between pairs of instructions. This definition also hints at why we only apply generic choices to combination of distinct objects instead of all tuples including the ones with multiple occurrence of the same object: most choices would not make sense when applied to a pair referencing the same object twice. In our example, this avoids expressing the order between an instruction and itself.

Integer Choices

Integer choices select a positive integer values from a small universe specific to each instance of the choice. Their definition includes a code snippet that retrieves the universe from the kernel representation.

```
choice integer tiling_factor($loop in Loops):
  "$loop.possible_tiling_factors()"
end
```

Listing 3.6: Integer Choice for Loops Tiling Factor

For example, Listing 3.6 defines an integer choice that specifies the tiling factor of each loop. A code snippet, defines the a different list of possible factors for each instance of `$loop`.

Counter Choices

Hardware limitations often impose constraints on sums or products of quantities, such as the sum of all arrays sizes (to ensure they fit in memory) or the product of the size of nested thread dimensions (to ensure it does not exceed the maximal number of threads). Counter choices track sums or products of values, to allow for writing constraints that reference their value. They are fully determined by the value of other choices. The DSL compiler inserts code to update counters domains when needed.

For example, Listing 3.7 defines a counter choice that tracks the amount of local memory used by arrays. It sums the value of the `$array.size()` expression for each `$array` that is allocated in local memory.

The DSL compiler represents counter domains by storing their minimal and maximal value. Most constraints on counter actually only need the minimum. In that case, programmers can prefix the counter definition with the `half` keyword to avoid computing and storing the maximal value the counter can take.

```
choice counter local_mem_used():
  forall $array in Arrays:
    sum "$array.size()" when
      mem_space($array) is LOCAL
end
```

Listing 3.7: Counter for the Local Memory Usage

3.3.3 Constraints

Constraints are first-order logic sentences on the choices, quantified over objects in specific sets. Currently, they are universally quantified disjunctions of conditions on zero, one or two choices: boolean constants, restrictions of a variable to specific values or comparisons between two variables. This is done to simplify the implementation, and we never needed more. This is not an inherent limitation of the system.

⟨*constraint*⟩ ::= 'forall' ⟨*ident*⟩ 'in' ⟨*set*⟩ ':' ⟨*constraint*⟩
| list(⟨*condition*⟩, '||')

⟨*condition*⟩ ::= ⟨*decision*⟩ 'is' list(⟨*variant-name*⟩, '|')
| ⟨*decision*⟩ 'is not' list(⟨*variant-name*⟩,'|')
| ⟨*decision*⟩ ⟨*cmp-op*⟩ ⟨*decision*⟩
| ⟨*decision*⟩ ⟨*cmp-op*⟩ ⟨*code snippet*⟩
| ⟨*code snippet*⟩

⟨*decision*⟩ ::= ⟨*decision-ident*⟩ '(' list(⟨*ident*⟩, ',') ')'

⟨*cmp-op*⟩ ::= '==' | '!=' | '<' | '>' | '<=' | '>='

Figure 3.3: Constraint Specification Syntax

Figure 3.3 presents the syntax of constraints. It comprises variable bindings introduced by the `forall` keyword around a disjunction of ⟨*condition*⟩s. The DSL compiler instantiate the conditions for each combination of distinct objects in the sets. A ⟨*condition*⟩ can be:

- a snippet of code evaluating to a boolean, for example to test if a loop may be parallelized,

  ```
  "$loop.has_loop_carried_dependencies()"
  ```

- a restriction of a choice to specific values, for example to indicate an array cannot be stored in global memory,

  ```
  memory_space($array) is not GLOBAL
  ```

- a comparison between a choice and the value returned by a snippet of code, for example to limit the amount of local memory the kernel allocates, or

  ```
  local_mem_used() <= "gpu.local_mem_size()"
  ```

- a comparison between two choices, for example to test if the order between two pairs of instructions are the same.

  ```
  order($a, $b) == order($b, $c)
  ```

```
require forall $a in Instructions:
  forall $b in Instructions:
    forall $c in Instructions:
      order($a, $c) is AFTER
      || order($a, $b) is not AFTER
      || order($b, $c) is not AFTER
```

Listing 3.8: Order Transitivity Constraint

Listing 3.8 gives an example of constraint that ensure ordering decisions are transitive.

The same constraint applies to all combinations of objects in the same sets. Snippets of code appearing in ⟨*condition*⟩s allow for the parameterization of constraints relatively to specific objects or depending on the hardware targets, for example, to parametrize the bound on the amount of memory used or to selectively disable a constraint to only apply to instructions with side effects. The value returned by snippets of code must remain constant throughout the entire implementation space exploration.

3.3.4 Kernel Representation Lowering

The DSL specifies when to lower high-level constructs with trigger statements. Listing 3.9 gives the syntax for triggers. A ⟨*lowering*⟩ statement comprises a snippet of code and a conjunction of conditions surrounded by `forall` statements. It defines a trigger that calls the code snippets when the decision vector meets the condition for all combinations of objects in the sets.

⟨*lowering*⟩ ::= 'trigger' ⟨*lower-forall*⟩

⟨*lower-forall*⟩ ::= 'forall' ⟨*ident*⟩ 'in' ⟨*set*⟩ ':' ⟨*lower-forall*⟩
  | ⟨*code snippet*⟩ 'when' list(⟨*condition*⟩, '&&')

Figure 3.4: Lowerings Syntax

The example in Listing 3.9 defines a trigger that runs when the decision vector indicates that a variable should be stored in memory, that is, when the domain of `is_in_memory($var)` only contains `TRUE`. Triggers run a callback that either adds properties to existing objects or creates new objects. In our example, it creates a store and a load instruction that imply new choices for the cache level they should use. The CSP compiler generates code to extend the decision vector with the new decisions and enforce the new constraints induced by the objects added to the sets representing each property.

```
trigger forall $var in Variables:
  "$fun.gen_store_load($var)" when
    is_in_memory($var) is TRUE
```

Listing 3.9: Lowering Specification to Generate Shared Memory Layouts

Triggers are only allowed to add objects to properties. They cannot remove objects so that previous domains and constraints remain valid. Moreover, we require that callbacks commute, so that the order in which we apply triggers does not impact the final implementations.

It is important to understand that triggers are not mandatory. One can add objects to properties upfront, and restrict the domain of choices to a special value if the objects they relate to do not appear in the generated code. In our example, constraints could force cache directives to `None` in implementations where the lowering condition is not satisfied.

3.3.5 Quotient Sets

Finally, we introduce ⟨*quotient*⟩ statements to manipulate classes of equivalent objects, with regard to a symmetric enum choice that defines an equivalence relation. A ⟨*quotient*⟩ statement defines a set of objects that contains a single representative of each equivalence class that respect a certain property. For example, a quotient set may contain one loop per class of fused loops that are nested outside a specific instruction. The DSL compiler generates code to automatically add objects to the set when decisions introduce a new equivalence class.

⟨*quotient*⟩ ::= 'quotient' ⟨*set-ident*⟩ ⟨*set-arg*⟩? 'of' ⟨*ident*⟩ 'in' ⟨*set*⟩ ':'
⟨*choice-ident*⟩ '=' ⟨*condition*⟩ '/' ⟨*choice-ident*⟩ 'is' ⟨*variant-name*⟩
(⟨*entry-name*⟩ '=' ⟨*code-snippet*⟩)*
'end'

Figure 3.5: Quotient Sets Syntax

Figure 3.5 gives the syntax of quotient definitions. A quotient definition contains the elements of a regular set definition (set name, set argument and mapping of attributes to snippets of code) with additional information to specify the original set of the objects, the condition to be part of the quotient and the equivalence relation. It also defines a generic choice that takes value in {`TRUE`, `FALSE`} that indicates if an object is the representative of an equivalence class. This generic choice is redundant but helps when writing constraints, as it allows for testing the inclusion of an element in the quotient set.

```
// Enum choice defining an equivalence relation.
choice enum is_fused($lhs in Dimensions, $rhs in Dimensions):
  symmetric
  value TRUE:
  value FALSE:
end

quotient OuterLoops($inst in Instructions) of $loop in Loops:
  is_outer_loop = order($loop, $inst) is OUT / is_fused is TRUE
  ... // Elided
end
```

Listing 3.10: Quotient Set for Fused Loops Nested Outside an Instruction

For example, Listing 3.10 defines a quotient set that contains one loop per equivalence class of fused loops nested outside of `$inst`. It defines a redundant generic choice `is_outer_loop` to test if a loop is the representative of an equivalence class.

3.4 Example: Instructions Scheduling Within a Basic Block

We now present a complete example of candidate representation that exposes potential schedules for the instructions of a basic block. For this, we translate the representation of partially specified schedules of Chapter 2 into the candidate formalism.

Listing 3.11 shows the interface of the kernel representation, written in Rust. It defines a structure `InstId` to uniquely identify instructions and two interfaces, `Instruction` and `Kernel`, to represent a single instruction and the whole kernel. An instruction exposes an identifier and a method to test if another instruction is one of its dependencies. The kernel exposes a list of instructions.

Listing 3.12 defines the decision space for schedules. It first specifies how to interact with the list of instructions, then gives the definition of the `order` choice. This definition includes the antisymmetry constraint and ensures that decisions respect data dependencies. Finally, it adds a constraint to ensure that `order` is transitive.

```rust
struct InstId(usize);

trait Instruction {
  fn id(&self) -> InstId;
  fn depends_on(&self, other: &Instruction) -> bool;
}

trait Kernel {
  fn instruction(&self, id: InstId) -> &Instruction;
  fn instructions(&self) -> impl Iterator<Item=&Instruction>;
}
```

Listing 3.11: Interface of the Kernel Representation

```
set Instructions:
  type = "Instruction"
  id_type = "InstId"
  id_getter = "$obj.id()"
  item_getter = "$fun.instruction($id)"
  iterator = "$fun.instructions()"
end

choice enum order($lhs in Instructions, $rhs in Instructions):
  value BEFORE:
  value AFTER:
    // Enforce data dependencies.
    requires "!$lhs.depends_on($rhs)"
  antisymmetric:
    BEFORE <-> AFTER
end

// Ensure the order is transitive.
require forall $lhs in Instructions:
  forall $mid in Instructions:
    forall $rhs in Instructions:
      order($lhs, $rhs) is BEFORE
        || order($lhs, $mid) is not BEFORE
        || order($mid, $rhs) is not BEFORE
```

Listing 3.12: Implementation Space Description

3.5 Constraint Propagation Code Generation

After providing an example of candidate representation, we now explain how the DSL compiler generates Rust code to represent and manipulate candidates from the description of the decision space. Specifically, it generates:

- a data structure to represent the decision vector and access individual variable domains,
- a function `init` that instantiates a candidate from a kernel instance and enforces propagators and triggers on the initial domain,

$$\texttt{init} : \texttt{Kernel} \to \texttt{Candidate} \tag{3.3}$$

- data structures to represent decisions, and
- a function `apply_decisions` that applies a list of decisions to the decision vector and runs propagators and triggers until reaching a fixed point.

$$\texttt{apply\_decisions} : \texttt{Decision}\ \text{list} \to \texttt{Candidate} \to \texttt{Candidate} \tag{3.4}$$

The DSL compiler was critical to conduct our experiments. It made it easier to write, modify and debug decision spaces. This allowed us to try different encoding of the optimizations while only changing a few lines of code in the candidate representation. The DSL compiler also makes our approach generic: one may write its own candidate representation, based on another kernel representation for a different domain of application, different optimizations and different hardware targets.

The main originality of the DSL compiler is that it generates static code to represent the decision vector and enforce constraints. It leverages the generic choices and constraints to generate code that is independent from the kernel instance. It also avoids storing the list of instantiated constraints in memory. Instead, it deduces them on the fly from the kernel instance when necessary.

Outline. Section 3.5.1 explains how we represent the decision vector in memory. Then, Sections 3.5.2 and 3.5.3 respectively present how we implement counters and quotients. Finally, Section 3.5.4 details how we generate propagators and Section 3.5.5 how we use them to enforce constraints and support high-level constructs lowering.

3.5.1 Decision Vector

The DSL compiler first generates code to represent individual domains of variables. That is, types to represent a single enum, integer or counter instance. It encodes enum domains in a bit-field with one bit per possible value. Similarly, it encodes integer domains with a fixed number of bits, each bit corresponding to a position in the array that specifies the possible choices. Finally, it encodes counter domains as intervals with their minimal and maximal possible values (only the minimal value for half counters).

The compiler then generates a data structure for the decision vector. This data structure comprises a hash map for each generic choice, that maps tuples of objects to the domain of the generic choice instantiated on that tuple of objects. The decision vector relies on shared pointers with a copy-on-write mechanism to share unmodified hash maps among the decision vector copies.

Finally, it puts together the decision vector and a kernel to form a candidate. The programmer must provide the kernel representation. Here again, a copy-on-write mechanism avoids copying the kernel, unless a high-level construct is lowered with a trigger.

3.5.2 Counters Implementation

Let K be a kernel instance. A counter generic choice $c = ((p_0, \ldots, p_{k-1}), \mathbb{N}) \in C$ tracks a distinct sum or a product of values for each combination of objects $x \in K(p_0) \times \cdots \times K(p_{k-1})$. To simplify notations, this section assumes that c computes a sum. The implementation is similar for products. A counter

instance $c(x)$ tracks the sum over all combination of objects $y \in \Omega^*$ that respect a list of properties $p_k, \ldots, p_{m-1} \in \Pi$ and a formula $f(x, y)$. Specifically, $c(x)$ is defined as:

$$c(x) = \sum_{y \in K(p_k) \times \cdots \times K(p_{m-1})} \delta(x, y).v(x, y) \tag{3.5}$$

where $\delta(x, y)$ equals to 1 if $f(x, y)$ evaluates to true and 0 otherwise.

To implement counters, the DSL compiler first creates a new generic choice with a boolean domain to materialize δ. Per the syntax of counter choices in Section 3.3.2, $f(x, y)$ is a conjunction of conditions:

$$f(x, y) = f_0(x, y) \wedge \cdots \wedge f_{n-1}(x, y) \tag{3.6}$$

The compiler translates (3.6) into the following constraints. These new constraints follow the structure of regular constraints in the DSL and the compiler treats them as such.

$$\delta(x, y) \vee \neg f_0(x, y) \vee \cdots \vee \neg f_{n-1}(x, y) \tag{3.7}$$

$$\forall i < n. \quad \neg\delta(x, y) \vee f_i(x, y) \tag{3.8}$$

The decision vector represents counters with their minimum and maximum possible values, or just the minimum for half counters. The DSL compiler inserts code to update the minimum or the maximum when the domain of an instance of δ changes to 0 or 1. In the case where v depends on the value of another choice c', the compiler also inserts code to update the counter domain when the minimum or maximum value of c' changes.

The introduction of δ enables us to use propagators that only manipulate a few CSP variables at once instead of all the variables appearing in the sum expression (3.5). The compiler modifies the code of propagators that would restrict the value of the counter to act on δ instead. If a propagator would limit the maximum value of $c(x)$ to M, the generated code forces to 0 the instances of δ that would make $c(x)$ greater than M if set to 1. Conversely, it forces to 1 the instances of δ required for the value of $c(x)$ to be above the minimum valid value. If v depends on another choice c', propagators also limit possible values of c'.

When a trigger runs, it adds new objects to the sets representing properties. Adding theses objects can add terms to the sum (3.6) and increase the maximum possible value of the counter. This could in turn make previously invalid decisions valid, and thus violate the commutativity of decisions. To avoid this, we force δ to 0 for new terms.

3.5.3 Quotients Implementation

Let K be a kernel instance. A quotient set Q accounts for classes of equivalent objects $x \in \Omega^*$ respecting a property $p \in \Pi$, and a condition $f(x)$, up to a relation r. Q must contain one representative of each equivalence class. The DSL compiler generates code to automatically populate the set.

For this, it counts, for each object $x \in K(p)$, the number of class representatives in Q that may be equivalent to x. When this number reaches 0 and $f(x)$ evaluates to true for all possible values in the decision vector, the code adds x to Q.

3.5.4 Propagators

We now explain how we generate propagators that enforce constraints. The main difference with classical CSP solvers is that we apply the same constraints to all combinations of objects respecting the same properties. We can thus statically generate code that runs a propagator for a single combination of objects and call it with different combinations of objects. Listing 3.13 gives an example of propagators that enforce the transitivity of the `order` generic choice. It implements the constraint expressed in Listing 3.8. The same propagator applies to all triplets of instructions.

A key corollary of statically generating the propagation code is that we do not need to store the constraints in memory. Instead, we generate code that iterates over the sets of objects and calls propagators code when necessary. This decreases the overhead for copying candidates as we only need to copy the decision vector and, in a few cases, the kernel representation (when a triggers runs). The CSP linked to a candidate is implicitly deduced from the kernel.

```
fn propagate_order_0(
  a: &Instruction,
  b: &Instruction,
  c: &Instruction,
  candidate: &Candidate
) -> Order {
  let order_a_b = candidate.domain.get_order(a, b);
  let order_b_c = candidate.domain.get_order(a, b);
  // Compute possible values for `order_a_c`.
  let order_a_c = Order::ALL;
  if order_a_b == Order::AFTER && order_b_c == Order::AFTER {
    order_a_c = Order::AFTER;
  }
  return order_a_c;
}
```

Listing 3.13: Propagator Enforcing the Order Transitivity

Propagator Generation

In the general case, the optimal propagator for a pair of constraints is stronger than the fixed point of the propagators for the individual constraints. That is, the propagator for the combined constraints may detect invalid values earlier than the combination of the propagators for the individual constraints. It is thus beneficial to group constraints together to generate propagators. However, increasing the number of significant variables of a propagator causes it to execute more frequently when restricting a domain. The compiler workflow to generate propagators, laid out in Figure 3.6, works as follows.

1. It normalizes constraints so that each CSP variable appears at most once in a single constraint.
2. It transforms constraints into propagators, with one propagator for each significant variable of the constraint. For now, a propagator has the form of an implication with the condition referencing the restricted variable on the right-hand side of the implication and other conditions on the left-hand side.
3. It applies symmetry rules to introduce new propagators, from existing ones.
4. It groups propagators with similar significant variables. This way, we can later generate propagators combining constraints without increasing the frequency at which they must run.
5. It combines propagators within a group. To this end, it generates a truth table that lists the possible values of restricted variable depending on the values of significant variables. The truth table only instantiates the values of enum choices. Other domains have either too many possible values (for counter domains) or have values that depend on the kernel instance (for integer domains). Each cell of the truth table thus contains a partially evaluate expression listing possible values.

   Creating the truth table allows the compiler to evaluate propagators on fixed domains, where they have maximal precision, and to combine propagators independently in each cell of the truth table.
6. It converts the truth tables into propagators. For that it builds trees of if-conditions that compute the union of the cells of the truth table that correspond to assignments of the input variables licensed by the current domain.

3.5.5 Propagation Loop

To enforce propagators, update counters and run triggers, the DSL compiler generates a function `propagate` that maintains a set U of the CSP variables whose domain has been updated. This function, outlined in Algorithm 1, iteratively picks an updated variable $v \in U$ and runs the propagators that

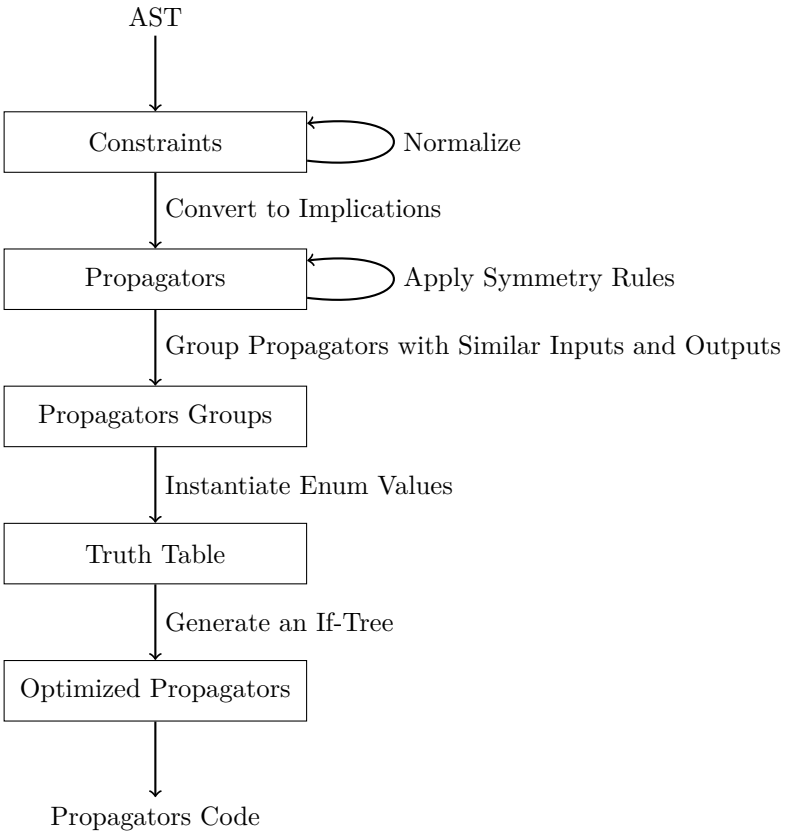


Figure 3.6: Propagators Generation Steps

depend on v. At the same time, it also updates affected counters and runs triggers whose condition is enabled by the modification of v. It adds back to U any variable that it modifies in the process.

The `propagate` function runs each time the search algorithm restricts a domain. To initialize the decision vector in the first place, we use a different strategy. Directly using `propagate` would lead to the execution of the same propagator for each of its significant variables. Instead, the initialization procedure, `init`, calls all propagator at once, computes the initial value of counters and tests if each trigger should run. It then runs all propagators a second time and calls `propagate` with U only containing the list of variables whose domains were restricted during the second run of propagators.

Running a trigger adds new objects to the sets representing the kernel properties. These objects induce new CSP variables and new constraints. When this occurs, `propagate` calls a function `partial_init` to extend the decision vector and to enforce the new constraints. It follows the same strategy as the `init` procedure, but only applies to new CSP variables and constraints.

3.6 Discussion

The key innovation of our approach is to work on classes of implementations up to semantics preserving implementation decisions. We first discuss in Section 3.6.1 how this obviates the problem of optimization ordering and in Section 3.6.2 how this enables global heuristics aware of all potential optimizations. We then explain in Section 3.6.3 how our system differs from other encodings of compilation problems as operational research problems.

3.6.1 Partial Implementations

Ordering decisions is a common problem in compilers: transformations may enable, disable or even undo others. Some domain-specific approaches tame the issue with a carefully curated set of rewrite rules, that provides a well-defined implementation space. They build and explore a tree (or graph) whose

Algorithm 1: Constraint Propagation Procedure `propagate`

```
Data: U, the list of modified variables
while U ≠ ∅ do
    remove v from U;
    foreach propagator p depending on v do
        run p and insert the variables it modifies in U;
    end
    foreach counter c whose value depends on v do
        update the counter and add c to U;
    end
    foreach trigger t where v appear in the triggering condition f do
        if the change to v makes f evaluates to true then
            update the kernel representation;
            call partial_init(U);
        end
    end
end
```

nodes are implementations and edges rules applications. For example, Spiral [4] uses this approach for fast Fourier transforms, LGen [5] for linear algebra kernels and TCE [12] for large tensor contractions in computational chemistry and physics simulation. More recently, LIFT [6] applied rules to rewrite a functional representation down to low-level OpenCL constructs and generate efficient GPU code.

However, the initial problem is still present: some transformations may be hidden behind others. We do not see how one could access the full set of potential downstream decisions from an early point of the compilation process without building the full search tree.

An alternative is to use algorithmic skeletons [13] and to map them to the hardware using a fixed strategy that leverages domain specific information. In their simplest form, they can be provided as parametric libraries, such as Thrust [14] and SkelCL [15]. Otherwise, it takes the form of high-level functional operators in a domain specific language such as Delite [16], Copperhead [17], Accelerate [18] or NOVA [19]. While theses systems allow for optimizations, such as the fusion of operators, they rely on a fixed strategy that makes it hard to adapt to different hardware, different input sizes and new kind of computations.

The Sea of Nodes approach [20] places instructions outside of basic blocks when possible, effectively representing classes of implementations up to scheduling decisions. However, this is limited to scheduling decision.

3.6.2 Global Heuristics

The partially instantiated vector offers a complete view of potential implementations. This allows for the definition of global heuristics that are aware of what a fully optimized implementation look like. The lower bound performance model mentioned in Chapter 5 could not work if it just had access to an intermediate implementation in the compilation process. We had previously introduced a similar performance model that relies on ad-hoc partial implementations [21]. We generalize the idea by encoding partial implementations as a CSP on top of a semantic backbone.

Our approach is close to Equality Saturation [22], with similar benefits. Equality Saturation uses rewrite rules but keeps both the original and the rewritten pattern. However, the number of patterns can grow exponentially with the number of rules applies. In contrast, extending the decision space with new decisions only adds a new partially specified decision to the decision vector. The fixed-size decision vector also makes it easier to extract features for machine learning algorithms.

Frameworks that dissociate the algorithm from the schedule, such as Halide [23] for image processing and TVM [24] for machine learning arguably also deal with partial implementations. The algorithm is akin to our kernel representation and the schedule to our decision vector. However, they do not have an easy way to reason about partial schedules. TVM applies machine learning techniques, but only on fully specified implementations. An interesting idea would be to use our approach on top of their

representation to explore schedules. This is also true for other script-based transformation tools such as UTF [25] or URUK [26] that start from a fully specified implementation, but lift it into a mathematical representation that abstracts the initial schedule.

3.6.3 Encoding as an Operation Research Problem

The idea of encoding compilation problems in logical frameworks is not new. Polyhedral compilation encodes transformations on loop nests as Integer Linear Programming [27, 28]. Superoptimization techniques also use CSPs [29] and SAT [30] solvers to generate optimal sequences of instructions.

The originality of our approach is to use CSPs to expose potential decisions, instead of using it to find a solution (Section 4.5.2 shows that finding a solution is easy in our case). This allows us to manipulate whole sets of potential implementations, which is the main motivation for using CSPs: it is easier to guide the search through the domains, while SAT and ILP solvers often act more like black boxes.

ILP-based approaches maximize a single metric (such as data locality in polyhedral schedules [31]) that does not reflect the complexity of the architecture. Since we do not try to embed performance constraints in the logical framework, we are more flexible and can use a combination of custom heuristics, actual evaluations and statistical search.

One way of solving this problem while staying in an ILP framework is to find schedules for a single loop level at a time, starting from the outermost [31] loop. This enables more complex heuristics and incremental evaluation at each level that goes beyond the expressive power of linear objective functions [32]. However, loop levels must still be considered in a fixed order. It would be interesting to use our approach with a similar encoding. Polyhedral compilers have been designed with search algorithms complementing or replacing linear programming. Pouchet et al. proposed custom genetic operators on affine schedules to search for dependence-preserving loop nest optimizations [33, 34]. Vasilache et al. constructed a two-level optimization strategy where the lower tier is a gray box based on integer linear programming, exposing a fixed set of strategies and parameters to a higher tier genetic algorithm [35].

3.7 Summary

This chapter presented a novel approach for program optimization. The compiler restricts a candidate that represents a whole set of potential implementations. This contrasts with traditional approaches that iteratively apply transformations to a single implementation. Our approach exposes optimization decisions upfront and makes sure that they are commutative, obviating the problem of optimization ordering. A search algorithm can make the most performance-impacting decisions first and expert programmers can manually specify any decision if necessary.

At the core of our representation of candidates is a decision vector listing all possible decisions for each choice. It allows for the definition of global heuristics that operate on sets of potential implementations. They are aware of what the final implementation may look like and can derive pertinent information, such as profitability estimates to guide the search and performance bounds to prune the space, long before all decisions have been made. Chapter 5 presents how to use our representation to define a performance model that provides a lower bound on the performance achievable by a candidate implementation. Chapter 6 combines our approach in actual search algorithms.

Decision Space Formalism. We based our concept of candidate on Constraint Satisfaction Problems. However, instead of using CSPs as optimization problems, we use them as a representation of the available implementation decisions. In particular, we have no analytical objective function. An external algorithm drives the search towards the best implementation using empirical evaluation. Section 4.5.2 shows that finding a single implementation is not a problem, as CSP variable domains rarely contain invalid values.

The formalism we propose for candidates defines a decision space as a generic CSP to instantiate for each kernel instance. This way, the programmer can specify the decision space once and automatically apply it to different kernel instances. The formalism supports lowering high level constructs, when some condition is met, without invalidating other decisions. This is useful, for example, to generate store and load instruction after deciding to store a variable in memory. While relying on the lowering of high-level constructs in the decision space is useful to avoid cluttering the implementation space with decisions that may not appear in the final implementation, we can always expose the choices and constraints

they introduce directly in the initial candidate. This ensures our claim that we can expose any decision upfront is not invalidated.

Candidate Representation Generation from the DSL. Starting from a declarative description of implementation choices and their interaction, we built a tool that generates code to represent and manipulate candidates. This tool facilitates building, debugging and experimenting with the decision space, and porting our approach to different architectures or domains of application.

We could implement our tool on top of a standalone CSP framework; apart from the need for a domain-specific front-end to provide the above-mentioned independence on the kernel instance, this would involve several optimizations and customizations to reproduce the search strategy and match the efficiency of our approach, motivating a custom design. In particular, we use the fact that the CSP definition is independent of the kernel instance to avoid storing constraint instances in memory. Instead, we statically generate code that we call for different objects. This reduces the memory footprint of the CSP to the sole decision vector, making it compact, easy to serialize for logging and fast to copy.

Chapter 4 applies our tool to the construction of a decision space for linear algebra kernels running on GPUs. We use this candidate representation with a search algorithm in Chapter 6. Overall, our domain-specific language for defining decisions and constraints proved useful for developing an decision space for linear algebra, facilitating the design and experiments with numerous decisions and constraints. We are not aware of any other approach reaching the same level of automation while remaining competitive with native GPU libraries.

Chapter 4

Application: Linear Algebra on GPUs

Graphics Processing Units (GPUs) offer massive parallelism with a raw processing power an order of magnitude above contemporary multicores. For dense linear algebra, this parallelism is game changing. For instance, the recent success of deep learning methods can largely be attributed to the performance of GPU-accelerated linear algebra and convolution libraries [1, 2].

Unfortunately, as explained in Chapter 1, writing or generating efficient code for GPUs is a complex problem. It requires making careful decisions, such as mapping computations to the appropriate level of parallelism and moving data across the memory hierarchy and through memory spaces of different sizes and properties, in addition to the many possible thread-local optimizations. While the set of potential implementation decisions is usually well known, complex interactions between them make it hard to find the best global implementation. The production and the specialization to a given target or problem size of highly tuned libraries remain a critical challenge.

This chapter proposes a candidate representation for linear algebra kernels on GPU, following the ideas and the formalism of Chapter 3. It shows that our approach is sufficiently expressive to model the interaction between complex transformations that fundamentally change the structure of the code. We present both the kernel representation and the decision space, using the DSL we presented in Section 3.3 for the latter.

We chose to focus on the most important transformations to achieve good performance. Rather than directly exposing switches to enable or disable a particular optimization, our representation exposes many small knobs that encode the local structure of the code. From there, supporting a transformation translates to ensuring that knobs can represent both the original and the transformed implementation.

Contributions. This chapter explains how we encode the implementation space, while Chapters 5 and 6 show how to use this encoding to derive pertinent information early in the compilation process and to find good implementations. Specifically, this chapter makes the following contributions.

- It demonstrates that the candidate formalism is expressive enough to represent complex decisions that fundamentally change the structure of the generated code, including compositions of strip-mining, loop fusion, loop interchange, unrolling, vectorization, mapping to multiple levels parallelism, and orchestrating data movements across the memory hierarchy.

- It studies the design of decision spaces, based on the experience we gained from the decision we built for linear algebra kernels on GPUs. In particular, it shows that finding a valid implementation does not require a complex solver apart from constraint propagation. This confirms the assertions made in Chapter 3 that individual decisions domains accurately represent valid decisions.

- It provides a novel program representation for optimizing linear algebra kernel for GPUs. Chapter 6 shows that this representation allows for the generation of code that is competitive with carefully hand-tuned reference libraries, and outperform them in some cases.

Outline. Section 4.1 first gives some background on GPU architectures. Section 4.2 then presents how the kernel representation encodes the semantics of the code to generate, Section 4.3 shows how we encode transformations in our formalism and Section 4.4 explains how we generate code from a fixed candidate. Finally, Section 4.5 discusses the specificities of our representation of GPU programs and the insights it provides on the design of candidate representations, while Section 4.6 summaries the chapter and explain how we use our candidate representation in subsequent chapters.

4.1 Background: GPUs Architecture

This section provides some background on the CUDA programming model [36] and the architecture of CUDA GPUs. Other GPUs rely on a similar programing model and architectures [37]. We first present the parallelism model of GPUs in Section 4.1.1 and their memory hierarchy in Section 4.1.2.

4.1.1 Parallelism Model

Thread Hierarchy. A CUDA program defines a function that is executed N times in parallel, by N different threads. Each thread has a unique identifier composed of two 3-dimensional vectors: a *thread index* $(t_x, t_y, t_z) \in \mathbb{N}^3$ and a *block index* $(b_x, b_y, b_z) \in \mathbb{N}^3$. These vectors identify threads as points in a 6-dimensional hyper-rectangle.

Threads sharing the same block index form a *block*; while the different blocks form a *grid*. Thread indexes identify threads within a block while block indexes identify blocks within the grid. We call (T_x, T_y, T_z) the number of threads along each dimensions of the thread index and (B_x, B_y, B_z) the number of blocks along each dimension of the block index. Every $(b_x, b_y, b_z) \in \mathbb{N}^3$ such that $b_x < B_x$, $b_y < B_y$ and $b_z < B_z$ indexes a block, while every $(t_x, t_y, t_z) \in \mathbb{N}^3$ such that $t_x < T_x$, $t_y < T_y$ and $t_z < T_z$ indexes a thread in each block. Figure 4.1 gives an example of a 2-dimensional grid with 2-dimensional blocks. In that case, $(T_x, T_y, T_z) = (2, 2, 1)$ and $(B_x, B_y, B_z) = (3, 2, 1)$.

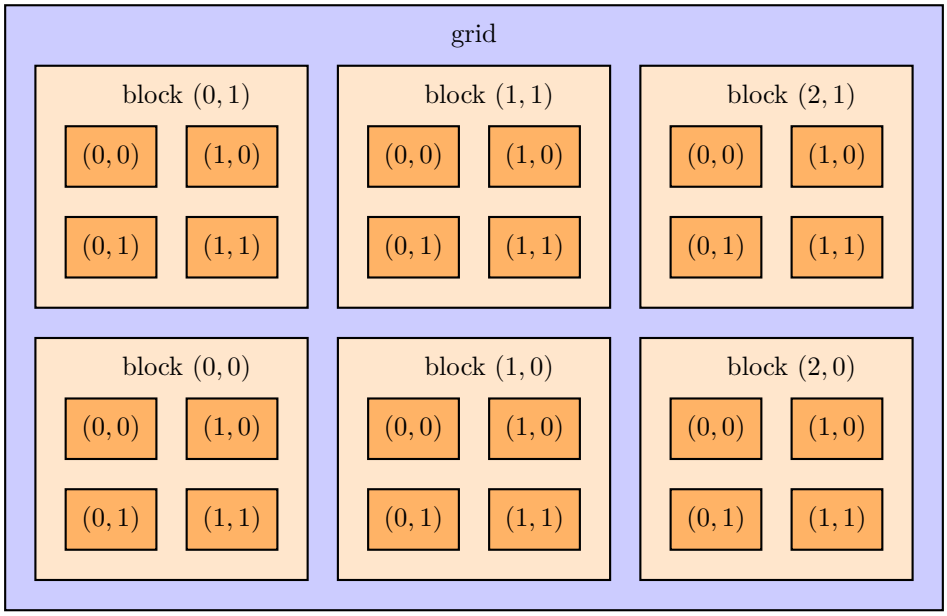


Figure 4.1: CUDA Thread Hierarchy in 2D

Threads within a block share a same L1 cache, have access to a fast shared memory and have fast synchronization primitives. On the opposite, communication and synchronization across blocks is expansive.

Warps. Within a block, CUDA groups consecutive threads into warps, following the lexicographical order on thread indexes. A warp contains up to 32 threads. On the GPUs we consider, threads within

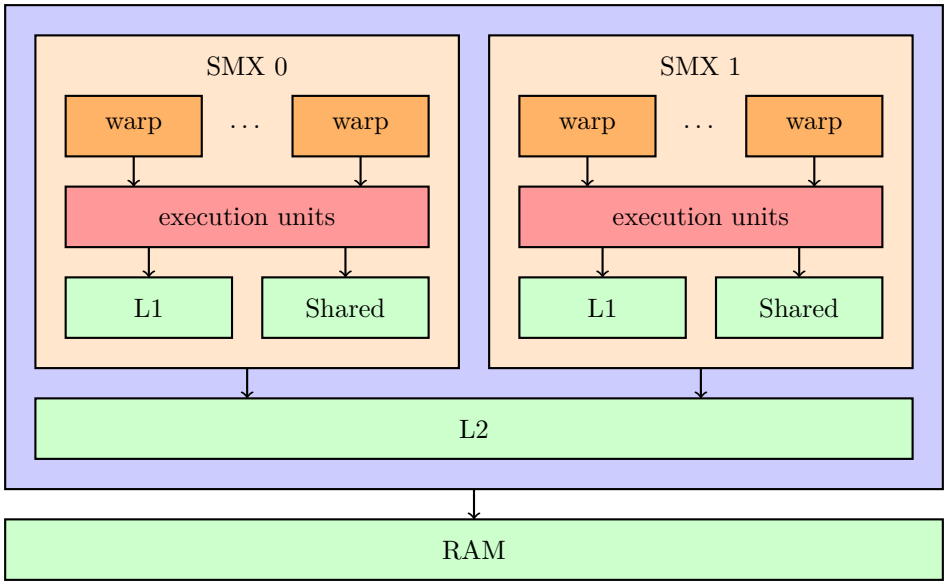


Figure 4.2: CUDA GPUs Architecture

a warp execute in lockstep; that is, they all execute the same instruction at the same time on different data. Different warps may execute in any order.

Mapping Threads to the Hardware. Figure 4.2 present the architecture of CUDA GPUs. A GPU processor contains multiple Streaming Multiprocessors (SMX for short) that execute warps in parallel. This figure features two SMXs but a GPU may contain more.

CUDA dispatches blocks across SMXs. A block may need to wait for other blocks to execute if all blocks do not fit on SMXs. The number of parallel blocks an SMX supports depends on the amount of memory and registers each block allocates.

An SMX contains a multitude of small cores that provide different kinds of execution units. At each cycle, an SMX can schedule a few warps on the cores and execute one or two instructions for each thread in the warps. Cores are pipelined so the SMX can hide instruction latency by scheduling other warps while the stalled warps complete.

4.1.2 Memory Hierarchy

GPUs have two memory spaces: global memory and shared memory. Global memory is stored in RAM and is accessible by any thread while shared memory is stored within SMXs and is private per block. In practice, CUDA also expose a constant memory space and a memory space local to each thread, but both are implemented on top of the global memory.

Global Memory. Figure 4.2 shows the memory hierarchy of CUDA GPUs. Global memory accesses may use an L1 and L2 cache. The L1 is not coherent so it is mostly used for accesses to constant data or to data local to a thread. Load and store instructions to global memory can choose to bypass the L1 or both caches.

Memory units coalesce accesses within a warp: they load L1 cache lines only once, even if multiple thread access them. This is critical for performance as most memory accesses bypass the L1. When threads access different cache lines, performance drops as the SMX needs to perform more memory operations.

Shared Memory. Shared memory is stored within an SMX. It provides fast access but has a limited size and is only shared between the threads of the same block. It does not have any caches as it is already close to execution units. Reusing data across threads using the shared memory is crucial to reduce the global memory footprint.

4.2 Kernel Representation

A candidate contains two parts: a kernel representation that encodes the semantics of the code to generate and a decision vector that encodes already fixed and open implementation choices. This section presents the kernel representation we defined for linear algebra on GPUs.

The main goal of the kernel representation is to define the semantics of the kernel without imposing a particular structure on the final code, leaving room for implementation decisions to reorganize computations. The kernel representation thus defines the operations to be executed without mentioning the loop structure nor the evaluation order. Since we focus on dense linear algebra, we assume that all loops are *for*-loops with data-independent bounds (i.e., loop bounds only depend on the input size) and that memory accesses are linear accesses to n-dimensional tensors.

We currently do not support any particular input language to define the kernel representation. Instead, a programmer must create the kernel representation with API calls. We intend to solve this in the future, but the kernels we consider are still sufficiently small to build them with API calls, thus allowing us to focus on the kernel representation and the implementation space representation.

A kernel is translated to a single function in the generated code. The kernel representation defines a function with a name, a list of arguments and a body that specifies the computations the kernel performs. The rest of this section explains how we encode the body of a kernel. Section 4.2.1 explains how we represent the computations and their iteration space, Section 4.2.2 discusses how we represent communication between instructions and Section 4.2.3 shows how we deal with memory accesses and arrays.

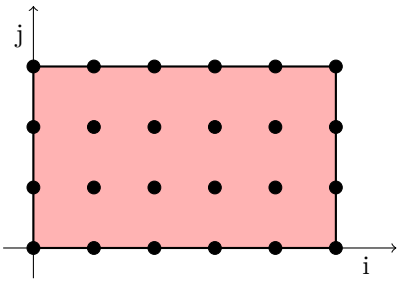
4.2.1 Instructions and Iteration Dimensions

Our goal is to represent dense linear algebra programs. Such programs are usually composed of multiple loop nests, with instructions nested inside the loops. However, in our case, the concept of loop is not directly applicable as the structure of the code is not yet specified. Instead, each instruction as an *iteration space* that specifies how it should be repeated. We first define the notion of iteration space, then explain how we specify instructions and finally detail how we handle strip-mining.

Iteration Space. Let's consider a loop nest of depth n in a traditional imperative programming language. Each iteration of the loop nests corresponds to an integer point in an n-dimensional space, specified by the vector composed of the index along each loop. In our case, we only consider loops with data-independent loops so that the iterations form an n-dimensional hyper-rectangle. Each instruction nested inside the loop nest is executed once for each point in the hyper-rectangle. We denote this hyper-rectangle the *iteration space* of the instruction. For example, Figure 4.3 gives the iteration space of an instruction in a loop nest of depth 2.

```
for i in 0..6:
  for j in 0..4:
    x = foo()
```

(a) Loop Nest with an Instruction x



(b) Iteration Space of x

Figure 4.3: 2-Dimensional Iteration Space

As mentioned earlier, the concept of loop is not directly applicable to our case, as the structure of the code may not be specified yet. Instead we describe the iteration space of each instruction by defining the dimensions of the hyper-rectangle. To that end, we introduce the concept of *iteration dimension*. An iteration dimension has a size, potentially fixed by implementation decisions as explained below. An iteration space is the product of any number of iteration dimensions. The size of the iteration space is

the product of the size of the dimensions. The same dimension may appear in the iteration space of multiple instructions.

Decisions may specify to implement iteration dimensions in various ways as long as the generated code executes the instruction the correct number of times along each dimension. For example, a dimension could be implemented as a regular counted loop, or mapped to a parallel dimension of the GPU.

Instructions. The kernel body defines a list of instructions to execute, each with its iteration space. An instruction can be an arithmetical operation on one of the basic data types supported by the GPU, a move, a cast operation or a memory access.

Instruction operands can be values produced by another instruction, constants, values of inputs of the kernel, indexes along one of the dimensions of the iteration space of the instruction, or addresses of elements from in-memory tensors. Section 4.2.3 details the specificities of memory access instructions and memory address operands.

Logical Iteration Dimensions. We expose strip-mining opportunities by creating multiple iteration dimensions for each dimension of the original problem. A dimension d strip-mined k times becomes a set of dimensions $\{d_0, \ldots, d_k\}$. Dimensions $d_1, \ldots, d_k$ each have a list of possible sizes that correspond to the strip mining factors, while the size of d_0 is inferred from the size of the other dimensions, so that the loop nest of $d_0, \ldots d_k$ matches the size of the original dimension. A strip mining with a factor of 1 corresponds to no strip mining. In the rest of this chapter, we call *logical dimension* the dimension d of the original problem.

The kernel representation maintains both the list of logical dimensions and the list of actual iteration dimensions. The logical dimension lists the possible strip mining factor of the original dimensions, that is the possible values of the product $\prod_{1 \leq i \leq k} \text{size}(d_i)$. This is useful to ensure the strip mining factor is a multiple of the logical dimension size.

Example: Vector-Scalar Multiplication. Table 4.1 gives the representation of a kernel that computes the vector-scalar addition. This kernel takes a scalar α and two arrays, X and Y, of size n as input and computes $Y = \alpha \times X$.

Instruction	Iteration Space
`x = load X[`$\langle d_0, d_1, d_2 \rangle$`]`	$d_0 \times d_1 \times d_2$
`y = x * `α	$d_0 \times d_1 \times d_2$
`store y in Y[`$\langle d_0, d_1, d_2 \rangle$`]`	$d_0 \times d_1 \times d_2$

(a) List of Instructions

Name	Size	Dimensions	Strip-Mining Factors
d	n	$\{d_0, d_1, d_2\}$	$\{1, 2, 4, 8, 16\}$

(b) List of Logical Dimensions

Dimension	Possible Sizes
d_0	$\frac{n}{\text{size}(d_1).\text{size}(d_2)}$
d_1	$\{1, 2, 4, 8\}$
d_2	$\{1, 2, 4, 8\}$

(c) List of Dimensions

Table 4.1: Kernel Representation for the Vector-Scalar Addition

This table shows the list of instructions, the list of logical dimensions and the list of actual iteration dimensions. The kernel has three instructions that respectively load a value of X, perform the multiplication by α and store the result in Y. Each instruction has the same iteration dimensions d_0, d_1 and d_2. The expression $\langle d_0, d_1, d_2 \rangle$ in the memory accesses converts the current index along dimensions d_0, d_1 and d_2 into a unidimensional index.

$$\langle d_0, d_1, d_2 \rangle := \text{index}(d_2) + \text{size}(d_2).\,(\text{index}(d_1) + \text{index}(d_0).\text{size}(d_1)) \tag{4.1}$$

Iteration dimensions d_0, d_1 and d_2 form a single logical dimension d of size n. Both d_1 and d_2 take values in $\{1, 2, 4, 8\}$, but the total strip mining factor $\text{size}(d_1) \times \text{size}(d_2)$ cannot exceed 16 as imposed by the list of possible strip mining factors of d.

4.2.2 Data Communication

We now explain how we represent data communication between different iteration spaces, and between different points of the same iteration space.

Point-to-Point Communication. Let a and b be two instructions such that a produces a value x and b consumes it. Instruction a produces a value for x at each point of its iteration space. By default, b reads the value of x produced within the same iteration of the common iteration dimensions of a and b and at the last iteration of the remaining dimensions. For example, if the iteration space of a is $\{d_0, d_1\}$ and the iteration space of b is $\{d_0, d_2\}$, then the iteration (i, j) of b will read the value produced at iteration $(i, \text{size}(d_1) - 1)$ of a.

Let's now assume d_1 and d_2 have the same size. The operand can specify that it maps dimension d_1 to d_2. In that case, iteration j of d_2 will use the values produced at iteration j of d_1. Each operand that reads the value produced by another instruction has a list of mapped dimensions, implementing point-to-point communication between iteration dimensions. Decisions can choose to implement point-to-point communication by fusing the source and destination iteration dimensions, or by temporarily storing the data in arrays allocated in registers or in memory.

Loop Carried Variables. Alternatively, an operand of an instruction a can take the value produced by a at the previous iteration along a given set of dimensions. In that case, the operand also specifies an instruction that initializes the value at the first iteration. In particular, this construct allows for the definition of computations that perform a reduction along a set of iteration dimensions. Note that the operand may still specify point-to-point communication between the initialization and the reduction instructions.

As an example, let a and b be two instructions with respective iteration spaces $\{d_0\}$ and $\{d_0, d_1\}$. Let's assumes that instruction a consumes the value it produces at the previous iteration of d_1, initialized with b. The result is that, iteration (i, j) of a will read the value produced by the iteration $(i, j - 1)$ of a if $j > 0$ and iteration i of b otherwise.

4.2.3 Arrays and Memory Accesses

In addition to instructions and dimensions, the kernel representation also specifies a list of arrays. An array can either be an input of the kernel or an internal array that holds temporary values exchanged between two instructions. In the latter case, the size and layout of the array are not fixed by the signature of the function, but by decisions. The kernel thus exposes two kinds of arrays: regular arrays with a fixed size and layout and partially specified arrays that only specify the element size and the list of potential dimensions.

Memory access instructions can specify the stride of memory accesses along each iteration dimension. Without this information, the performance model ignores the instruction and some optimizations are forbidden. The access patterns our kernel representation can express with strides correspond to linear accesses to n-dimensional tensors.

Linear algebra kernels often perform linear accesses to n-dimensional tensors. This results in memory addresses computations that have the following form:

$$a = a_0 + \sum_{i<k} s_i.\texttt{index}(d_i) \tag{4.2}$$

where for all $i < k$, d_i is a dimension from the iteration space of the current instruction and $s_i \in \mathbb{N}$ a constant stride. Generating optimized code for such computations is straightforward. Therefore, we provide a special construct to express addresses of linear accesses into tensors without exposing the internal details of the computation. This avoids polluting the implementation space with decisions about code that can be generated following a fixed strategy. The code generator lowers these expressions once all decisions are fixed.

4.3 Decisions Encoding

We now describe the second part of candidates for linear algebra on GPUs: the decision vector and its associated constraints and triggers. We give the definition of all the generic choices that compose the decision vector using the DSL presented in Section 3.3. Overall, generic choices focus on exposing small decisions rather than entire transformation. We detail the optimizations each generic choice enables.

To define generic choices, we first need to declare the sets of objects exposed by the kernel representation. This comprises the three sets describe in Section 4.2, but also some derivatives of these sets, such as the list of instructions that respect a certain property. For the sake of clarity, we omit the attributes that specify how to generate code from set definitions. The elided code is indicated with three dots.

We also expose some constraints, when they help understand how we enforce the correctness of the final implementation and how decisions interact with each other. For the full list of constraints, the reader should refer to the decision space definition in the source code of our code generator for linear algebra kernels on GPUs. Constraints serve the following purposes.

- They enforce the coherency between decisions, e.g. that the instruction order is transitive.
- They ensure that the implementation respects the semantics of the kernel representation, e.g. that it respects data dependencies.
- They enforce hardware-specific limitations. For example, that the size of the arrays allocated in a memory space does not exceed the size of the memory space.

Finally, we specify triggers that lower high-level constructs in the kernel representation. This only concerns the implementation of point-to-point communication between iteration spaces through arrays.

Section 4.3.1 first details the choices and constraints relative to iteration dimensions. Next, Section 4.3.2 explains how decisions specify the structure of the generated code: the organization of loops and the order of instructions. Then, Section 4.3.3 explains how our decision space maps computations to hardware threads and Section 4.3.5 presents memory spaces and caching decisions. Finally, Section 4.3.4 details how we handle point-to-point communications.

4.3.1 Iteration Dimension

This section first explains how we expose iteration dimensions in the decision space, then how we choose how to implement them and last how we specify and constrain strip-mining factors.

Dimension Sets. Listing 4.1 shows the declaration of the `Dimensions` set that lists all iteration dimensions. It is a subset of `Statements`, later defined in Section 4.3.2. Within `Dimensions`, we distinguish two kinds of dimensions: static dimensions whose size is known during code generation and dynamic dimensions whose size depends on the kernel inputs. The `StaticDims` set lists the iteration dimensions with a static size. Such dimensions also return `true` to the `$dim.is_static()` method call.

```
set Dimensions subsetof Statements: ... end

set StaticDims subsetof Dimensions: ... end
```

Listing 4.1: Dimension Sets Declaration

Dimension Implementation. Listing 4.2 defines the `dim_kind` generic choice that specifies how to implement iteration dimensions. An iteration dimension d of size n may be implemented as:

- a regular counted loop (`dim_kind` = `LOOP`),
- a parallel loop mapped to a block (`dim_kind` = `BLOCK`) or thread dimension (`dim_kind` = `THREAD`) of the GPU,
- a fully unrolled loop, by repeating its body n times in the generated code (`dim_kind` = `UNROLL`),

- a fully vectorized loop, by replacing instructions iterated over d by vector instructions of width n (`dim_kind` = `VECTOR`).

```
choice enum dim_kind($dim in Dimensions):
  // $dim is a regular for loop.
  value LOOP:
  // $dim is a hardware block dimension.
  value BLOCK:
    requires "$dim.is_parallelizable()"
  // $dim is a hardware thread dimension.
  value THREAD:
  // $dim is totally unrolled.
    requires "$dim.is_static() && $dim.is_parallelizable()"
  value UNROLL:
    requires "$dim.is_static()"
  // $dim is vectorized.
  value VECTOR:
    requires "$dim.is_static() && $dim.is_parallelizable()"
end
```

Listing 4.2: Dimensions Decisons Encoding

`THREAD`, `UNROLL` and `VECTOR` decisions only apply to static iteration dimensions. In the case of `THREAD`, this is to ensure we do not exceed the maximal number of thread supported by CUDA in a block. For `UNROLL` and `VECTOR` this is for the code generator to know how much to replicate the body of the loop or which vector instruction width to use. `BLOCK`, `THREAD` and `VECTOR` also require the loop to be parallelizable, that is, to be free of loop-carried dependencies.

Strip-Mining. As explained in Section 4.2.1, the kernel representation offers strip-mining possibilities by splitting a single logical dimension of the original problem into multiple iteration dimensions, with a size fixed by decisions.

```
choice integer size($dim in StaticDims):
  "$dim.possible_sizes()"
end

set LogicalDims: ... end
set StripedDim($logical_dim in LogicalDims) subsetof Dimensions: ... end
set StripDims($logical_dim in LogicalDims) subsetof StaticDims: ... end

choice counter strip_mining_factor($logical_dim):
  forall $dim in StripDims($logical_dim):
    mul size($dim)
end

require forall $logical_dim in LogicalDims:
  strip_mining_factor($logical_dim) =
    "$logical_dim.possible_strip_minings()"
```

Listing 4.3: Dimensions Decisons Encoding

Listing 4.3 shows the definitions that support strip-mining decisions. It first defines a `size` generic choice that specify the size of static dimensions. This choice takes values in a list of possible sizes. For

a dimension that is not part of a strip-mined dimension, this list contains a single value. Otherwise, it contains the potential tiling factors.

Next, Listing 4.3 shows the definition of three sets (`LogicalDims`, `StripedDim` and `StripDims`), a counter (`strip_mining_factor`) and a constraint to limit the strip-mining factor of dimensions to valid values. The `LogicalDims` set lists logical dimensions, while `StripedDim` and `StripDims` list the dimensions that compose a given logical dimension.

Let d be a logical dimension of size n composed of dimensions $d_0, \ldots, d_k$ where d_0 is the dimension with the outermost index and d_k the dimension with the innermost index. The content of `StripedDim` and `StripDims` differ depending on if n depends on the kernel input.

- If n depends on the kernel inputs, then `StripedDim` $= \{d_0\}$ and `StripDims` $= \{d_1, \ldots, d_k\}$. The size of $d_1, \ldots d_k$ is fixed by the `size` choice, while the size of d_0 is inferred at code generation from the size of other dimensions. The `strip_mining_factor` counter computes the strip-mining factor of d_0 and limits it to the divisor of size(d_0) listed by `$logical_dim.possible_strip_minings()`.
- If n is static, then `StripedDim` is empty and `StripDims` contains all dimensions. In that case, the `size` choice specifies the size of all dimensions and `strip_mining_factor` computes the total size of $d_0 \times \cdots \times d_k$. We set `$logical_dim.possible_strip_minings()` to n so that the product of the dimensions size is equal to the size of the logical dimension.

4.3.2 Ordering, Fusion and Nesting

We now explain how we specify the control flow structure of the generated code. As detailed in Section 4.3.1, we implement iteration dimensions as loops. These loops may be sequential, parallel, unrolled or vectorized but all have the same basic structure: a body composed of statements that is executed for each iteration of the loop.

We first define the `order` choice that specifies the sequential order, nesting and fusion of loops and instructions. We then give the constraints that ensure that the generated control flow respects the semantics of the kernel. Finally, we explain how we represent the iteration space of each instruction.

Order Choice. We define a *statement* as either an instruction or an iteration dimension. We further define the `order` generic choice between pairs of statements as an antisymmetric relation that dictates the sequential ordering and the nesting of statements, as well as the loops fusion. Listing 4.4 gives the definition of the `Statements` set and the `order` choice, with `Instructions` and `Dimensions` subsets of `Statements`. Let x, y be two statements. The order between x and y can take the following value.

- `BEFORE` indicates that x executes before y.
- `AFTER` is the antisymmetric counterpart of `BEFORE`.
- `IN` indicates that x is an iteration dimension with y nested inside the body of x.
- `OUT` is the antisymmetric counterpart of `IN`.
- `MERGED` indicates x and y are iteration dimensions implemented as fused loops.

For example, Figure 4.5 shows the effects of setting the relative order of two iteration dimensions x and y to either `BEFORE`, `OUT` or `MERGED`. The latter case assumes that x and y have the same size. Note that the relative placement of `do_x_body` and `do_y_body` is not fixed by the sole order between x and y. The only constraint here is that they must be inside their respective loops.

The `order` choice gives the flexibility to perform several control flow optimizations. In particular, it can encode the following transformations.

- It encodes loop invariant code motion and re-materialization by setting the relative order of a statement and a dimension to `BEFORE`, `IN` or `AFTER`.
- It encodes loop interchange, by setting their relative order to `IN` or `OUT`.
- It encodes loop distribution and fusion by setting their relative order to `BEFORE`, `AFTER` or `MERGED`.
- It allows reordering statements by setting their relative order to `BEFORE` or `AFTER`.

```
set Statements: ... end
set Instructions subsetof Statements: ... end
set Dimensions subsetof Statements: ... end

choice enum order($lhs in Statements, $rhs in Statements):
  // $lhs executes before $rhs
  value BEFORE:
  // $rhs executes before $lhs
  value AFTER:
  // $lhs is nested inside $rhs
  value IN:
  // $rhs is nested inside $lhs
  value OUT:
  // $lhs and $rhs are fused dimensions
  value MERGED:
  antisymmetric:
    BEFORE <-> AFTER
    IN <-> OUT
end
```

Listing 4.4: Control-Flow Related Definitions

```
for i in 0..size(x):
  do_x_body()
for j in 0..size(y):
  do_y_body()
```

(a) x Ordered Before y

```
for i in 0..size(x):
  for j in 0..size(y):
    do_x_body()
    do_y_body()
```

(b) x Ordered Outer y

```
# size(x) == size(y)
for i in 0..size(x):
  do_x_body()
  do_y_body()
```

(c) x Merged With y

Listing 4.5: Effect of Ordering on Loops

Ordering Constraints. Below are the constraints that ensure the order is coherent and respects data dependencies.

- The order is sequentially transitive.

$$order(x, y) = \texttt{BEFORE} \wedge order(y, z) = \texttt{BEFORE} \implies order(x, z) = \texttt{BEFORE} \tag{4.3}$$

- The statements nested inside a dimension inherit from order between the dimension and the statements that are not in the dimension. This ensures loops have a tree-like structure in implementations.

$$order(x, d) = \texttt{IN} \wedge order(y, d) \neq \texttt{IN} \implies order(x, y) = order(y, d) \tag{4.4}$$

- Instructions can only be before, nested inside or after other statements.

$$\forall x \in \texttt{Instructions}.\ order(x, y) \in \{\texttt{BEFORE}, \texttt{AFTER}, \texttt{IN}\} \tag{4.5}$$

- If a statement y depends on a value produced by x, then y is after x. This includes the case where y is an instruction that produced x but also the case where x reads the last value of a variable produced in the loop y.

$$x \text{ depends on } y \implies order(x, y) = \texttt{AFTER} \tag{4.6}$$

- Merged dimensions have the same order with other statements.

$$order(x, y) = \texttt{MERGED} \implies \forall z \in \texttt{Statements}.\ order(x, z) = order(y, z) \tag{4.7}$$

These constraints are only the most important to understand how `order` works. The encoding has many more constraints involving the order, for example to ensure only dimensions with the same `size` and `dim_kind` are merged or to ensure `BLOCK` dimensions are outermost and `VECTOR` dimensions are innermost.

Instructions Iteration Dimensions. We define a quotient set `IterationDims` that contains one representative of each class of merged iteration dimension nested outside an instruction. Initially, `IterationDims($inst)` contains the dimensions of the iteration space of `$inst`. When decisions nest `$inst` into a new class of dimensions, a representative of this class is automatically added to the set.

```
quotient IterationDims($inst in Insts) of $dim Dimensions:
  is_iter_dim = order($dim, $inst) is OUT / order is MERGED
  ...
end

choice half counter threads_per_blocks($inst in Instructions):
  forall $dim in StaticDims:
    mul size($dim) when:
      is_iter_dim($inst, $dim) is TRUE
      dim_kind($dim) is THREAD
end

require forall $inst in Instructions:
  threads_per_blocks($inst) <= "target.max_threads_per_block"
```

Listing 4.6: Instructions Iteration Dimensions Set

Listing 4.6 defines `IterationDims` as a set that contains one representative of each class of merged dimensions nested outside `$inst`. This listing also shows an example of how the decision space use this quotient to limit the size of dimensions nested outside instructions: the `threads_per_blocks` counter limits the size of thread dimensions nested outside an instruction so that it does not exceed the maximal number of threads supported by CUDA. We use the same approach to also limit the number of block dimensions, the number of thread dimensions and the number of times an instruction repeated in unrolled loops.

4.3.3 Block and Thread Dimensions

Supporting `BLOCK` dimensions is simple: constraints force them to be nested outside every other statement so that they spawn under the entire computation. They translate into GPU block dimensions that parallelize the entire computation. `THREAD` dimensions are more complex as threads can synchronize within a block. We first explain how we represent synchronizations in the decision space and then how we decide how to map iteration dimensions to hardware thread dimensions.

Thread Dimensions Structure. We encode thread synchronization by expressing multiple loop nests of thread dimensions in the decision vector. Each loop nest of thread dimensions represents a piece of code executed in parallel between two synchronization instructions. At code generation, we fuse the loop nests together and introduce synchronizations between them. This transforms the loop structure so that thread dimensions are outer parallel dimensions, matching the programing model of GPUs, while keeping the original execution order. If two loop nests have different number of iterations, we introduce *if* statements to disable part of the threads.

```
for i in 0..n:
  parallel_for j in 0..m:
    do_a()
  do_b()
  parallel_for j in 0..m:
    do_c()
```

(a) Loop Structure in the Decision Vector

```
parallel_for j in 0..m:
  for i in 0..n:
    do_a()
    synchronize_threads
    if j == 0:
      do_b()
    synchronize_threads
    do_c()
    synchronize_threads
```

(b) Generated Pseudo-Code

Listing 4.7: Translation from Thread Dimensions to Synchronizations

Listing 4.7 shows an example of how we generate parallel code for a given loop structure. It features a sequential outer loop j that contains two parallel loops executing instructions a and c, separated by an instruction b. In the generated code, the parallel loop becomes the outermost loop. We introduce synchronizations so that the execution order remains the same. An *if* statement ensure that b does not execute in parallel.

Mapping to Hardware Dimensions. Decisions specify how to map thread dimensions from the different thread loop nests to hardware thread dimensions. For this, we define the `thread_mapping` choice between pairs of dimensions in Listing 4.8. This antisymmetric choice on pairs of static dimensions (dynamic dimensions cannot be threads) gives the relative order of thread dimensions, with regard to the hardware dimensions they map to. Let x and y be two dimensions. `thread_mapping`(x, y) can take the following values.

- `NOT_THREADS` indicates that either x or y is not a thread dimension.
- `MAPPED` indicates that x and y are fused dimensions or two thread dimensions from different loop nests mapped to the same hardware dimension.
- `OUT` indicates that x and y are two thread dimensions mapped to different hardware dimensions and that the hardware dimensions of x is nested outside the hardware dimensions of y. Dimensions x and y may be in the same or in different loop nests.
- `IN` is the antisymmetric counterpart of `OUT`.

Constraints ensure that `thread_mapping` defines a total quasiorder on thread dimensions and that it respects the nesting specified by `order`. A quotient set counts the classes of mapped dimensions to ensure that their number does not exceed the amount of hardware thread dimensions.

```
choice enum thread_mapping($lhs in StaticDims, $rhs in StaticDims):
  // Either $lhs or $rhs is a not thread dimension.
  value NOT_THREADS:
  // Map $lhs and $rhs to the same hardware thread dimension.
  value MAPPED:
  // Map $lhs and $rhs to different hardware thread dimension, with
  // $lhs outer than $rhs.
  value OUT:
  // Map $lhs and $rhs to different hardware thread dimension, with
  // $lhs inner than $rhs.
  value IN:
  antisymmetric: OUT <-> IN
end
```

Listing 4.8: Thread Dimension Mapping Choice Definition

4.3.4 Point-to-Point Communication

As explained in Section 4.2.2, the kernel representation can express point-to-point communication between loop nests by mapping a dimension to another. Decisions can choose to implement the communication between two dimensions the following ways.

- Merging both dimensions removes the communication, as the producer and consumer instruction end-up in the same loop.
- Unrolling or vectorizing both dimensions allows the producer and consumer code to use different registers for each iteration of the loops. This way, the data produced can live for more than one iteration and reach the consumer loop.
- Mapping both dimensions to the same hardware thread ensures that matching iterations of the producer and consumer dimensions map to the same thread. This is similar to merging dimensions except that it introduces a synchronization instruction between them.
- Temporarily storing the data into an array that lives until the consumer loop reads it.

Temporary Array Generation. A trigger extends the kernel representation with a temporary array, a store instruction, and a load instructions to implement point-to-point communication when decisions rule out other options.

Point-to-point communication typically involves multiple pairs of dimensions in the producer and the consumer loop nest, but only pairs of dimensions that are not merged should appear in the temporary array. At the time the trigger introduces the array, the list of merged dimensions might not be specified yet. This makes it impossible to fix the size and layout of the array upfront.

Instead, a counter tracks the product of the size of non-merged dimensions and a second trigger exposes potential layouts for the array once the relevant loop fusion choices are fixed. We have a prototype running that exposes the layout as a regular decision. The results we present in the rest of this dissertation, however, are based on a version where the trigger generates one candidate per potential layout. It behaves as if it was instantiating a decision, but without the possibility for constraint propagators to detect if a layout is valid or not beforehand.

Applications. In most cases, it is possible to avoid point-to-point communication, by reusing the same dimensions in the loop nests of the different instructions. However, using separate dimensions for different instructions offers flexibility for the control structure of potential implementations. For example, consider the code in Listing 4.9a that produces values $(x_i)_{i<4}$ in a dimension d_0 and consumes them in a dimension d_1. The notation `x.i` indicates point-to-point communication between the iterations i of d_0 and d_1. This communication enables the following optimizations among others.

- It allows implementing the producer and consumer dimensions with different `dim_kind`, such as in the Listing 4.9b with the producer vectorized and the consumer unrolled. This is particularly important as CUDA only allows the vectorization of load and store instructions.
- It allows distributing the producer and consumer loops to hide the latency of the producer instruction as in Listing 4.9c.
- It allows copying a chunk of data to shared memory, as in Listing 4.9d. This is useful to limit the accesses to global memory if the code reuses the data.

```
# Dimension d0
for i in 0..4:
  x.i = load X[i]
# Dimension d1
for i in 0..4:
  y_i = x.i * 2
```

(a) Original Code to Implement

```
# Dimension d0 vectorized
x0, x1, x2, x3 = vload X[0..3]
# Dimension d1 unrolled
y0 = x0 * 2
y1 = x1 * 2
y2 = x2 * 2
y3 = x3 * 2
```

(b) Different `dim_kind` for d_0 and d_1

```
# Dimension d0 unrolled
x0 = load X[0]
x1 = load X[1]
x2 = load X[2]
x3 = load X[3]
# Dimension d1 unrolled
y0 = x0 * 2
y1 = x1 * 2
y2 = x2 * 2
y3 = x3 * 2
```

(c) Distributed Loops

```
# Allocate T in shared memory
T = allocate(int, 4, SHARED_MEM)
# Dimension d0, load X into T
for i in 0..4:
  x = load X[i]
  T[i] = x
# Dimension d1, read x from T
for j in 0..4:
  x = load T[i]
  y_i = x * 2
```

(d) Communication Through a Temporrary Array T

Listing 4.9: Different Implementations of Point-to-Point Communication

4.3.5 Memory Accesses

Listing 4.10 shows the definition of memory-related sets and generic choices. It declares three sets: `Arrays`, `MemInsts` and `AccessedArrays`. These sets list the arrays allocated in memory, the memory access instructions and the arrays accessed by each memory access respectively. The listing then defines two generic choices: `mem_space` that attributes a memory space of arrays and `cache` that indicates the level of cache to use for memory accesses.

Constraints ensure that cache directives are coherent with memory spaces. A counter also ensures that the size of arrays allocated in shared memory does not exceed the size of the shared memory.

We detailed our decision space for linear algebra kernel on GPUs. This decision space exposes size generic choices: `dim_kind`, `size`, `order`, `thread_mapping`, `mem_space` and `cache`. We now explain how we generate code that follows those choices for a fixed candidate.

4.4 Code Generation

This section explains how we generate GPU code from a fixed candidate. The code generation procedure works in four steps, each detailed in a dedicated section. First, the search procedure annotates statements and arrays with the decisions that relative to them (Section 4.4.1). It then builds a tree encoding the loop structure hierarchy (Section 4.4.2) and generates variables to hold the values produced by instructions (Section 4.4.3). Finally, it prints the CUDA code following the loop tree (Section 4.4.4).

```
set Arrays: ... end
set MemInsts subsetof Instructions: ... end
set AccessedArrays($inst in MemInsts) subsetof Arrays: ... end

choice enum mem_space($array in Arrays):
  value GLOBAL:
  value SHARED:
end

choice enum cache($inst in MemAccesses):
  value L1:        // Use L1+L2 caches
  value L2:        // Use L2 cache
  value READ_ONLY: // Use read-only cache
  value NONE:      // Do not use caches
end
```

Listing 4.10: Memory Placement and Cache Decisions

4.4.1 Statements and Arrays Annotation

The first step to generate code is to extract information from the fixed decision vector to annotate statements and arrays. This step adds the following annotations.

- It annotates instructions with cache directives and vectorization factors.
- It computes groups of merged iteration dimensions and annotates them with their common size and `dim_kind`.
- It computes groups of thread dimensions mapped to the same hardware thread, and annotates it with the corresponding hardware dimension.
- It annotates arrays with their size and memory space. Note that in fixed candidates, all arrays have a fixed layout and a fixed set of dimensions. The size of an array is thus just the product of the size of its dimensions.

This step is also the occasion to implement the computation of addresses for linear accesses to tensors. To this end, the code generation procedure translate the pointer increments along iteration dimensions into induction variables. It then annotates the groups of merged dimensions with the induction variables they must compute.

4.4.2 Control Flow Tree

The second step of code generation is to build a tree, that encodes the nesting structure and the sequential order of statements. The root of this tree represents the entire program, internal nodes represent loops and leaves represent instructions. The children of a node are the statements directly nested inside the loop (or in the whole program) it represents. A node lists its children in sequential order.

Initial Tree Construction. To build the tree, we define an order $\prec$ on statements that indicates if a statement starts executing before another.

$$x \prec y \iff \texttt{order}(x, y) \in \{\texttt{BEFORE}, \texttt{OUT}\} \tag{4.8}$$

We then use $\prec$ to sort the instructions and one representative of each group of merged dimensions. Applied to that subset of statements, $\prec$ is the total order that corresponds to the pre-order traversal of the control flow tree. Algorithm 2 shows the algorithm that builds the tree. It iterates over the sorted list of statements S and maintains a pointer to the current inner most loop x. To simplify notations, we consider the entire kernel as the body of a loop of size one. For each statement $s \in S$, the algorithm tests if $\texttt{order}(s, x) = \{\texttt{IN}\}$. If that is not the case, it walks through the ancestors of x until it finds a loop that is nested outside s and updates the pointer x. It then appends s to the body of x.

Algorithm 2: Control Flow Tree Construction

```
Data: a list S of statements ordered by ≺
Result: a control flow tree
T ← empty tree;
x ← T.root;
foreach s ∈ S do
    while s is not nested in x do
        x ← x.parent;
    end
    y ← new tree node labeled by s;
    x.children.push_back(y);
    if s is a dimension then
        x ← y;
    end
end
return T
```

Parallel Dimension Erasure. After building the tree, the code generation erases parallel dimensions to obtain the code of a single thread. Since instruction annotations already contain the vectorization factor, so vector instructions are redundant. It keeps the order and size block and thread dimension aside to generate the code that launches the CUDA kernel. When erasing thread dimensions, it inserts thread synchronization instructions after each nest of thread dimensions. It also inserts *if* condition to disable the hardware thread dimensions that are not used by the current loop nest of thread dimensions. Specifically, it disables threads that have an index greater than 0 along an unused hardware thread dimensions.

4.4.3 Variables Generation

The third step of code generation is to allocate variables that hold the values produced and consumed by instructions. This only concerns the values stored in registers, as the point-to-point communications that go through memory are already lowered into load and store instructions to buffers during the decision phase. The register allocation is performed by the CUDA driver. This steps can thus use an arbitrary number of variable names.

Relation Between Variables and Instructions. Due to loop unrolling and vectorization, a single instruction of the kernel representation may store its result in multiple variables. Indeed, an unrolled loop creates multiple instances of the same instruction, each with its own result variable, while vectorized memory accesses store their result in multiple variables at once.

Conversely, we implement a loop-carried variable x, initialized with instruction y, by storing the result of x and y at the same location. Thus, different instructions or different instances of the same instruction may store their result in the same variable.

Listing 4.11 gives an example of kernel to implement and the pseudo code of a potential implementation, with the variables name assigned. The kernel contains three variables x, y, z, x is used in a different loop nest with point to point communication and z is a loop carried variable initialized with y. The potential implementation assumes both dimensions d_1 and d_2 are unrolled. Variable x is assigned a different name for each iteration of d_1 in order to allow point to point communication with d_2. Variables y and z are merged into a single one so that the loop carried variable is correctly initialized.

Instruction to Variable Mapping. When assigning the variables that hold the return values of instructions, the code generator iterates over instructions in sequential order. For each instruction i, it computes the set of dimensions X along which i is instantiated. That is the set of dimensions in the iteration space i that are either unrolled or vectorized and do not carry the instruction result across iterations.

```
for d0 in 0..4
  for d1 in 0..2:
    x(d1) = load A[(d0, d1)]
  y = 0
  for d2 in 0..2:
    z = phi(y, z) + x(d2)
```

(a) Kernel to Implement

```
for d0 in 0..4:
  x0 = load A[4*d0 + 0]
  x1 = load A[4*d0 + 1]
  y = 0
  y += x0
  y += x1
```

(b) Pseudo-Code of a Potential Implementation

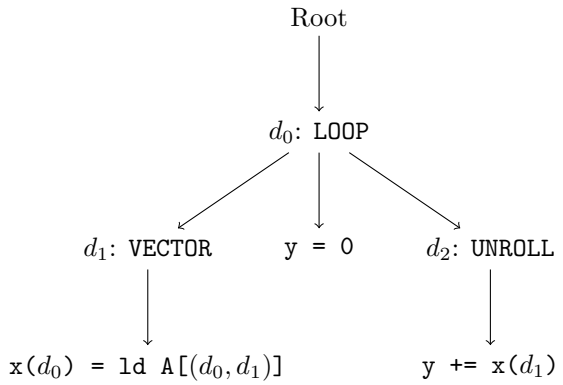
Listing 4.11: Example of Variable Names Assignation

It then assigns a variable to each iteration in the product of the dimensions of X. If the variable is a loop-carried variable, it reuses the variables assigned to the initialization instruction. Otherwise, it introduces new variables.

4.4.4 Code Printer

The code generator prints the code of a single GPU thread. It targets the PTX instruction set architecture, which is akin to assembly but with an infinite number of registers. PTX offers a finer control over the final executable than CUDA and avoids the cost of compiling CUDA code to PTX.

The generated code starts with the function declaration, then the declaration of the variables and the buffers, the loading of the parameters into registers and finally the body of the function. All buffers are allocated statically, since the constraints ensure their size is known at code generation. Printing the body of the function is simply a matter of walking the control flow tree and outputing the loop headers and footers and the instructions. The code generator walks unrolled loops once for each iteration, to instantiate their code. Figure 4.4 shows an example of control flow tree with the corresponding PTX code. The control flow tree contains a dimension d_0 implemented as a for-loop d_0, a vectorized dimension d_1 replaced with a vector instruction in the PTX and an unrolled dimension d_2 whose body is instantiated in the PTX.



(a) Control Flow Tree

```
// For-loop d0 header.
mov.i32 %d0, 0;
mov.i64 %a0, %A;
D0:
  // Induction variable
  add %a0, %a0, 16;
  // Vectorized d1.
  ld.global.v2.f32
      {%x0, %x1}, [%a0];
  mov.f32 %y, 0;
  // Unrolled d2.
  add.rn.f32 %y, %x0, %y;
  add.rn.f32 %y, %x1, %y;
  add.i32 %d0, 1;
  // For-loop d0 footer
  setp.lt.i32 %p, %d0, 4;
  @%p bra.uni D0;
```

(b) Generated PTX Code

Figure 4.4: PTX Code Generation from a Control Flow Tree

The code generator can also print C code that executes on the CPU, in order to call the CUDA driver and launch the computation the GPU with the right number of blocks and threads. This is only used to return the best implementation found by a search algorithm to the user. During the search, the code

generator makes the call to the CUDA driver itself in order to avoid the cost of compiling and linking the C code.

4.5 Discussion

This section reviews the specificities of our candidate representation for linear algebra kernels on GPUs, compared to other representations of possible optimizations in the literature. Section 4.5.1 first discusses the expressiveness of our representation and how its design applies to the representation of transformations with candidate representations in general. Section 4.5.2 then shows that our claim that candidates accurately represent valid decisions is well funded. Finally, Section 4.5.3 explores the relation between our representation and the polyhedral representation of loop nests.

4.5.1 Expressiveness of the Decisions Space

The first quality of a candidate representation is its expressiveness, and in particular the code transformations it supports. In the context of candidates, supporting a transformation is the capacity to represent an implementation with and without the transformation applied in the same implementation space.

We first recall the transformations our candidate representation supports. We then compare our candidate representation with annotation based code transformations and discuss our encoding of the control flow structure. Finally, we take a step back from our domain of application –linear algebra on GPUs– to discuss the insights we gained about the design of candidate representations in general.

Supported Transformations. The decision space this chapter defines is sufficiently expressive to support many of the transformation required to generate optimized code on GPU. It supports strip-mining with the `size` choice, loop fusion and loop interchange with the `order` choice, loop unrolling, vectorization with the `dim_kind` choice, mapping of computations to the different level of parallelism with the combination of the `dim_kind` and `order` choices and orchestrating data movements across the memory hierarchy with the `cache` and `memory_space` choices.

Annotation Based Transformation Representation. All the generic choices except `order` are relative to a single object. They are similar to compiler directives that specify when to unroll, vectorize of parallelize a loop or where to allocate an array. There are entire frameworks based on this concept. For example, OpenMP [38] allows programmers to parallelize a program by adding directives in front of loops. Following our terminology, an OpenMP candidate would be an OpenMP program with a list of possible directives at every place they may be added.

Another similar approach to expose options in a fixed template. This is, for example, the path taken by the ATLAS framework [39] on CPUs and the MAGMA framework on GPUs [40, 41]. These approaches, however, are limited to orthogonal decisions. In contrast, we expose interacting decisions, such as loop fusion and loop interchange, or strip-mining factors (acting on the size of arrays) and memory placement.

Control Flow Structure. The main originality of our decision space is its encoding of the control flow structure. The `order` choice simultaneously specifies the sequential order, the nesting and the fusion of instructions and loops, enabling a great flexibility in the code structures our candidates can represent. Indeed, it provides a local ordering information that can be tuned separately for each pair of statements.

This flexibility also applies to the classes of implementations we can represent. Indeed, the structure of loops only needs to be locally coherent before the candidate is fixed. Thus, the global loop structure does not need to be a tree for partially specified candidates. Table 4.2 shows an example for the domain of the `order` choice between three instructions a, b and c and two dimensions d_m and d_n that compute the outer product of arrays A and B into a variable c. Instruction a is nested in d_m, b in d_n and c in both dimensions. For a fixed candidate, this loop structure would be invalid. As Listing 4.12 shows, either d_n must contain d_m and a, or d_m must contain d_n and b. For partially specified candidates, this is not a problem as both solutions are still possible.

Statement	d_m	d_n
`a = load A[`d_m`]`	`IN`	`IN, BEFORE`
`b = load B[`d_n`]`	`IN, BEFORE`	`IN`
`c = a * b`	`IN`	`IN`
d_m	/	`IN, OUT`
d_n	`IN, OUT`	/

Table 4.2: Possible Orders Between Statements and Dimensions for the Outer Product Kernel

```
for i in 0..m:    # d_m
  a = load A[i]
  for j in 0..n: # d_n
    b = load B[j]
    c = a * b
```

(a) d_n and b Nested Inside d_m

```
for j in 0..n:    # d_n
  b = load B[j]
  for i in 0..m: # d_m
    a = load A[i]
    c = a * b
```

(b) d_m and a Nested Inside d_n

Listing 4.12: Possible Implementations for the Outer Product Kernel

The ability a superposition of implementations without instantiating the code itself –demonstrated here by the superposition of potential loop structure– is one of the key benefits or our encoding. This contrasts with other approaches such as Equality Saturation [22] that keep the different versions of the code.

Insights on the Design of Candidate Representations. In theory, candidate representations could support any transformation. We just need a boolean choice every place where it could apply, to enable or disable it. However, this approach causes several problems:

- Constraints will be complex, making propagators unable to prune invalid values before a candidate is fully specified. This makes it difficult to find any valid implementations, let alone an efficient one.
- Heuristics will not be able to model the interaction between transformations.
- Choices will need to consider transformations on the code generated by other transformations. Computing the transitive closure of all possible transformations may not be possible, or at least time and memory consuming.

For these reasons, this chapter adopted a different strategy to support transformations. Instead of focusing on the transformations themselves, the choices specify the structure and the parameters of the final code. The kernel representation specifies the semantics of the kernel and the computation to perform, while choices map them to the hardware. An example of this is the `order` choice, which specifies the relative order of statements rather than transformations such as loop interchange or loop invariant code motion.

To support a new transformation, the first step is to decompose it in atomic modifications of the generated code that only affect a couple of objects at most. For example, changing the memory space of an array will a) affect the allocation of the array and b) affect the cache directives of each instruction that access the array. We thus generate two distinct choices `mem_space` and `cache`. Similarly, the `order` choice decomposes control flow transformations into pairwise ordering decisions.

A corollary of encoding the implementation space with many small decisions that directly specify the structure and the properties of the final code is that the compiler does not need to be aware of high-level transformations to implement them. They are just a particular combination of decisions among others. This approach gives more flexibility on the transformations we support and makes it easier to combine them as they are all expressed using the same choices.

4.5.2 Effectiveness of Propagators

By providing information about the legality of implementations, constraints play a crucial role. However, constraint propagators working on fully specified implementations are not enough. Without constraint propagation on partially specified candidates, search algorithms would have to specify all decisions before deciding if a candidate is valid. Propagators implementing constraints should thus be strong enough to detect invalid decisions early.

We measured the effectiveness of propagators by evaluating the probability to end-up with only decisions leading to failed candidate when taking random decisions. Table 4.3 shows the results for the implementation spaces of the kernels listed below.

axpy: computes $z = \alpha.x + y$, where α is a scalar and x, y and z vectors of size $n = 2^{26}$.

matmul $m \times n \times k$: computes $C = A \cdot B$ when A and B are matrices of size $m \times k$ and $k \times n$.

strided matmul: computes matmul $1024 \times 1024 \times 1024$ with a stride of 32 between the elements of A.

batched matmul: computes 512 matrix multiplications of size $32 \times 32 \times 64$.

reuse matmul: is the same that batched matmul but reuses the same matrix B for all matrix multiplications.

Note that we consider matrices of different sizes for *matmul*, as a smaller size can restricts the potential strip mining factors. For more information about the implementation space of these kernels and the transformation they enable, see Section 6.2.

Kernel	Probability of Dead-End
axpy	$0.1\% \pm 0.02$
matmul $256 \times 256 \times 32$	$14\% \pm 0.7$
matmul $1024 \times 1024 \times 1024$	$2.8\% \pm 0.3$
strided matmul	$2.3\% \pm 0.3$
batched matmul	$17\% \pm 0.7$
reuse matmul	$31\% \pm 0.9$

Table 4.3: Probability to Backtracks Because of Constraints

The results show that the ratio of trials that needed to backtrack to find a valid implementation is below a few percent for simple kernels and below a third of trials for the most complex one. This is to put in perspective with the fact that the search algorithm we use, in Chapter 6, already backtrack when it detects the current candidate cannot lead to efficient code. The amount of backtracking due to failed candidates is negligible compared to other sources of backtracking. The impact of having suboptimal propagators on the exploration time is thus minimal. Still, we are working on the decision space and on the propagators to improve these numbers.

4.5.3 Relation With the Polyhedral Representation of Loop Nests

Section 3.6.3 already mentions the relation between candidates and polyhedral compilation [27, 28]. Both approaches encode an implementation as a vector of choices that respects logical constraints. In the case of polyhedral compilation, the vector contains the coefficients of an affine function that specifies the instructions schedule. We first discuss polyhedral code generators dedicated to GPUs and then explain how we could combine polyhedral approaches with candidates.

Polyhedral Code Generation for GPUs. Compilers based on polyhedral representations and targeting GPUs have shown promising results [42, 35]. In particular, Pouchet et al. a convex optimization space for loop transformations [34].The main limitation of their polyhedral representation compared to our approach is that they can only express the schedule of instructions while our candidates also expose orthogonal decisions, such as cache directives, memory placement, parallelization and unrolling decisions.

Existing polyhedral compilers rely on pre- or post-processing passes for these decisions. This makes it more complex to explore the space of potential implementations, to derive information early in the compilation process and to pick most important decisions first. For example, Diesel [43] is a recent framework from Nvidia applying an ILP-based scheduler to a specific domain, specializing it for each kernel with parameters specific to a GPU micro-architecture (tile sizes, mapping strategies), complemented with target-specific transformations (e.g., software pipelining, instruction-level and register-level optimizations). This framework reaches competitive performance levels matching or outperforming native libraries. However, this involves deep knowledge of the target architecture to manually tune the optimization strategy for each kernel.

Polyhedral Candidates. The polyhedral representation of loop transformation is actually compatible with the candidate approach. The decision vector could list the coefficients of a polyhedral schedule alongside other choices and use propagators to enforce affine constraints. In a future work, we hope apply this idea to support more kernels, such as convolutions, and more transformations, such as pipelining.

The difficulty of using affine schedule with our approach is to avoid losing too much expressiveness in the classes of schedules a candidate can represent. Affine schedules are more expressive when looking at final implementations but a partially specified affine schedule cannot encode arbitrary partial orders. Moreover, it is sometimes complex to understand the implication of a single coefficient on the final code before fixing other coefficients. This makes it harder to develop heuristics working on the decision vector.

4.6 Summary

This chapter defines a candidate representation for GPU linear algebra kernels. This representation supports both thread-local and global implementation decisions, including compositions of strip- mining, loop fusion, loop interchange, unrolling, vectorization, mapping to multiple levels parallelism, and orchestrating data movements across the memory hierarchy. It shows the ideas and formalism presented in Chapter 3 are sufficiently expressive to encode the interaction between complex transformations that fundamentally change the structure of the code.

The structure of the implementation space is simple enough that finding a valid implementation does not require a complex solver apart from constraint propagation. This confirms the assertions made in Chapter 3 that individual decisions domains accurately represent legal decisions.

Besides providing information about the legality of decisions, the candidate representation also helps understand what final implementation may look like. Chapter 5 demonstrates how we leverage this information to give a lower bound on the execution time achievable by a candidate. This model works early in the compilation process, even when most choices are still open. It is still unclear how the design of the decision space affects such heuristics.

The purpose of the candidate representation is to help search algorithms find more efficient implementation of linear algebra kernels for GPUs faster. Chapter 6 applies a simple search algorithm on top of our candidate representation and shows it allows for the generation of code that is competitive with hand-tuned libraries, and outperforms them in some cases.

The representation assumes that all loops are for-loops with data-independent bounds (i.e., loop bounds only depend on the input size) and that memory accesses are linear accesses to n-dimensional tensors. While this is sufficient for most dense linear algebra computations, we intend to broaden the class of computations the representation supports, in particular to convolutions. A promising research direction for that is to combine polyhedral compilation with the candidate formalism.

Chapter 5

An Optimistic Performance Model of Candidates

The benefits of using candidates come from the information they expose about the potential implementations they represent, and in particular about the best of these implementations. This chapter demonstrates how candidates can provide such information, even when only a few choices are fixed. It develops an analytical performance model that computes a lower bound $B(c)$ on the execution time of any implementation that derives from a candidate c. In the following, we denote $T(x)$ the execution time of implementation x on the hardware. The bound $B(c)$ respects the following equation.

$$\forall x \in [\![c]\!] \,.\, T(x) \geq B(c) \tag{5.1}$$

The model we provide is generic in the sense that it supports architectures with various depths of parallelism hierarchies and different resource constraints. It can also adapt to different decision spaces. The only choices it directly depends on are `order` and, to a lesser extent, `dim_kind`. Other choices impact the performance of individual statements, but do not affect the structure of the model.

Lower Bound. The first distinctive feature of our model is that it computes a lower bound for the execution time of implementations rather than trying to estimate the exact execution time. Of course, the bound should be as close as possible to the execution time; but in case of uncertainty, our model will adopt conservative abstractions instead of trying to guess a value.

This lower bound approach has two advantages. First, it avoids misleading a search algorithm with inaccurate information when the model is uncertain about the execution time. Instead, the model gives a conservative bound. This way, the search algorithm can reliably use the bound to avoid parts of the implementation space that only contain inefficient implementations.

Second, in presence of open choices, the idealistic performance achievable by a candidate is a proxy for the performance of the best implementation that derives from the candidate. Trying to accurately estimate the execution time of the best implementation would be impossible without enumerating all possible implementations. Extracting information about the average performance of implementations that derive from the candidate would be meaningless as the search algorithm only looks for the best ones.

Open Choices. The second distinctive feature of our model is that it works even when some implementation choices are left open. Generating and inspecting the code of all the implementations that derive from the candidate is infeasible due to the potentially large number of implementations to consider. Instead, the model works directly with the sets of potential decisions.

When facing open choices, the model strategy is to make conservative assumptions that minimize the execution time. The assumptions are local and independent for each hardware resource. Two assumptions may assume different values for the same choice or incompatible values for two different choices.

While making such inconsistant assumptions may reduce the accuracy of the model, it allows for efficiently computing a lower bound on the execution time without enumerating the implementations. This source of inaccuracy only affects candidates with open choices and it vanishes as more decisions are specified.

Contributions. This chapter defines a generic performance model that computes a lower bound for the execution time of any implementation that derives from a candidate. Its main contributions are the following.

- It demonstrates that it is possible to extract actionable information from candidate implementations, even with only a few choices fixed. The concepts developed to handle open decisions are not limited to the performance model: they could be applied to define other heuristics on partially specified candidates.
- It defines a performance model that gives a lower bound on the execution time of any implementation that derives from a candidate. This performance model is not specific to GPUs. It can be instantiated for other architectures with different bottlenecks and a different parallelism hierarchy.
- It instantiates this model for Nvidia GPUs and evaluates the pertinence of the resulting bound. It shows that the model allows for pruning candidates early in the compilation process and reducing the implementation space size by several orders of magnitude.

Outline. Section 5.1 first gives an overview of how the performance model works. Section 5.2 then dives into the details of how the model computes the latency of a single thread. Next, Section 5.3 instantiates the generic performance model for CUDA GPUs and Section 5.4 evaluates it on actual candidates. Finally, Section 5.5 debates the specificities of the model and compares it to the literature while Section 5.6 concludes the chapter.

5.1 Model Overview

This section presents a generic version of the performance model that makes minimal assumptions about the hardware and the statement-specific choices the decision space exposes. Let c be a candidate implementation with a set of instructions $\mathcal{I}$ and set of dimensions $\mathscr{D}$. Let $\mathscr{S} := \mathcal{I} \cup \mathscr{D}$ be the set of statements. We denote $x \in [\![c]\!]$ an implementation x that derives from c. The goal of the model is to provide a lower bound $B(c)$ for the execution time of any implementation $x \in [\![c]\!]$, for a fixed value of the kernel parameters. Let $T(x)$ be the execution time of $x \in [\![c]\!]$. Since $B(c)$ is a lower bound for $T(x)$, it satisfies the following equation.

$$\forall x \in c.\, T(x) \geq B(c) \tag{5.2}$$

To compute a lower bound $B(c)$ on the execution time on any implementation $x \in [\![c]\!]$ that derives from C, the performance model processes in four steps.

1. It first analyzes every statement in isolation to obtain a lower bound for its latency and on the hardware resources it consumes.
2. It then accounts for the total usage of hardware resources generated by the statements enclosed in each loop to compute a lower bound for the execution time of a single iteration of that loop.
3. It combines the bounds for the execution time of loop iterations, the data dependencies between instructions and the `order` decisions between statements to compute a lower bound for the latency of a single thread.
4. It finally combines the latency of a single thread, the usage of hardware resources generated by statements at each level of the parallelism hierarchy and the parallelism available at these levels to compute the final bound for the execution time of any implementation that derives from the candidate.

The third step, computing a lower bound for the latency of a thread, is by far the most complex. We assume that this model exists for now and detail it later in Section 5.2. Section 5.1.1 defines the abstraction of the hardware the model relies on. Section 5.1.2 presents the analysis of individual statements, Section 5.1.3 the analysis of loop iterations and Section 5.1.4 explains how we compute the final bound, taking into account the parallelism hierarchy.

5.1.1 Hardware Abstraction

We define a hardware abstraction that supports architectures with various depths of parallelism hierarchy and different resource constraints. This abstraction allows us to apply the model to different architectures by only providing the depth of the parallelism hierarchy k, the list of hardware resources to model $\mathcal{R}$, a function that computes the individual impact of statements on hardware resources and a function that computes the number of threads that can execute simultaneously at each level of the parallelism hierarchy.

Parallelism Hierarchy. A parallelism hierarchy of depth k defines a parallelism level for each $i \in [\![0, k]\!]$. Parallelism level 0 sequentially executes a single thread. Then, each level $i > 0$ instantiates the computation at level $i-1$ a fixed, implementation-dependent, number of times. For example, on GPUs, parallelism level 0 corresponds to a single thread, parallelism level 1 to a block of threads and parallelism level 2 to the computation on the entire GPU.

For an implementation x and $i > 0$, we denote η_i the number of instances of the computation of level $i-1$ that execute at level i. We also denote μ_i the number of such instances that can execute in parallel. The amount of parallelism η_i available at level i, depends on the size of parallel dimensions that map to level i in x, while μ_i typically depends on the amount of memory and registers allocated by computations at level $i-1$.

Some levels of parallelism induce synchronizations at the beginning and at the end of each dimension mapped to these levels. When a thread encounters a synchronization instruction, it waits for all other threads in the same instance of the corresponding parallelism level. For example, this is the case for dimensions mapped to a thread dimension on GPUs, as explained in Section 4.3.3.

Hardware Resources. The set of hardware resources $\mathcal{R}$ lists the bottlenecks considered by the model. These can be resources local to a single core, such as floating point units, or global resources, such as the memory bandwidth. For $i \in [\![0, k]\!]$ and $r \in \mathcal{R}$, we denote $\texttt{resource}(i, r)$ the maximal amount of resource r parallelism level i can process per cycle.

$$\texttt{resource} : [\![0, k]\!] \times \mathcal{R} \to \mathbb{R}^+ \tag{5.3}$$

Each instruction in an implementation has a *latency* and a *usage* of each of the resources. The latter counts the number of units of a resource an instruction consumes to execute.

5.1.2 Local Statements Analysis

The first step of the performance model is to analyze statements in isolation, in order to compute the latency and the usage they induce on each hardware resource. This step defines two functions, `usage` and `iter_usage_overhead`. The first one indicates the usage of a single instance of each statements, while the second indicates the overhead caused by a single iteration of the implementation of a dimension (e.g. the overhead of a loop header if the dimension is implemented as a regular counted loop).

$$\texttt{usage} : \mathscr{S} \times \mathcal{R} \to \mathbb{R} \tag{5.4}$$

$$\texttt{iter\_usage\_overhead} : \mathscr{D} \times \mathcal{R} \to \mathbb{R} \tag{5.5}$$

The model also computes the latency of instructions and the minimal latency of a single iteration of a dimension, independently of the statements its body may contain.

$$\texttt{inst\_latency} : \mathcal{I} \to \mathbb{R} \tag{5.6}$$

$$\texttt{iter\_min\_latency} : \mathscr{D} \to \mathbb{R} \tag{5.7}$$

For example, $\texttt{iter\_min\_latency}(d)$ can be the time required to increment the loop index and test if it has exceeded the size of the loop. In the case of our decision space for GPUs, it depends on the possible decisions for the `dim_kind` choice.

For an iteration dimension $d \in \mathscr{D}$, $\texttt{usage}(d)$, $\texttt{iter\_usage\_overhead}(d)$ and $\texttt{iter\_min\_latency}(d)$ only account for the overhead per thread of the loop implementing d. They do not account for the statements nested inside d.

Modeling Open Decisions. Although the statement analysis is target-dependent, the strategy to handle open decisions remains the same. The model enumerates potential combinations of decisions relative to a particular statement and takes the minimal usage or latency over all alternatives. The decisions that lead to the minimum can differ for each statement and each resource.

This strategy works because the number of decisions relative to each statement is usually relatively small. In the rare cases where a statement's performance depends on a large number of choices, we use ad hoc models that conservatively estimate the behavior of the statement; the very simple cache model presented in the following is one such example. In the worst case, these models can just return zero.

5.1.3 From Usage to Latency

After computing the usage induced by individual statements, the model analyzes how they affect the execution times between instructions. If the statements that execute between two instructions $a, b \in \mathcal{I}$ induce a usage u of resource $r \in \mathcal{R}$, then the execution time t between a and b respects the following equation.

$$t \geq \frac{u}{\texttt{resource}(0, r)} \tag{5.8}$$

Ideally, the model would consider the usage of the hardware resources between each pair of statements. In practice, there are too many such pairs to consider all of them. Instead, the model only considers the usage between the beginning and the end of loop iterations. This reduces the granularity at which we consider bottlenecks but makes the model tractable. We first introduce the concept of loop level in order to represent the loops of potential implementations, even when the loop structure imposed by the candidate is not fixed yet. We then explain how the model counts the number of instances of a statement inside a loop level. Finally, we discuss how we compute a lower bound for the latency of any iteration of a loop level.

Loop Level

The model accounts for the impact of the usage induced by statements at the granularity of the loops' bodies. In a fixed candidate, a loop corresponds to a group of merged dimensions. This definition, however, makes it impossible to represent the body of the innermost loop of a loop nest when the nesting order of the dimensions that compose the loop nest is left unspecified. For example, let $d_1, d_2 \in \mathscr{D}$ be two dimensions that are nested with each other in a not-yet-specified order. Then, neither d_1 nor d_2 can be considered as the innermost loop by the model. This is a problem as the body of the innermost loop is guaranteed to be executed $\texttt{size}(d_1) \times \texttt{size}(d_2)$, therefore having a bigger impact on performance than the body of the dimensions d_1 and d_2 taken in isolation. We thus introduce the concept of loop level to represent the innermost iteration of loop nests.

Definition 5.1 (Loop Level)**.** A *loop level* is a set $L \subseteq \mathscr{D}$ of dimensions that are either merged or nested with each other. We denote $\mathbb{L}$ the set of all loop levels.

$$\mathbb{L} := \{L \subseteq \mathscr{D} \mid \forall d_1, d_2 \in L.\ d_1 = d_2 \vee \texttt{order}(d_1, d_2) \subseteq \{\texttt{MERGED}, \texttt{IN}, \texttt{OUT}\}\} \tag{5.9}$$

For a set of statements $S \subseteq \mathscr{S}$, we define the restriction $S|_L$ of S to L as the statements of S necessarily nested in all dimensions of D.

$$S|_L := \{s \in \mathscr{S} \mid \forall d \in L.\ \texttt{order}(d, s) = \{\texttt{OUT}\}\} \tag{5.10}$$

We define the *body* $\mathscr{S}|_L$ of L as the set of all statements restricted to L.

A loop level $L \in \mathbb{L}$ represents the innermost loop in the loop nest composed of the dimensions $d \in L$. If all the dimensions of L are merged with each other, L represents a single loop. The empty loop level $L = \emptyset$ represents the entire computation as $\mathscr{S}|_\emptyset = \mathscr{S}$. For example, Listing 5.1 shows the possible implementations of a kernel with two dimensions d_0 and d_1 such that d_0 contains instructions x and z and d_1 contains instructions y and z. The loop level $L = \{d_0, d_1\}$ represents the innermost the loop nest composed of d_0 and d_1, independently of the nesting order. The body $\mathscr{S}|_L = \{z\}$ of L contains the instructions that are common to both d_0 and d_1.

The number of loop levels may be exponential in the number of iteration dimensions. Sections 5.1.4 and 5.2.3 show that in practice, the performance model only considers of few selected loop levels.

```
for i in 0..size(d_0):
  x = 2 * i
  for j in 0..size(d_1):
    y = 3 * j
    z = x + y
```

```
for j in 0..size(d_1):
  y = 3 * j
  for i in 0..size(d_0):
    x = 2 * i
    z = x + y
```

Listing 5.1: Possible Implementations of a Loop Nests with Permutable Dimensions

We extend `iter_usage_overhead` and `iter_min_latency` to work on loop levels instead of dimensions. For every $L \in \mathbb{L}$, the usage $\texttt{iter\_usage\_overhead}(L)$ and the latency $\texttt{iter\_min\_latency}(L)$ of L are the minimal usage and the minimal latency over all the dimensions $d \in L$ that compose L.

$$\forall r \in \mathcal{R}.\, \texttt{iter\_usage\_overhead}(L)(r) := \min_{d \in L} \texttt{iter\_usage\_overhead}(d)(r) \tag{5.11}$$

$$\texttt{iter\_min\_latency}(L) := \min_{d \in L} \texttt{iter\_min\_latency}(d). \tag{5.12}$$

In both cases, we consider the minimum over an empty set to be equal to 0.

Counting Statement Instances

We now explain how the model exploits the information available in the domain in order to computes a lower bound $N_L^i(s)$ on the number of instances of a statement $s \in \mathscr{S}$ that are executed at parallelism level $i \in [\![0, k]\!]$ within a single iteration of loop level $L \in \mathbb{L}$.

The first difficulty is to account for merged statements. In a fixed candidate, merged statements form equivalence classes. In the general case, however, statements may end up in different classes depending on further decisions, and the final number of classes may vary. The model computes a lower and an upper bound for the number of classes by defining two sets $\underline{S}$ and $\overline{S}$ that respectively exhibit *at most* and *at least* one statement of each class for each implementation $x \in [\![c]\!]$.

On a fixed candidate, $\underline{S} = \overline{S}$ exhibits exactly one statement per class of merged statements. In the general case, however, finding a set $\underline{S}$ of maximal size is NP-hard.[1] Instead, we define a conservative approximation of $\underline{S}$. Let $<$ be an arbitrary total order on statements. For example, it can be the order in which the kernel representation lists the statements. The model defines $\underline{S}$ and $\overline{S}$ for $S \subseteq \mathscr{S}$ as follows.

$$\overline{S} := \{s \in S \mid \forall s' \in S.\, s' < s \implies \texttt{order}(s, s') \neq \{\texttt{MERGED}\}\} \tag{5.13}$$

$$\underline{S} := \{s \in S \mid \forall s' \in S.\, s' < s \implies \texttt{MERGED} \notin \texttt{order}(s, s')\} \tag{5.14}$$

The model then relies on these sets to bound the *size* of loop levels, that is, the number of iterations of the loop nests they represent. In fixed candidates, the size of dimensions is a function of the kernel parameters, which are known to the model. In the general case, however, the size of dimensions also depends on the `size` choice. The model thus computes a lower and an upper bound for the size of loop levels based on possible values of `size`. Specifically, it computes the lower bound $\texttt{size}_{min}(L)$ (respectively upper bound $\texttt{size}_{max}(L)$) on the size of $L \in \mathbb{L}$ as follows.

1. It replaces L by $\underline{L}$ (respectively $\overline{L}$) to account for merged dimensions.

2. It groups dimensions into logical dimensions when possible. Recall that logical dimensions are groups of dimensions that correspond to a single dimension of the original problem. Contrary to regular dimensions, they have a fixed size.

3. It computes the minimal (respectively maximal) possible size of each remaining dimension, based on the `size` choice. It then computes the product of all these values and of the size of the logical dimensions to obtain the total minimal (respectively maximal) size of L.

[1] It is possible to encode a graph coloring problem as the computation of $\underline{S}$: each node of the graph is a statement and $\texttt{MERGED} \in \texttt{order}(s_1, s_2)$ if and only if there is an edge between s_1 and s_2. Then each statements in $\underline{\mathscr{S}}$ is a color.

The model applies the same strategy to compute the least common multiple $\texttt{size}_{lcm}(L)$ of the potential sizes of $L \in \mathbb{L}$, based on the least common multiple of the potential sizes of each dimension $d \in \mathbb{L}$.

For $i \in [\![0, k]\!]$, let $\mathscr{D}_i \subseteq \mathscr{D}$ be the set of dimensions that may map to parallelism level i. The model computes the number of instances $N_L^i(s)$ of a statement $s \in \mathscr{S}$, in a loop level $L \in \mathbb{L}$ at parallelism level $i \in [\![0, k]\!]$ as the size of the loop nest L' composed of the dimensions that are nested outside s and inside L and that map to a parallelism level $j \leq i$.

$$N_L^i(s) := \texttt{size}_{min}\left(\{d \in L \mid \texttt{order}(s, d) = \{\texttt{IN}\} \wedge \exists j \leq i.\ d \in \mathscr{D}_i\}\right) \tag{5.15}$$

Note that equation (5.15) accounts for all the dimensions that *may* map to a parallelism level $j \leq i$, instead of the dimensions that *must* map to such a parallelism level j. Section 5.1.4 explains the justification for this choice.

Loop Level Iteration Latency

We now explain how the model computes a lower bound for $\texttt{latency}(i, L)$, the execution time of a single iteration of loop level L, based on the hardware resource available at parallelism level i. To this end, the model first defines $\texttt{iter\_usage}(i, L, r)$ as the usage of resource $r \in \mathcal{R}$ generated by a single iteration of L within parallelism level i.

$$\texttt{iter\_usage}(i, L, r) := \texttt{iter\_usage\_overhead}(L, r) + \sum_{s \in \underline{S|_L}} N_L^i(r) \times \texttt{usage}(s, r) \tag{5.16}$$

The model then defines the lower bound $\texttt{latency}(i, L, r)$ as the maximum of the latency incurred by the usage of each resource and the minimum latency of a loop iteration.

$$\texttt{latency}(i, L) := \max\left(\max_{r \in \mathcal{R}} \frac{\texttt{iter\_usage}(i, L, r)}{\texttt{resource}(i, r)},\ \texttt{iter\_min\_latency}(L)\right) \tag{5.17}$$

In the rest of this section, we only use `latency` at the loop level $L = \emptyset$, that represents the usage of the entire computation. Section 5.2, however, uses it at other loop levels to model the execution time of a single thread.

5.1.4 Global Bound

We now show how the model uses `latency` to compute a lower bound B_i on the execution time T_i of each parallelism level $i \in [\![0, k]\!]$. For this, we assume that a separate model, detailed in Section 5.2, gives a lower bound B_0 on the execution time of a single thread.

Let $i \in [\![1, k]\!]$. Let's assume for a moment the parallelism at levels $j > i$ fixed. Following the hardware abstraction of Section 5.1.1, the parallelism level i computes η_i instances of parallelism level $i - 1$, with μ_i instances at most executing in parallel. The model thus defines B_i as follows for fixed candidates.

$$B_i := \max\left(B_{i-1}.\left\lceil \frac{\eta_i}{\mu_i} \right\rceil,\ \texttt{latency}(i, \emptyset)\right) \leq T_i \tag{5.18}$$

The first term of the maximum accounts for the execution time of lower parallelism levels, while the second term accounts for the usage of the hardware resources available at level i.

In the general case, the exact value of η_i and μ_i is unknown except for fixed candidates. Computing the maximal value μ_i^{max} that μ_i can take is easy since it only depends on the minimal amount of memory and/or registers that level $i - 1$ allocates. Unfortunately, this approach fails with η_i. Indeed, independently minimizing η_i and B_{i-1} produces a bound that is far too inaccurate when the mapping of dimensions to parallelism levels is not fixed yet: dimensions that may map to both parallelism levels i and $j < i$ will be ignored.

Parallelism-Mapping Agnostic Bound. For $D \subseteq \mathscr{D}_i$, we denote B_{i-1}^D and η_i^D the values of B_{i-1} and η_i assuming that the dimensions that map to parallelism level i are exactly the dimensions of D (recall that $\mathscr{D}_i$ is the list of dimensions that may map to parallelism level i). With this definition, B_{i-1}^D is

maximal when D is minimal and η_i is maximal when D is maximal. We can then bound T_i by computing the minimum of the first term of equation (5.18) for all the possible values of D:

$$T_i \geq \min_{D \subseteq \mathscr{D}_i} B_{i-1}^D \cdot \left\lceil \frac{\eta_i^D}{\mu_i^{max}} \right\rceil . \tag{5.19}$$

Let $\eta_i^{\text{lcm}} := \texttt{size}_{lcm}(\mathscr{D}_i)$ be a common multiple of the possible values of η_i^D, across every possible value of D and every possible assignment of the `size` choice. Since the possible sizes of a dimension often are multiples of each other, this usually corresponds to the product of the maximal size of each dimension of $\mathscr{D}_i$. Then, for every $D \subseteq \mathscr{D}_i$, there exists $\alpha^D \in \mathbb{N}$ such that $\eta_i^{\text{lcm}} = \alpha^D.\eta_i^D$. We use α^D and the fact that $\forall k \in \mathbb{N}.\ k.\lceil x \rceil \geq \lceil k.x \rceil$ to rewrite equation (5.19) as follows.

$$T_i \geq \min_{D \subseteq \mathscr{D}_i} B_{i-1}^D \cdot \frac{\alpha^D.\eta_i^D}{\eta_i^{\text{lcm}}} \cdot \left\lceil \frac{\eta_i^D}{\mu_i^{max}} \right\rceil \tag{5.20}$$

$$\geq \min_{D \subseteq \mathscr{D}_i} B_{i-1}^D \cdot \frac{\eta_i^D}{\eta_i^{\text{lcm}}} \cdot \left\lceil \frac{\alpha^D.\eta_i^D}{\mu_i^{max}} \right\rceil \tag{5.21}$$

$$\geq \min_{D \subseteq \mathscr{D}_i} B_{i-1}^D \cdot \frac{\eta_i^D}{\eta_i^{\text{lcm}}} \cdot \left\lceil \frac{\eta_i^{\text{lcm}}}{\mu_i^{max}} \right\rceil \tag{5.22}$$

We then notice that $B_{i-1}^D.\eta_i^D$ is minimal when D is minimal. Indeed, let $D, D' \in \mathbb{L}$ such that $D \subseteq D'$. The assumptions the model makes about the mapping to parallelism levels of the dimensions $\{d \in D' \mid d \notin D\}$ when computing B_{i-1}^D do not impact the overhead of dimensions nor the individual usage and latency of statements. Indeed, they are computed based on independent assumptions in Section 5.1.2. Assumptions about a dimension $d \in D' \backslash D$ can only impact η_j for $j < i$ and the number of iterations of d at each parallelism level $j < i$. In the worst case, mapping the dimension to loop level j causes the entire computation to be sequentially repeated $\texttt{size}(d)$ times, either by having $j = 0$, by increasing the factor $\left\lceil \frac{\eta_j}{\mu_j} \right\rceil$ in (5.18) or by increasing usage at parallelism level j. Thus, for all $D, D' \in \mathbb{L}$,

$$D \subseteq D' \implies B_{i-1}^D \eta_i^D \leq B_{i-1}^{D'}.\texttt{size}(D' \backslash D).\eta_i^D \tag{5.23}$$

$$\leq B_{i-1}^{D'}.\eta_i^{D'} \tag{5.24}$$

Let $B_{i-1}^{\max}$ be the value of B_{i-1}^D with D minimal. The combination of equations (5.20) and (5.24) gives the following lower bound for B_i, where $\eta_i^{\min}$ is a lower bound for the value of η_i.

$$T_i \geq B_{i-1}^{\max} \cdot \frac{\eta_i^{\min}}{\eta_i^{\text{lcm}}} \cdot \left\lceil \frac{\eta_i^{\text{lcm}}}{\mu_i^{\max}} \right\rceil \tag{5.25}$$

$$\eta_i^{\min} := \texttt{size}_{min}\left(\{d \in \mathscr{D}_i \mid \forall j < i.\ d \notin \mathscr{D}_j\}\right) \tag{5.26}$$

This bound is independent of D but still assumes that the mapping of dimensions to levels above i is fixed. We now explain how the performance model recursively use this equation to bound the execution time of the entire kernel, and why the condition that parallelism level above i are fixed is respected.

Bound on the Execution Time of a Kernel. The performance model recursively applies equation (5.25) to compute a lower bound B_i on the execution time of each parallelism level $i \in [\![1, k]\!]$, assuming that all the dimensions map to a parallelism level $j \leq i$ when possible,and to the lowest possible parallelism level above i otherwise. When $i = k$, this assumption impose no constraints on the mapping of dimensions to parallelism levels. Thus B_k is a lower bound on the execution time of the entire computation, independently of any assumption on the mapping of dimensions to parallelism levels.

For each $i \leq k$, the assumption ensures that $B_{i-1} = B_{i-1}^{\max}$ in equation (5.25), and that the parallelism level of all the dimensions above i is fixed. The performance model can thus apply this equation to compute B_i. Moreover, the defintion of `latency`, in equation (5.15), already follows the assumption. We thus define B_i as follows.

$$B_i := \max\left(B_{i-1} \times \frac{\eta_i^{min}}{\eta_i^{\text{lcm}}} \times \left\lceil \frac{\eta_i^{\text{lcm}}}{\mu_i^{\max}} \right\rceil,\ \texttt{latency}(i, \emptyset)\right). \tag{5.27}$$

Equation (5.27) accounts for the parallelism and the usage at each level of parallelism $i \in [\![1, k]\!]$. We now explain how we compute a lower bound B_0 on the execution time of a single thread.

5.2 Single Thread Model

Section 5.1 explained how the performance model computes a lower bound on the execution time of a partially specified candidate. The explanation assumed that a bound B_0 on the execution time of a single thread was provided. We now explain how this bound is computed. Following the hypothesis of Section 5.1.4, we consider that dimensions map to the lowest parallelism level possible. Any dimension that may be sequential is thus considered sequential by the model. The exceptions are the dimensions that may be vectorized, as we consider vector dimensions as part of parallelism level 0.

Overview. The model expresses the execution time of a thread as the weight of the longest path in an *execution graph*, that represents all instances of each statement and the latency constraints between them. This graph is too big to build in practice, as it grows exponentially with the number of loops, but it makes a convenient abstraction to define the bound that the model computes and prove that it is indeed a lower bound for the execution time of any implementation $c' \in [\![c]\!]$ that derives from c.

The difficulty of computing a lower bound for the execution time of a thread is that the structure of loops may be only partially specified. In particular, it might be unclear in which iteration dimensions a statement will be nested, or how many instances of that statement the final code will execute. We solve this problem in four steps.

1. We define the execution graph G_c of a candidate c to account for all potential instances of statements.
2. We define the concept of *concrete* statement instance to represent the statement instances that are actually executed. This concept only makes sense for fixed candidates. We show that if c is a fixed candidate, the weight of any path in G_c that only visits concrete statement instances is a lower bound for the execution time of a thread.
3. We show that in the case where c is a fixed candidate (i.e. an implementation), any path in the execution graph G_c has a longer path that only visits concrete statement instances. This shows that we can compute B_0 from the weight of any path in G_c.
4. We extend the previous result to the case where c is a partially specified candidate. For that we show there exists an injection from G_c to the execution graph of any fixed candidate $c' \in [\![c]\!]$ that derives from c. This implies that any path in G_c has a longer path in the execution graph $G_{c'}$ of every $c' \in [\![c]\!]$. The weight of any path in G_c is thus a lower bound for the execution time of a thread for any implementation that derives from c.

As stated earlier, the graph G_c is far too big to be constructed explicitly. Instead, our algorithm builds a conservative abstraction of the paths in G_c, and computes the longest path in this abstraction.

Outline. Section 5.2.1 first defines the concept of *execution point* to represent the potential points in the execution trace of implementations. This concept refines the notion of statement instance. Section 5.2.2 then defines the execution graph G_c of a candidate c and proves that the weight of any path in G_c is a lower bound on the execution time of a thread. Finally, Section 5.2.3 explains how the model builds a conservative abstraction of the paths in G_c in order to compute B_0.

5.2.1 Execution Points

This section defines the concept of *execution point* in order to represent points in the execution trace of implementations. To that end, it first defines the concept of *program point* to represent points in the code of implementations. It then defines the *program order* to specify the order in which program points appear in the code of implementations. Finally, it defines an execution point as an instance of a program point.

Program Points

Let c be a candidate. A *program point* of c represents a point in the code of the implementations that derive from c. It can be an instruction (in which case it represents the point just before the instruction), the beginning of the iteration of a loop level or the end of the iteration of a loop level.

Definition 5.2 (Program Point)**.** Let $L \in \mathbb{L}$ be a loop level. We define E_L the *entry point* of L and X_L the *exit point* of L. Together with the instructions, the entry and exit points of loop levels from the *program points*. We denote $\mathbb{P}$ the set of all program points.

$$\mathbb{P} := \mathcal{I} \cup \{E_L \mid L \in \mathbb{L}\} \cup \{X_L \mid L \in \mathbb{L}\} \tag{5.28}$$

The reasons why we consider loop levels instead of single dimensions are already detailed in Section 5.1.3. In particular, they allow considering the innermost body of a loop nest even when the nesting of dimensions is left unspecified. The loop level $\emptyset$ represents the entire computation. Thus, $E_\emptyset$ and $X_\emptyset$ represent the beginning and the end of the computation respectively.

In fixed candidates, only the loop levels that are classes of merged dimensions correspond to loops in the code. We reflect that with the concepts of *concrete* and *virtual* program point. Let c be a fixed candidate. A program point p is *concrete* if and only if it is an instruction or the entry point or the exit point of a loop level $L \in \mathbb{L}$ such that L exactly matches a class of merged dimensions. This includes the entry and exit point of the $\emptyset$ loop level. A program point that is not concrete is *virtual*. Note that this terminology does not make sense for partially specified candidates as the classes of merged dimensions may not be fixed yet.

Program Order

Let c be a candidate. We define the *program order* of c as a partial order on program points that reflects the order in which they appear in the code of any implementation that derives from c. Unordered program points indicate unspecified ordering choices.

To define this order, we first define three functions, `before`, `current` and `after` that indicate the position a program point $p \in \mathbb{P}$ relatively to statements. First, `before` lists the statements that necessarily execute before p. This is the dimensions whose exit points are before or equals to p and the instructions that are before p. Formally, we define `before` as follows.

$$\texttt{before}(p) := \begin{cases} \{s \in \mathscr{S} \mid s \neq i \wedge \texttt{order}(s, i) = \{\texttt{BEFORE}\}\} & \text{if } p = i \in \mathcal{I} \\ \left\{ s \in \mathscr{S} \;\middle|\; s \notin L \wedge \bigcup_{d \in L} \texttt{order}(s, d) = \{\texttt{BEFORE}\} \right\} & \text{if } \exists L \in \mathbb{L}.\ p = E_L \\ \left\{ s \in \mathscr{S} \;\middle|\; s \in L \vee \bigcup_{d \in L} \texttt{order}(s, d) \subseteq \{\texttt{BEFORE}, \texttt{IN}\} \right\} & \text{if } \exists L \in \mathbb{L}.\ p = X_L \end{cases} \tag{5.29}$$

Similarly, `after` lists the statements that necessarily execute after p. This is the dimensions whose entry points are after or equal to p and the instructions that are after p. Formally, we define `after` as follows.

$$\texttt{after}(p) := \begin{cases} \{s \in \mathscr{S} \mid s \neq i \wedge \texttt{order}(s, i) = \{\texttt{AFTER}\}\} & \text{if } p = i \in \mathcal{I} \\ \left\{ s \in \mathscr{S} \;\middle|\; s \in L \vee \bigcup_{d \in L} \texttt{order}(s, d) \subseteq \{\texttt{AFTER}, \texttt{IN}\} \right\} & \text{if } \exists L \in \mathbb{L}.\ p = E_L \\ \left\{ s \in \mathscr{S} \;\middle|\; s \notin L \wedge \bigcup_{d \in L} \texttt{order}(s, d) = \{\texttt{AFTER}\} \right\} & \text{if } \exists L \in \mathbb{L}.\ p = X_L \end{cases} \tag{5.30}$$

Finally, `current` lists the statements that execute when p executes. This includes the dimensions nested outside p and p itself.

$$\texttt{current}(p) := \begin{cases} \{s \in \mathscr{S} \mid s = i \vee \texttt{order}(s, i) = \{\texttt{OUT}\}\} & \text{if } p = i \in \mathcal{I} \\ \left\{ s \in \mathscr{S} \;\middle|\; s \in L \vee \bigcup_{d \in L} \texttt{order}(s, d) \subseteq \{\texttt{OUT}, \texttt{MERGED}\} \right\} & \text{if } \exists L \in \mathbb{L}.\ p = E_L \vee p = X_L \end{cases} \tag{5.31}$$

We define the *program order* from the relative inclusion of `before`, `current` and `after`. A program point p_1 is before another program point p_2 if the statements that execute before p_1 also execute before p_2, the statements that execute at the same time as p_1 execute at the same time or before p_2, the statements that execute after p_2 execute after p_1 and the statements that execute at the same time as p_2 execute after or at the same time as p_1. This definition ensures that the program order is a partial order even if the domain of the `order` choice is partially specified and contains invalid values.

Definition 5.3 (Program Order)**.** The *program order* is the partial order $\preceq$ on program points such that for all $p_1, p_2 \in \mathbb{P}$, $p_1 \preceq p_2$ if and only the condition below are satisfied.

$$\texttt{before}(p_1) \subseteq \texttt{before}(p_2) \tag{5.32a}$$
$$\texttt{before}(p_1) \cup \texttt{current}(p_1) \subseteq \texttt{before}(p_2) \cup \texttt{current}(p_2) \tag{5.32b}$$
$$\texttt{after}(p_1) \supseteq \texttt{after}(p_2) \tag{5.32c}$$
$$\texttt{after}(p_1) \cup \texttt{current}(p_1) \supseteq \texttt{after}(p_2) \cup \texttt{current}(p_2) \tag{5.32d}$$
$$p_1 \neq X_\emptyset \vee p_2 \neq E_\emptyset \tag{5.32e}$$

Section A.2.1 proves that $\preceq$ is indeed a partial order. Equation (5.32e) ensure it remains a partial order even if the kernel contains no statements. Section A.2.2 proves that if c is a fixed implementation, $\preceq$ defines a total order on the concrete execution points that matches the control flow structure of the code of c.

The following proposition, proven in Section A.2.3, extends this property to partially specified candidates. It shows that the ordering constraints exposed by $\preceq$ for a candidate c remain valid for any implementation $c' \in [\![c]\!]$ that derives from c. The program order of a partially specified candidate c thus reflects ordering constraints on the code of any implementation $c' \in [\![c]\!]$ that derives from c.

Proposition 5.1. *Let c be a candidate and $c' \in [\![c]\!]$ be a fixed candidate that derives from c. Let $\mathbb{P}$ and $\mathbb{P}'$ be the program points and $\preceq$ and $\preceq'$ the program orders of c and c', respectively. Then $\mathbb{P} \subseteq \mathbb{P}'$ and $\preceq$ implies $\preceq'$:*

$$\forall p, q \in \mathbb{P}.\ p \preceq q \implies p \preceq' q. \tag{5.33}$$

We could actually prove that $\preceq$ is monotone when specifying decisions by making assumptions about the strength of the propagators that enforce the constraints on the domain of the `order` choice. This would show that the ordering constraints exposed by $\preceq$ remain valid for any implementation c', fixed or not, that derives from c. The assumptions are respected in practice, but would make our model less generic. And the monotony is not needed to prove the correctness of the bound.

We now introduce the concept of *execution point* to represent instances of program points during the execution of an implementations.

Execution Points

During the execution of an implementation, each program point $p \in \mathbb{P}$ is executed once for each point in the iteration space composed of the dimensions nested outside d and of the dimensions that p represents (in the case where $p = E_L$ or $p = X_L$ with $L \in \mathbb{L}$). We thus introduce the concept of *execution point* to represent the different instances of each program point. An execution point is the combination of a program point and an *index* that specifies the current iteration along each loop level. The following definition first formalizes the notion of index.

Definition 5.4 (Index)**.** An *index* I is a mapping from loop levels to natural numbers, such that the number assigned to a loop level L is inferior to the minimal potential size of L. We denote $\mathbb{I}$ the set of all indexes. An index $I \in \mathbb{I}$ thus has the following form.

$$\begin{array}{rccl} I: & \mathbb{L} & \to & \mathbb{N} \\ & L & \mapsto & i \in [\![0, \texttt{size}_{min}(L) - 1]\!] \end{array} \tag{5.34}$$

For two indexes $I_1, I_2 \in \mathbb{I}$, $I_1 \leq I_2 \iff \forall L \in \mathbb{L} I_1(L) \leq I_2(L)$.

Note that indexes count iterations orthogonally along each loop level, even if some loop levels may overlap. This makes it easier to model the impact of a loop level without considering the others. We later extend the concepts of *concrete* and *virtual* program points to indexes, in order to filter out the indexes that do not make sense in the final code.

Definition 5.5 (Execution Point)**.** An *execution point* is the combination (p, I) of a program point $p \in \mathbb{P}$ and of an index $I \in \mathbb{I}$. We denote $\mathbb{E} := \mathbb{P} \times \mathbb{I}$ the set of all execution points.

An execution point $(p, I) \in \mathbb{E}$ represents the instance of the program point p that the execution of an implementation reaches after iterating $I(L)$ times along each loop level $L \in \mathbb{L}$. In particular, $I(L) = 0$ before entering the dimensions of L and $I(L) = \texttt{size}_{min}(L) - 1$ after exiting the dimensions of L.

We extend the concepts of concrete and virtual program points to execution points in order to filter out program points instances that do not appear in the execution of the final code. Let c be a fixed implementation. An execution point $(p, I) \in \mathbb{E}$ is *concrete* if p is concrete and for all $L \in \mathbb{L}$, $I(L)$ respects the following conditions.

- If L does not match a class of merged dimensions, then $I(L) = 0$. This includes the case where L contains dimensions from multiple classes, but also the case where L is only a subset of a class of merged dimensions. It ensures that concrete execution points are only iterated along the loops levels that correspond to a single loop of the final code.
- Otherwise, if $p \prec E_L$, then $I(L) = 0$. This states that $I(L)$ is zero before entering the loop represented by L.
- Otherwise, if $X_L \prec p$, then $I(L) = \texttt{size}(L) - 1$. This states that $I(L)$ is maximal after exiting the loop represented by L.

An execution point that is not concrete is *virtual*.

Note that while the concepts of concrete and virtual execution points only apply to fixed candidates, the concept of execution point is defined for any candidate, even when some decisions are left open. We now define *execution graphs* in order to represent latency constraints between execution point.

5.2.2 Execution Graph

We define the *execution graph* G_c of a candidate c to represent the latency constraints between execution points of c. We then prove that the weight of any path in G_c is a lower bound for the execution time of a thread of any implementation $c' \in [\![c]\!]$ that derives from c. For that, we first prove this property on fixed candidates and then generalize it to any candidate.

Definition

An execution graph links execution points with edges that indicate latency constraints. For $p_1, p_2 \in \mathbb{E}$, an edge $p_1 \xrightarrow{w} p_2$ indicates p_2 cannot execute less than w cycles after p_1. An edge $p_1 \xrightarrow{0} p_2$ indicates p_2 must execute after or at the same time as p_1. Edges between execution points with the same index reflect data dependencies, ordering dependencies and the latency induced by the usage of loop levels. Edges across indexes reflect the ordering of iterations, point-to-point communications and loop-carried dependencies.

The execution graph forbids iterating along multiple loop levels that contain the same dimension. This ensures each loop iteration is only considered a correct number of times. We say that a loop level $L \in \mathbb{L}$ is *exclusive* for an index $I \in \mathbb{I}$ if for every loop level $L' \in \mathbb{L}$ distinct from L that contains a dimension in common with L, or a dimension merged with a dimension of L, $I(L') = 0$. We are now ready to give the definition of the execution graph.

Definition 5.6 (Execution Graph)**.** Let c be a candidate. We define the execution graph G_c of c as the weighted directed graph whose vertices are the execution points $\mathbb{E}$ and whose edges are defined as follows.

1. The execution order of program points at a fixed index follows the program order. Let $p_1, p_2 \in \mathbb{P}$ and $I \in \mathbb{I}$. Then G_c has an edge $(p_1, I) \xrightarrow{0} (p_2, I)$.
2. The usage generated by a single iteration of a loop level $L \in \mathbb{L}$ iterations imposes a latency constraint between the entry and the exit point of L (recall that this latency includes the usage of all the statements that execute in an iteration of L). Let $i \in [\![0, k]\!]$ be a parallelism level that produces synchronization operations (including the sequential case, where $j = 0$) and $I_1, I_2 \in \mathbb{I}$ such that $I_1 \leq I_2$. Then, G_c has the following edge.

$$\forall L \in \mathbb{L}, \forall I \in \mathbb{I}.\ (E_L, I_1) \xrightarrow{\texttt{latency}(i,L)} (X_L, I_2). \tag{5.35}$$

3. Data dependencies between pairs of instructions induce latency constraints from the source instructions to the destination instructions. Let $a, b \in \mathcal{I}$ be two instructions such that b reads the value that a produces, with point-to-point communication from dimensions $(d_i)_{i<n}$ to dimensions $(d'_i)_{i<n}$ (i.e. where the values produced by a at iteration j of d_i are consumed by b at the corresponding iteration j of d'_i).

   Let $(L_i)_{i<n}$ and $(L'_i)_{i<n}$ be two series of loop levels such that for every $i < n$, $d_i \in L_i$, $d'_i \in L_i$ and L_i and L'_i only contain dimensions that are necessarily merged with each other. Let $I, I' \in \mathbb{I}$ such that $I \leq I'$ and for every $i < n$, $I(L_i) = I'(L'_i)$, and either $I(L'_i) = I'(L'_i)$ or L'_i is exclusive in I. Then G_c contains an edge from (a, I) to (b, I') of weight $\texttt{inst\_latency}(a)$.

$$(a, I) \xrightarrow{\texttt{inst\_latency}(a)} (b, I'). \tag{5.36}$$

   Note that when $n = 0$, this implies an edge with a constant index $I = I'$. On a fixed candidate, or $i < n$ L_i and L'_i are the subsets of the class or merged dimensions of d_i and d'_i respectively. For $i < n$, the condition $I(L'_i) = I'(L'_i)$ or L'_i is exclusive in I' ensures the index along L'_i only increases if L'_i is exclusive.

4. The exit point of an iteration of a sequential loop level $L \in \mathbb{L}$ is before the entry point of the following iteration. Following the hypothesis that dimensions are mapped to the lowest parallelism level possible, we consider that L is sequential if $L \subseteq \mathscr{D}_0$ and no dimensions of L may be vectorized.

   Let $I \in \mathbb{I}$ such that L is exclusive and $I(L) + 1 < \texttt{size}_{min}(L)$. Let $\mathbb{L}_{\texttt{IN}}$ be a set of loop levels ordered within L (that is $\forall L' \mathbb{L}_{\texttt{IN}}, E_L \prec E_{L'} \prec X_{L'} \prec X_L$). Let $I' \in \mathbb{I}$ be defined as follows.

$$\forall L' \in \mathbb{L}.\ I'(L') := \begin{cases} I(L) + 1 & \text{if } L = L' \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I(L') & \text{otherwise} \end{cases} \tag{5.37}$$

   Then, G_c contains an edge $(X_L, I) \xrightarrow{0} (E_L, I')$. The definition of I' accounts for the fact the loop levels of $\mathbb{L}_{\texttt{IN}}$ may execute in full for each iteration of L.

5. An instruction $a \in \mathcal{I}$ that consumes the value it produces at the previous iteration of loop level $L \in \mathbb{L}$ induces a latency constraint from one iteration of L to the next. Formally, let $I \in \mathbb{I}$ such that L is exclusive and $I(L) + 1 < \texttt{size}_{min}(L)$, let $\mathbb{L}_{\texttt{IN}}$ be a set of loop levels ordered within L and let $I' \in \mathbb{I}$ be defined as follows.

$$\forall L' \in \mathbb{L}.\ I'(L') := \begin{cases} I(L) + 1 & \text{if } L = L' \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I(L') & \text{otherwise} \end{cases} \tag{5.38}$$

   Then, G_c contains an edge $(a, I) \xrightarrow{\texttt{inst\_latency}(a)} (a, I')$.

6. Let $L \in \mathbb{L}$ be a sequential loop level and $(p, I) \in \mathbb{E}$ an execution point such that $I(L) + 1 < \texttt{size}_{min}(L)$. Then (p, I) is executed before the following instance of p along loop level L. Thus, G_c has an edge $(p, I) \xrightarrow{0} (p, I[L/I(L) + 1])$.

The rest of this section shows that the weight of any path in G_c –and in particular the longest path– is a lower bound for the execution time of a thread of any implementation $c' \in [\![c]\!]$. The following proposition, proved in Section A.3.1, first shows that the concept of longest path in G_c is well-defined.

Proposition 5.2. *The execution graph G_c of a candidate c is acyclic.*

We now show that the weight of this path is indeed a lower bound for the execution time of a thread of implementation $c' \in [\![c]\!]$. For that, we first study fixed candidates before generalizing the result to partially specified candidates.

Execution Graphs of Fixed Candidates

Let us suppose for the moment that c is a fixed candidate. We want to show that the weight of any path in c is a lower bound for the execution time of a thread of c. For this, we first state that this is true for paths that only visit concrete program points. We then generalize this result to any path in c.

Proposition 5.3. *Let c be a fixed implementation and $e_0 \xrightarrow{w_0} \ldots \xrightarrow{w_{n-1}} e_n$ be a path in G_c such that for all $i \leq n$, e_i is a concrete execution point. Then, the weight $w = \sum w_i$ of the path is a lower bound on the execution time of a thread of c.*

The proof of Proposition 5.3 comes from the fact that each edge $e_1 \xrightarrow{w} e_2$ between two concrete execution points $e_1, e_1 \in \mathbb{E}$ in G_c indicate that e_2 cannot execute less than w cycles after e_1. The next proposition is the central idea of our performance model. It states that for any path in G_c, potentially visiting virtual execution points, there is a longer path that only visits concrete execution points.

Proposition 5.4. *Let c be a fixed implementation and p a path of weight w in the execution graph G_c of c. Then, there exists a path p' of weight $w' > w$ in G_c that only visits concrete execution points.*

We prove Proposition 5.4 in Section A.4. Combined with Proposition 5.3, this proposition gives the following theorem.

Theorem 5.1. *Let c be a fixed implementation and P a path of weight w in G_c. Then w is a lower bound for the execution time of a thread of c.*

In practice, the model tries to consider the longest path possible in G_c, in order to obtain a more accurate bound for the execution time of a thread of c. We now extend the result of Theorem 5.1 to candidates with open choices.

Execution Graphs of Candidates

Let c be a candidate, potentially with open choices. The following proposition shows that the paths in G_c are preserved when specifying decisions.

Proposition 5.5. *Let c be a candidate and $c' \in [\![c]\!]$ be a fixed candidate that derives from c. Then, there exists a function $f : \mathbb{E} \to \mathbb{E}'$ from the vertices of G_c to the vertices of $G_{c'}$ that preserves the edges:*

$$\forall e_1, e_2 \in \mathbb{E}.\ e_1 \xrightarrow{w} e_2 \in G_c \implies \exists w' \geq w.\ f(e_1) \xrightarrow{w'} f(e_2) \in G_{c'} \tag{5.39}$$

Section A.3.2 explains how to define such a function f. It results from this proposition that any path in G_c has a corresponding path of greater weight in the execution graph $G_{c'}$ of every implementation $c' \in [\![c]\!]$ that derives from c. Combined with Theorem 5.1, Proposition 5.5 thus provides a lower bound for the execution time of a thread of any implementation $c' \in [\![c]\!]$.

Theorem 5.2. *The weight of any path in the execution graph G_c of a candidate c is a lower bound for the execution time of a thread of any implementation $c' \in [\![c]\!]$ that derives from c.*

This theorem generalizes Theorem 5.1 to candidates with open choices. In particular, it shows the longest path in the execution graph provides a lower bound for the execution time of a thread of c.

In practice, building G_c to find its longest path would be too slow for our application, as it contains far too many nodes. Instead, the next section shows how the performance model computes the length of a path in G_c, based on abstractions of the paths in G_c. While the path might not be the longest, it still provides a correct and accurate enough bound.

5.2.3 Abstraction of the Execution Graph

The number of execution points in $\mathbb{E} = \mathbb{P} \times \mathbb{I}$ is far too large to directly manipulate execution graphs. The reasons for the large size of $\mathbb{E}$ are twofold. First, the number of loop levels may be exponential in the number of dimensions, leading to an exponential number of program points in $\mathbb{P}$. Second, each program point is instantiated for each index in $\mathbb{I}$.

The model solves the first problem by only considering a carefully selected subset $\mathbb{L}_0 \subseteq \mathbb{L}$ of all loop levels. It thus operates on a restricted set of program points $\mathbb{P}_0 \subseteq \mathbb{P}$ that only contains the instructions and the entry and exit points of the selected loop levels.

$$\mathbb{P}_0 := \mathcal{I} \cup \{E_L \mid L \in \mathbb{L}_0\} \cup \{X_L \mid L \in \mathbb{L}_0\} \tag{5.40}$$

The model overcomes the second problem with the observation that, under certain conditions detailed below, the paths operating at a fixed index $I_0(L)$ along loop level $L \in \mathbb{L}$ are the same for any value of $I_0(L)$. This makes it possible to abstract away the indexes in the definition of execution points, and to significantly reduce the number of vertex to consider when computing the longest path.

We first assume that the loop levels $\mathbb{L}_0$ to consider are given, and we explain how the model abstracts the paths in G_c. We then explain how these abstraction fit in the computation of the lower bound. Finally, we explain how we pick $\mathbb{L}_0$.

Abstraction of the Execution Graph.

For each $D \subseteq \mathscr{D}$, we define a graphs G_c^D that abstracts the path of G_c that only iterates along the loop levels $L \subseteq D$ included in D. The vertices of each G_c^D are the program points $\mathbb{P}_0$. The goal is to build $G_c^{\mathscr{D}}$ that considers the paths along any loop levels. For $p_1, p_2 \in \mathbb{P}_0$, a path from (p_1, I_1) to (p_2, I_2) in $G_c^{\mathscr{D}}$ indicates that there exists $I_1, I_2 \in \mathbb{I}$ and a path from (p_1, I_1) to (p_2, I_2) of the same weight in G_c.

We define each G_c^D based on the graphs $G_c^{D'}$ for $D' \subset D$. The performance model thus starts from the graph $G_c^{\emptyset}$ that only considers the paths within a fixed index. It then iteratively build graphs that consider iterations along more loop levels until it reaches $G_c^{\mathscr{D}}$.

When accounting for the effects of iterating along loop level $L \in \mathbb{L}_0$, the performance model computes the length of the paths starting from a program point $p \in \mathbb{P}_0$ ordered before E_L and ending at the last iteration of L. For that it needs to know the length of the paths from p to the first iteration of E_L and the length of the paths within a single iteration of L. The weight of these paths is maximal when the model has already accounted for the effects of iterating along the dimensions that are ordered before or nested inside L. We thus define D_L as the dimensions that are ordered before or nested inside L and compute the effect of L based on D_L.

$$D_L = \left\{ d \in \mathscr{D} \;\middle|\; X_{\{d\}} < X_L \right\} \tag{5.41}$$

We are now ready to give the definition of G_c^D for each $D \subseteq \mathscr{D}$. In practice, the model only computes G_c^D for a few selected D. Informally, a path from p_1 to p_2 in G_c^D represents the paths in G_c that start at the last instance of p_1 along the loop levels $L \subseteq D$, and end at the last instance of p_2 along the same loop levels.

Definition 5.7. Let $D \subseteq \mathscr{D}$. We define G_c^D as the graph whose vertices are $\mathbb{P}_0$ and whose edges are defined as follows.

1. G_c^D reflects the edges of G_c that originate from the ordering constraints and from the usage of loop level iterations. These edges operate within a fixed index, and are thus present independently of the value of D. G_c^D thus contains the following edges, where $i \in [\![0, k]\!]$ is a parallelism level that produces synchronization operations (including the sequential case, where $i = 0$).

$$\forall p_1, p_2 \in \mathbb{P}_0.\ p_1 \prec p_2 \implies p_1 \xrightarrow{0} p_2 \tag{5.42}$$

$$\forall L \in \mathbb{L}_0.\ E_L \xrightarrow{\texttt{latency}(i,d)} X_L \tag{5.43}$$

   This corresponds to items 1 and 2 of the execution graph definition.

2. G_c^D considers the latency between two data-dependent instructions $a, b \in \mathcal{I}$, whenever the dimensions $(d_i)_{i<n}$ and $(d'_i)_{i<n}$ involved in the potential point-to-point communication are all included in D: $\forall i < n.\ d_i \in D \wedge d'_i \in D$. In that case, G_c^D contains the following edge.

$$a \xrightarrow{\texttt{inst\_latency}(a)} b \tag{5.44}$$

   This corresponds to item 3 of the execution graph definition.

3. G_c^D accounts for the paths that iterate along the loop levels $L \in \mathbb{L}_0$ such that both L and D_L are included in D ($L \cup D_L \subseteq D$) and every dimension in L is sequential. Let $p \in \mathbb{P}_0$ be a program point that is before the entry point $E_{L'}$ of any loop level $L' \in \mathbb{L}$ that contains a dimension in common with L, or a dimension that can be merged with a dimension of L. Then, G_c^D has an edge that reflects the paths that starts from p, complete the first iteration of L and then iterate along L until it reaches the entry of the last iteration of L. Let w_1 and w_2 be the weights of the longest paths from p to X_L and from E_L to X_L respectively in G_c^D. Then G_c^D contains the following edge.

$$\texttt{size}_{min}(L) \geq 2 \implies p \xrightarrow{w_1 + w_2.(\texttt{size}_{min}(L) - 2)} E_L \tag{5.45}$$

This corresponds to item 4 of the execution graph definition.

4. G_c^D accounts for the paths that follow loop-carried data dependencies along any loop levels $L \in \mathbb{L}_0$ such that both L and D_L are in D ($L \cup D_L \subseteq D$). Let $p \in \mathbb{P}_0$ be a program point that is before the entry point $E_{L'}$ of any loop level $L' \in \mathbb{L}$ that contains a dimension in common with L, or a dimension that can be merged with a dimension of L. Let $a \in \mathcal{I}$ be an instruction that depends on the value it produces at the previous iteration of L.

   Then, G_c^D has an edge that reflects the paths starting from p, go to the first instance of a along L and then follow the data dependency up to the last iteration of L. Let w be the longest paths in $G_c^{D_L}$ between p and a. Then G_c^D has the following edge.

$$p \xrightarrow{w + \texttt{inst\_latency}(a).(\texttt{size}_{min}(L) - 1)} a \tag{5.46}$$

This corresponds to item 5 of the definition of the execution graph.

The last two points of Definition 5.7 consider paths in G_c that start from a program point p that is before E_L. This is to ensure that the execution points (p, I) that p represent in G_c^D have $I(L) = 0$, and thus that L has not been already iterated. Equation (5.45) handles the first and the last iteration of L differently that the others. This is in order to account for the data dependencies from before L to the first iteration of L and for the data dependencies from the last iteration of L to after L.

The following proposition ensures G_c^D correctly represents the paths in G_c along the loop levels that are included in D.

Proposition 5.6. *Let $D \subseteq \mathscr{D}$ and $p_1, p_2 \in \mathbb{P}'$ such that there exists a path from p_1 to p_2 in of weight w in G_c^D. Let $I_1, I_2 \in \mathbb{I}$ such that $I_1 \leq I_2$ and the following conditions are respected for every loop level $L \in \mathbb{L}$.*

$$\forall i \in \{1, 2\} \,.\, L \subseteq D \wedge p_i \not\prec E_L \implies I_i(L) = \texttt{size}_{min}(L) - 1 \tag{5.47}$$

$$\forall i \in \{1, 2\} \,.\, L \cap D \neq \emptyset \wedge p_i \prec E_L \implies I_i(L) = 0 \tag{5.48}$$

$$L \cap D = \emptyset \implies I_1(L) = I_2(L) \tag{5.49}$$

Then, there exists a path from (p_1, I_2) to (p_2, I_2) of weight w in G_c.

Equations (5.47) and (5.48) show that G_c^D represents the paths between the last possible instances along the dimensions of D of the program points. Note that this includes the paths that start at the entry point of the kernel $E_\emptyset$, at index 0, and finish at the exit point of the kernel $X_\emptyset$, at the maximal index. Equation (5.49) indicates that the paths represented by G_c^D are valid for any fixed index along the loop levels that do not intersect D.

We give the proof of Proposition 5.6 in Section A.5. This proposition shows, among other things, that the weight of any path in $G_c^{\mathscr{D}}$ is less than the weight of the longest path in G_c. Combined with Theorem 5.2, it gives us the following theorem.

Theorem 5.3. *The weight of any path in $G_c^{\mathscr{D}}$ is a lower bound for the execution time of a thread for any implementation $c' \in [\![c]\!]$ that derives from c. In particular, the weight of the longest path from $E_\emptyset$ to $X_\emptyset$ in $G_c^{\mathscr{D}}$ is a lower bound for the execution time of a thread for any implementation $c' \in [\![c]\!]$ that derives from c.*

This theorem gives an algorithm to compute the lower bound from $G_c^{\mathscr{D}}$. Moreover, for any $D \subseteq \mathscr{D}$, the definition of G_c^D only depends on the graphs $G_c^{D_L}$ such that $D_L \subset D$. The performance model thus compute $G_c^{\mathscr{D}}$ by computing $G_c^{D_L}$ for each $L \in \mathbb{L}_0$, starting from $G_c^{\emptyset}$ and then following the inclusion order of the loop levels.

Each G_c^D has a number of vertices that is linear in the number of instructions and the number n of loop levels considered. The following section shows that in practice, the number of loop levels the model considers is bounded by the square of the number of statements. Contrary to an algorithm building G_c, the time complexity of the performance model is thus polynomial in the number of statements, and independent from the number of instances of each statement. In our experiments, the execution time of the performance model was always negligible compared to the time required for constraint propagation and to the time required for the evaluation of the implementations on the hardware.

Loop Levels Selection.

We previously assumed that the set $\mathbb{L}_0$ of loop levels to consider was given. We already mentioned that $L_n = \emptyset$. We now explain how the performance model chooses the other loop levels to consider. Picking the right loop levels is important for the accuracy of the bound. However, increasing the number of loop levels to consider also increases the number of graph $G_c^{D_L}$ to compute, and the number of vertices in each of them. We first make some observations on how to choose the loop levels to consider and then detail the algorithm that picks them.

Pertinent Loop Levels. The set of loop levels $\mathbb{L}_0$ to consider limits the paths in G_c that the performance model considers. Fortunately, not every loop level is pertinent when looking for the longest path in G_c. Some loop levels incur no additional latency while some others are redundant with combinations of smaller loop levels. In particular, if a loop level $L \in \mathbb{L}$ is nested inside another $L' \in \mathbb{L}$, then considering the combined loop level $L \cup L'$ will not bring any additional insights on the performance of the candidate. Indeed, the paths that iterate along $L \cup L'$ all have equivalent paths that iterate separately along L for each iteration of L'. Moreover, the usage of the hardware resources generated by $L \cup L'$ is already accounted for by L.

The algorithm we present below picks enough loop levels to consider paths that iterate along any combination of dimensions of the iteration spaces of statements, while avoiding redundant loop levels. It always picks a number of loop levels that is inferior to the square of the number of statements.

Algorithm. In the following, we denote $\mathscr{I}_s$ the set of dimensions that compose the iteration space of statement $s \in \mathscr{S}$. Note that $\mathscr{I}_s \in \mathbb{L}$ for any $s \in \mathscr{S}$ as the dimensions that are part of the same iteration space are necessarily nested in each other. Moreover, there exists $s_0 \in \mathscr{S}$ such that $\mathscr{I}_{s_0} = \emptyset$ indeed, at least one statement has no other statements nested outside (excluding the case where $\mathscr{S} = \emptyset$, in which case the lower bound for the execution time is 0).

The algorithm builds a directed acyclic graph whose nodes are the $(\mathscr{I}_s)_{s \in \mathscr{S}}$. An edge $\mathscr{I}_{s_1} \to \mathscr{I}_{s_2}$ indicates that any dimension $d_2 \in \mathscr{I}_{s_2}$ is either nested inside $\mathscr{I}_{s_1}$ or part of $\mathscr{I}_{s_1}$:

$$\forall d_2 \in \mathscr{I}_{s_2}.\ d_2 \in \mathscr{I}_{s_1} \vee \forall d_1 \in \mathscr{I}_{s_1}.\ \texttt{order}(d_2, d_1) = \{\texttt{IN}\}\,. \tag{5.50}$$

This indicates that the performance model can account for the iteration space $\mathscr{I}_{s_2}$ by considering the loop level $\mathscr{I}_{s_2} \backslash \mathscr{I}_{s_1}$, and that the remaining dimensions in $\mathscr{I}_{s_1}$ will be accounted for by the edges on the paths from $\emptyset$ to $\mathscr{I}_{s_1}$.

In practice, considering all the edges that respect equation (5.50) is unnecessary. If $s_1, s_2, s_3 \in \mathscr{S}$ such that $\mathscr{I}_{s_1} \to \mathscr{I}_{s_2}$ and $\mathscr{I}_{s_2} \to \mathscr{I}_{s_3}$, then the edge $\mathscr{I}_{s_1} \to \mathscr{I}_{s_3}$ is redundant as the iteration space of s_3 will already be accounted for by the first two edges. The algorithm thus computes the transitive reduction of the graph composed of all the edges that satisfy equation (5.50). It then computes the set $\mathbb{L}_0$ of loop levels to consider as the loop levels $\mathscr{I}_{s_2} \backslash \mathscr{I}_{s_1}$ induced by each edges $\mathscr{I}_{s_1} \to \mathscr{I}_{s_2}$.

5.3 Instantiation of the Model for Kepler GPUs

Instantiating the performance model for a particular hardware target requires specifying the hardware resources, the parallelism levels and the amount of each resource they offer. It also requires providing

functions that compute the usage and latency of statements as well as the amount of parallelism available at each level. In this section, we instantiate the performance model for the Kepler family of Nvidia GPUs. Section 5.3.1 first explains how we expose the parallelism hierarchy of GPUs to the model. Then Section 5.3.2 presents the hardware resources we consider.

5.3.1 Parallelism Levels

As detailed in Section 4.3.1, our decision space for linear algebra on GPUs exposes three kinds of parallel dimensions: vector dimensions, thread dimension and block dimensions. We define two parallelism levels ($k = 2$) to model thread and block dimensions. We model vector dimensions together with sequentiel dimensions in parallelism level 0.

Vector Dimensions. Vector dimensions can only contain a single instruction. In the model, we treat them as part of the parallelism level 0, but without dependencies between iterations. In practice, this translates into ignoring potential vector dimensions in the single thread latency model but not in the computation of `latency`.

Thread Dimensions. Thread dimensions map to parallelism level 1. The maximal number of GPU threads that can execute in a block is fixed to 1024, thus $\mu_1^{\max} = 1024$. Multiple thread dimensions in different loop nests may map to the same hardware thread dimension. We thus modify how we compute $\eta_1^{\min}$ and η_1^{lcm} to only account once for the dimensions mapped to the same hardware thread dimension. In practice, this translates into considering classes of dimensions mapped to the same hardware dimension instead of classes of dimensions merged with each other.

As Section 4.1 explains, Kepler GPUs execute threads per groups of 32, called warps. When the number of threads is not a multiple of 32, the unused slots of a warp still consume some resources. We thus correct the usage that each block of thread generates based on the minimal ratio of unused threads in warps.

Block Dimensions. Block dimensions map to parallelism level 2. Unlike thread dimensions, they do not require a special handling of $\eta_2^{\min}$ and η_2^{lcm}. The amount of parallelism available at the block level, μ_2^{max}, depends on the minimal amount of shared memory allocated per block and on the maximal number of parallel block supported by the GPU.

5.3.2 Statements Usage and Latency

The hardware resources we model are the issue units, the ALU units, the load/store units, the L1 bandwidth, the L2 bandwidth and the number of synchronization instructions. Note that global memory accesses use both the L1 and the L2 bandwidth even for accesses bypassing the caches.

We obtained the latency and the usage of instructions as well as the amount of resources available at each parallelism level by combining micro benchmarks with public data from the CUDA documentation. Except for memory accesses, all instructions have a fixed latency and usage.

The usage and latency of memory accesses depend on the number of L1 and L2 cache lines or the number of shared memory banks each warp accesses. We developed ad hoc models to compute lower bounds of these values based on the decisions already fixed. These models rely on the fact memory accesses in our area of interest are dominated by linear access patterns. The model returns the minimal possible usage and latency of a memory access for non-linear accesses.

Sources of Inaccuracy. They are a few bottlenecks we do not model, either by lack of information or by lack of time to support them. In particular, our instantiation of the performance model has the following limitations.

- It does not model the overhead induced by threads, blocks of threads and by the kernel launch.
- It assumes that all ALUs support all operations. In practice, different kind of ALUs may support different operations. For example, some ALU may only support single precision operations, double precision operations or integer operations.

- It does not model cache misses. For this, it would need to be aware of the cache replacement policy and of the warps and block scheduling algorithm. The consequence is that the model ignores the RAM bandwidth. This is less a problem than on CPUs as the L2 bandwidth is already limited and the L1 does not cache global memory accesses.

- It does not model the number of registers allocated by threads and blocks of threads.

5.4 Empirical Evaluation

We evaluated the pertinence of the performance model, instantiated for GPUs, on actual candidates and implementations. The goal was to show that it provides actionable information that helps drive the search towards the most efficient implementations. To that end, we generated 35,000 random implementations of the *matmul* kernel, described in Section 6.2. For each sample, we started from the candidate representing the full implementation space and specified choices following a fixed order: first the memory layout, then `size`, `dim_kind`, `thread_mapping`, `memory_space`, `order` and finally `cache`. For each choice, we uniformly picked a random decision amongst the possible values listed in the domain of the choice. We recorded the lower bound computed by the performance model both on the final implementation and on intermediate candidates.

Section 5.4.1 first studies the accuracy of the performance model on fully specified candidates. It shows that, although the performance model does not compute precise execution times, it still finds the right order of magnitude. Then, Section 5.4.2 shows this information is enough to prune the implementation space by several orders of magnitude. Finally, Section 5.4.3 shows our model can derive pertinent information from candidates even when only a few decisions are set.

5.4.1 Accuracy of the Performance Model

We first study the accuracy of the performance model on fully specified candidates. For this, we compare the execution time of implementations with the lower bound the model provides.

Methodology. Figure 5.1 plots the cumulative distribution of the ratio of the execution time of fully specified implementations with the lower bound the model provides for such implementations, from 0 to 100%. Note that the x axis is logarithmic.

This figure excludes the samples for which the execution on the GPU timed out, as we do not have their exact execution time. This occurs when the kernel takes more than a couple of seconds to execute, instead of a few milliseconds for the best implementation. In our case, they accounted for 10% of the samples. These samples are not representative of the candidates search algorithms manipulate as the performance model tends to prune them early on.

Interpretation. The figure shows that all the bounds the model provides fall within two orders of magnitude of the actual execution time and that 70% of the bounds fall within one order of magnitude. This is to put in consideration with the large difference on the execution time between implementations on GPUs. The best implementation runs in a few milliseconds while the worst implementations can take several seconds to execute.

It also shows our model still leaves much room for improvement. The next section shows that the information it provides is actually enough to reduce the implementation space size by the several orders of magnitude.

5.4.2 Pruning Opportunities

The performance model allows for reducing the size of the implementation space without missing the best implementation. Indeed, if a search algorithm has found an implementation x, with an execution time $T(x)$, then it can safely prune any implementation y with a lower bound $B(y)$ such that $B(y) \geq T(x)$. The performance model guarantees that $T(y) \geq B(Y) \geq T(x)$, and thus that x will not perform better than y.

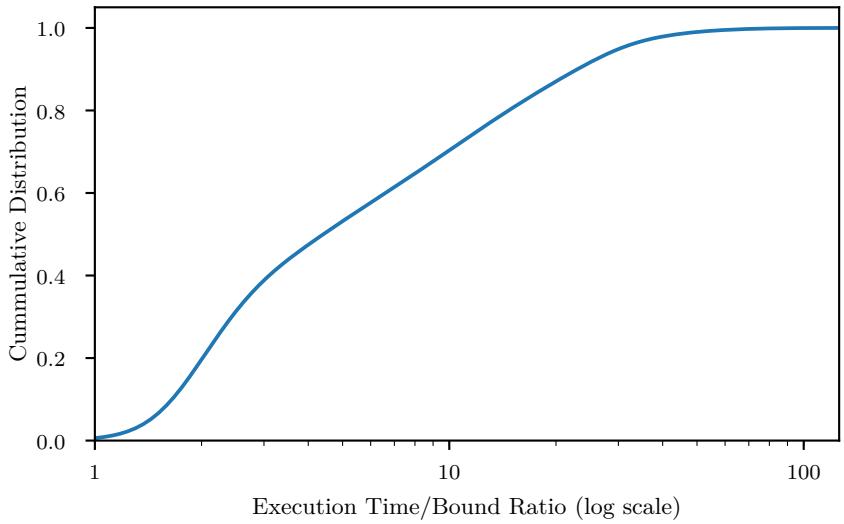


Figure 5.1: Accuracy of the Performance Model on Fully Specified Implementations

We conducted experiments to study the likelihood of sample to be pruned before being executed on the hardware. For that, we compared the bound computed by the performance model for each implementation to the execution time of the other samples.

Methodology. For each sample a, we computed the probability $p(a)$ of the existence of another sample b that has an execution time lower than the bound provided by the performance model for the fully specified implementation of a. The value $p(a)$ is thus the probability that a can be pruned by a randomly chosen implementation b.

Figure 5.2 shows the number of samples that can be pruned by at least one out of x samples. A point (x, y) indicates that one out of x sample allows reducing the number of samples to evaluate by a factor of y without missing the best implementation.

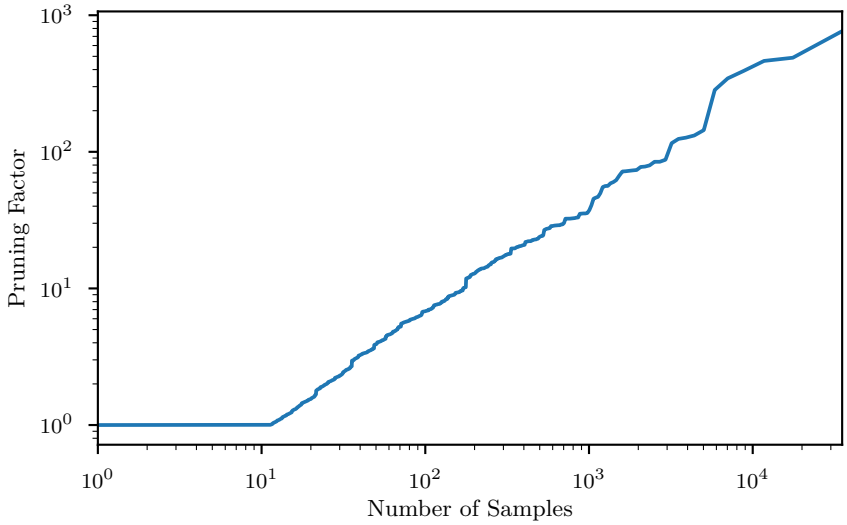


Figure 5.2: Pruning Opportunities per Number of Samples

Interpretation. Figure 5.2 shows that one sample out of 10 allows for pruning other samples. After that, the pruning power of samples grows polynomially. One sample out of 1,000 allows for pruning the samples by a factor of 37, and one sample in 10,000 allows for pruning by a factor of 420.

This experiment shows that the performance model allows for reducing the number of fully specified implementations to evaluate on the hardware based on the execution time of previous evaluation. This is the case even if the number of previous evaluation is small. Further evaluations keep improving the pruning power of the model.

In this experiment, we did not use the performance model to drive the random samples towards good implementations. An actual search algorithm could find more efficient implementations faster, and thus prune more efficiently. For experiments on the performance model in the context of a search algorithm, see Section 6.3.2. Another limitation of this experiment is that we only considered the bound of fully specified implementations. The following section shows that the performance model allows for pruning some samples earlier, before all the choices have been fixed.

5.4.3 Early Information on the Performance

The performance model is able to compute a lower bound for the execution time of any implementation that derives from a candidate. This property allows for pruning samples before all the choices are fixed. We now show that the lower bound computed by the performance model is pertinent even in the early steps of the generation process of the samples, when only a few choices have been made.

Methodology. We computed the number of sample that could be pruned by the performance model when considering only the n first steps of the generation process of each sample, that is, the candidates obtained after making at most n decisions for each sample. We assumed a candidate could be pruned if its lower bound was bigger than the execution time of the best implementation across all samples.

Figure 5.3 shows the number of samples that may be pruned for a given value of n. A point (x, y) indicates that the performance model can reduce the number of samples to consider by a factor of y after making x decisions. Note that the pruning factor axis use a logarithmic scale.

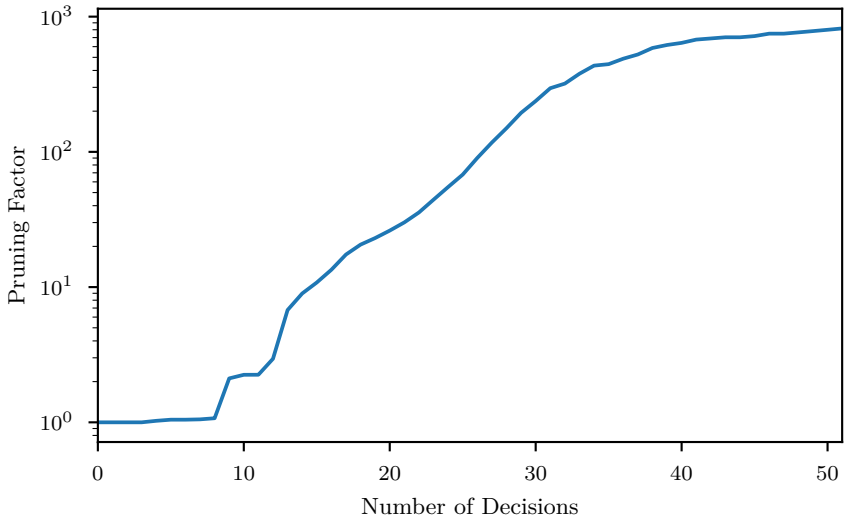


Figure 5.3: Pruning Opportunities after a Given Number of Decisions

Interpretation. The figure shows that the performance model allows for pruning early in the generation process of each sample, even when only a few decisions are made. 55% of the samples can be pruned after 10 decisions and 94% after 20 decisions. For comparison, the average number of decisions for our samples is 44.8. Note that this does not necessarily mean that the performance model is not accurate

enough in previous steps, as faster implementations might have been derivable from the early candidates of the generation process.

The plateau at the beginning of the graph corresponds to the `size` choices, that have a significant impact on the accuracy of the performance model. Before these choices are fixed, the lower bound is not accurate enough to prune more than a few candidates. Ongoing work is trargeting this problem to improve the efficency of the performance model even more.

Overall, this experiment validates our claim that candidate implementations expose actionable information early in the compilation process. They enable heuristics that reason about an entire set of implementations, without enumerating the implementations that compose the set. They also allow search algorithms to estimate the benefit of a decision independently of subsequent decisions.

5.5 Discussion

This section discusses the specificities of our performance model. Section 5.5.1 first compares our model to existing work. Section 5.5.2 then shows that how our model can justify the bounds it computes, providing insights on how to achieve better performance. Finally, Section 5.5.3 discusses the sources of inaccuracies in the model.

5.5.1 Comparison with Other Models

The two distinctive features of our model are a) that it works on candidates with decisions left open and b) that it computes a lower bound for the execution time rather than trying to model precisely the performance of implementations. We first discuss models that come close to these features before extending our discussion to performance models for fully specified implementations.

Lower Bound Models. The performance model that comes the closest our approach is the Roofline Model [44], that bounds the performance of implementations depending on their computational intensity and the optimization techniques they use. The roofline model computes a lower bound for the execution time by considering the usage of each hardware resource in isolation. Lai et al. demonstrate the effectiveness of this approach on highly parallel architectures and deep memory hierarchies, where parallelism, memory bandwidth, or another dominating factor hides secondary bottlenecks. They provide a bound within 77% percent of the actual execution time of the matrix multiplication kernel just by analyzing the different throughput constraints of a highly optimized code [45].

Our model builds on the same idea, but at a finer granularity. In particular, it reflects the impact of parallelization at each level of the parallelism hierarchy and it can model the iteration between bottlenecks provided they show up at the granularity of loop iterations. Moreover, our model allows for considering whole classes of implementations. Note that roofline-based approaches abstract away implementation details, but that does not make them applicable to partially specified candidates: for example the exact arithmetic intensity is not defined until register allocation decisions are taken.

Fully Specified Implementations Models. Other analytical performance models have been developed to drive the search for efficient implementations on GPUs [46, 47], These models might be more accurate than the one presented here. However, they address a different problem, as they try to estimate the execution time rather than providing a lower bound for it. Code generators based on such models, such as [48], are highly dependent on the accuracy of the model they use. This can be a problem when hardware vendors micro-architectural details are kept private. Statistical performance models solve these problems by automatically deriving such details [49]. However, these model only work on fully specified implementations, making it impossible to derive information from early compilation steps.

5.5.2 Bound Justification

One of the particularities of our model is that it justifies the bounds it computes. For that, it tracks of the origin of all the intermediate bounds it computes and passes the information down to the global bound. These intermediate bounds are, for example, the bound B_i on the execution time of a parallelism level or the bound for the latency between two program points in the performance model of a single thread.

The places where a bound may have multiple origins are either minimum or maximum operations. The model thus tracks the argmin or argmax of such expressions.

$\langle bound \rangle := (\mathbb{R}, \langle explanation \rangle)$

$\langle explanation \rangle :=$ Dependency
| Usage($\mathbb{N}$, $\mathcal{R}$)
| Chain($\langle bound \rangle$, $\mathbb{P}$, $\langle bound \rangle$)
| LoopLevel($\langle bound \rangle$, $\mathbb{L}$, $\mathbb{N}$)
| ParallelismLevel($\langle bound \rangle$, η_{min}, η_{max}, μ_{max})

Figure 5.4: Bound Explanation Grammar

Figure 5.4 gives the grammar of bounds and their justification. Any bound is a bound for the latency between two program points a and b. For the bounds $(B_i)_{i \leq k}$ on whole parallelism levels, these program points are $E_\emptyset$ and $X_\emptyset$. A bound is a pair of a value and a justification. The justification depends on the origin of the bound, which can be any of the following.

- It can be a data or ordering dependency between a and b.
- It can be the usage of a hardware resource $r \in \mathcal{R}$ at parallelism level $i \in \mathbb{N}$, originating from equation (5.17).
- It can be a chain of two bounds between program points a and c and c and b, originating from computing the longest path in the execution graph.
- It can be the iteration over a bound, originating from computing the effect of $n \in \mathbb{N}$ iterations of a loop level $L \in \mathbb{L}$ in one of the graphs G_c^L abstracting the paths of the execution graph.
- It can be a bound B_i of the execution time of parallelism level $i > 0$, when computed from the execution time B_{i-1} of parallelism level $i - 1$, by equation (5.27).

The justification allows the user to interpret the bound and act in consequence to widen the implementation space with new optimization. For example, if resource R is limiting at level i, the user can either reduce the usage on R by using different instructions, improve the parallelism at level i or use an accelerator with more resource R at level i. We strongly believe that a search algorithm could also use this information to efficiently drive the search.

5.5.3 Sources of Inaccuracy

The bound $B(c)$ on a candidate c should be as close as possible to the execution time of the fastest implementation $x \in [\![c]\!]$ to help the search algorithm find more efficient implementations faster. However, the lower bound should, first of all, be correct and fast to compute, even when some decisions are left open. To that end, the performance model makes optimistic assumptions and conservative abstractions on the hardware and of the set of possible implementations.

Abstraction Over the Hardware. Hardware vendors tend to keep the micro-architectural details of their accelerators private. As a consequence, it is impossible to model the hardware exactly, except by reverse engineering. We overcome this limitation by focusing on a conservative abstraction of the hardware, based on parallelism levels and resource constraints. The bound remains valid even if some hardware resources are not modeled, some resource available are underestimated or if the available parallelism is overestimated.

Even if the hardware details were public, cycle-accurate simulators are too slow for our application. Instead, our model is an analytical function that does not need to iterate over the loops.

Steady State Assumption. The performance model only considers the interaction between bottlenecks at the granularity of loop levels. The underlying assumption is the limiting bottleneck only changes at, or near, the frontier of loops. This is a reasonable optimistic assumption for the highly parallel architectures we consider. The wide parallelism of such architectures hides secondary bottlenecks and facilitates the apparition of steady states.

Abstraction Over the Set of Implementations. The performance model handles candidates with choices left open by abstracting the set of potential implementations. It makes local assumptions without enforcing any form of global coherency between them.

First, it does not enforce the constraints between decisions. It relies instead on the Cartesian over-approximation of possible implementation that the domain of each candidate provides.

Second, the performance model may make different assumptions for each of the intermediate values it computes. For example, it can assume that a load instruction accesses shared memory when computing the usage of the global memory bandwidth and assume that the same instruction accesses global memory when computing the usage of the shared memory bandwidth.

The abstraction of the hardware and the steady state assumption affect both fixed and partially specified candidates. The information they miss must be recovered via other heuristics or empirically by the search algorithm. On the other hand, abstractions on the set of implementations only affect candidates with open choices. The inaccuracy they induce disappear when specifying conditions. Their impact is thus limited. Even if the effect of a decision is not reflected in the bound before a few more decisions are specified, it will not lead to evaluations on the hardware that could have been avoided.

5.6 Summary

This chapter presented a performance model that computes a lower bound for the execution time of any implementation that derives from a candidate. This model provides a correct lower bound even if some decisions are left open, allowing a search algorithm to make motivated decisions early in the compilation process. In order to handle open choices, the performance model makes conservative abstractions of the hardware and of the set of implementations. In particular, it makes local optimistic assumptions that minimize the execution time.

The model is generic in the sense that it supports architectures with various depths of parallelism hierarchies and different resource constraints. It can also adapt to different implementation spaces. The only choices it depends on are `order` and, in a lesser extent, `dim_kind`. Other choices impact the performance of individual statements but do not affect the structure of the model.

We instantiated the model for linear algebra kernels on GPUs. Our experiments showed that, while the model does not precisely predict the execution time, it is accurate enough to prune the implementation space by several orders of magnitude. The model provides actionable information early in the compilation process, even after making only a few decisions. This validates our claim that the candidate approach allows for the definition of global heuristics, aware of the transformations that may feature in final implementations.

Providing a lower bound for the execution time of a partially specified candidate is only possible because the model is aware of all potential decisions. We do not see how such a model could work with an iterative rewriting of a fully specified implementation. The ideas developed to handle open decisions are not limited to the performance model and could apply to define other heuristics on partially specified candidates.

We also discovered that the performance model provides key insights to the designer of a code generator and auto-tuner. It can pinpoint which performance bottlenecks should be addressed to achieve better performance, hinting at missing decisions in the implementation space. This point has been of tremendous help when building and refining the decision space for linear algebra kernels presented in Chapter 4. We believe that it can also help support design space exploration, to select the most appropriate platform for a given computational domain.

Chapter 6

Scaling the Search with the Properties of Candidates

This chapter illustrates the benefits of the "candidate" concept by developing a simple search algorithm. It supports our claim that candidates empower search algorithms. The algorithm generates code optimized for specific features of a given input (matrix sizes, model hyper-parameters) and a given hardware target (GPU microarchitecture). It starts from a single candidate, representing the initial implementation space, and returns the fastest implementation it can find within a fixed amount of time.

Ideally, the performance model of Chapter 5 alone would be enough to guide the search towards the most efficient implementation. This is indeed the case for simple kernels, such as the vector addition kernel, where a search algorithm can use the lower bounds to prune large region of the implementation space at exhaustively evaluate the remaining implementations of the hardware. However, this approach does not scale to bigger implementation spaces as too many implementations remain to evaluate. We thus propose an algorithm that combines a traditional Monte Carlo Tree Search with the lower bound performance model and actual evaluation on the hardware. This algorithm leverages the unique feature of candidates: it relies on the commutativity of decisions to prioritize the choices we think are the most important and uses the performance model to derive profitability information from early compilation steps.

We apply our algorithm to find implementations of a few linear algebra kernels for GPUs, using the representation of candidates defined in Chapter 4. We show that the generated code is competitive with hand-tuned libraries and even outperforms them in some cases. We also make dedicated experiments to study the impact of specifying the most discriminant choices first and the impact of using the performance model to prune the implementation space early in the compilation process. These experiments show how the commutativity of decisions and the information exposed by candidates early in the compilation process benefit search algorithms.

Note that the algorithm is only a first step towards efficient search strategies that fully exploit the benefits of candidates. Its principal purpose is to put the ideas of previous chapters into the context of a search procedure.

Contributions. This chapter presents a search algorithm that operates on candidates in order to find efficient implementations of linear algebra kernels on GPUs. It combines the idea of candidates from Chapter 3, the encoding of implementation spaces from Chapter 4 and the performance model from Chapter 5 to show how they help finding better implementations. Precisely, this chapter makes the following contributions.

- It defines a search algorithm to find efficient implementations of a kernel. This algorithm combines a lower bound for the execution time of partially specified candidates with actual evaluations to drive the search towards good candidates.

- It shows that this algorithm can generate code competitive with hand-tuned reference libraries, outperforming them in some cases.

- It demonstrates how the structure of candidates helps the search algorithm find better implementations. In particular it shows how the commutativity of decisions affects search performance.

- It shows that deriving information early, after making only a few decisions, is critical to drive the search algorithm towards the best implementations. This validates the experiments of Section 5.4 in an actual search procedure. Together with the previous point, this shows the value of the candidates approach for optimizing compilation.

Outline. Section 6.1 first defines our search algorithm. Section 6.2 presents the kernels we use to evaluate our approach. It details the computation they perform, how we express them in our kernel representation and the characteristics of their implementation space. Section 6.3 evaluates the search algorithms on the kernels and shows how the performance model and the order of decisions affect the performance of the search. Finally, Section 6.4 places our algorithm in the context of existing work and Section 6.5 discusses how our experiments with our search algorithms shows the benefits of candidate implementations.

6.1 Search Algorithm

The core of our algorithm is similar to the algorithm that we developed in Section 2.4 to find the best schedule of instructions within a basic block. Indeed, it builds a search tree whose nodes are candidates, and leaves implementations. One key difference with Section 2.4 is that the tree is far too big for exhaustive enumeration, even with a performance model to prune parts of it. Our algorithm thus combines two components: a statistical component that we develop in Section 6.1.1, and a performance model based component that we develop in Section 6.1.2.

The statistical component drives the search towards the most efficient implementations, based on previous evaluation of implementations on the hardware, while the performance model component prunes the search tree and drives the search in the absence of previous evaluation in a sub-tree.

6.1.1 Statistical Exploration

The exploration of the implementation space is driven by a Monte-Carlo Tree Search (MCTS) algorithm. We use a variant of Threshold Ascent on Graph (TAG), which was previously applied to Spiral [50], a code generator for fast Fourier transforms.

In our case, the algorithm builds a tree whose nodes are candidates. The root of the tree is the candidate representing the entire implementation space. The algorithm starts with the tree containing only the root node and iteratively selects a leaf to expand by using evaluation statistics from previous iterations. The leaf expansion instantiates an open choice, creating one child per possible value. Then, the MCTS performs a Monte-Carlo simulation to set remaining decisions. It runs the resulting implementation on the GPU and adjusts the statistics along the selected path with the execution time. The order in which choices are specified is fixed upfront. We manually select an order that specifies most important decisions first. Section 6.3.3 discusses the impact of the order on exploration performance.

To select the leaf to expand, the algorithm iteratively builds a path from the root of the tree. At each node along the path, it applies the TAG formula to select the next node on the path among the children of the current node. The TAG formula works as follows. For each node, the algorithm stores the T best execution times obtained when expanding a leaf in the sub-tree rooted at the node. Let us consider a node with k children. For $i < k$, we denote n_i the number of times the algorithm visited the i-th children of the node and s_i the number of evaluations in branch i that leads to an execution time among the T best. Then, the algorithm selects the next children to visit with:

$$\underset{i<k}{\operatorname{argmax}} \frac{s_i + \alpha + \sqrt{2s_i\alpha + \alpha^2}}{n_i} \tag{6.1}$$

where $\alpha := \ln(2nk/\delta)$, n the sum of the $(s_i)_{i<k}$ and δ a constant to select the trade-off between exploration and exploitation.

6.1.2 Performance Model

We complement the TAG algorithm using the performance model defined in Chapter 5, that provides a lower bound for the execution time of any implementations deriving from a candidate. We use the bound in two ways:

- When selecting a leaf to expand, we ignore children with a lower bound that is higher than the execution time of the current best implementation.
- During Monte Carlo simulations, when choosing amongst candidates $(X_i)_i$, we pick X_i with probability:

$$p(X_i) \sim \max(T - b(X_i),\, 0) \tag{6.2}$$

  where T is the execution time of the best implementation so far and $b(X_i)$ is the lower bound given by the performance model for the candidate X_i. Note that $p(X_i) = 0$ when $b(X_i) > T$.

  The idea behind equation (6.2) is that we select a candidate proportionally to its potential improvement of the best execution time found so far. In particular, if $b(X_i) > T$, $p(X_i) = 0$: we ignore candidates with a lower bound higher than the current best implementation.

In both cases, we use the performance model to ignore regions of the implementation space which cannot possibly improve on the best implementation already found. This is similar to how we prune the search tree in the exhaustive algorithm of Section 2.4. The difference here is that we also use the model to influence the probability of choosing a candidate in the Monte-Carlo simulation.

6.2 Sample Kernels

We now present a few kernels and their related implementation spaces that we use to run experiments on the search algorithm. We first describe the computations the kernels perform and the implementation space we expose to implement them in Section 6.2.1. We then study the characteristics of these implementation spaces in Section 6.2.2.

6.2.1 Kernels Description

We give below the list of kernels we consider. We assume that input and output matrices are stored in column major order. We created the implementation spaces of each kernel following the same procedure. We placed each instruction in its own loop nest, with point to point communication between them. We then strip-mine dimensions in each loop nest.

With this procedure, the search algorithm is free to reorder, fuse, unroll or vectorize loops or to map them to the different levels of parallelism. It can implement point-to-point communications using registers or by allocating temporary arrays in shared memory and chooses the level of cache to use for each memory access. It can also pick the strip mining factors of dimensions, as long as they divide the logical dimension size and fall in the intervals we give below for each kernel.

axpy : computes $z = \alpha.x + y$, where α is a scalar and x, y and z vectors of size n. We strip-mine n twice, with factors in $[\![2, 4]\!]$ and $[\![2, 1024]\!]$.

matmul : computes $C = A \cdot B$ when A and B are matrices of size $m \times k$ and $k \times n$ respectively. We strip-mine m and n twice, with factors in $[\![2, 32]\!]$ and $[\![2, 4]\!]$.

strided matmul : is the same as *matmul*, but with consecutive elements of A stored with stride of 32, that is, with one value per line of cache.

batched matmul : computes b matrix multiplications in parallel. We strip mine each dimension once: m, n and k with a factor in $[\![2, 64]\!]$ and b with a factor in $[\![2, 128]\!]$. We instantiate the size of dimensions in the generated code, except for the batch size.

reuse matmul : is the same as *batched matmul*, but reuses the same matrix B for all matrix multiplications.

The benchmarks are representative of both compute-intensive (*matmul* and its derivatives) and bandwidth-bound (*axpy*) kernels. They span a variety of memory access patterns, including strides at different dimensions, transposed layouts, and batched versions, all of which are typical for higher dimensional tensor algebra in computational chemistry, simulation codes, and machine learning [12].

We compare the implementations found for each kernel with a reference implementation. Except for *strided matmul*, the reference implementation calls CuBLAS [51], Nvidia's hand-tuned implementation of basic linear algebra kernels. The *strided matmul* kernel showcases the advantages of our approach for kernels that are not covered by GPU vendors libraries. Indeed, CuBLAS does not support such strided accesses. Instead, the reference is a naive implementation that computes one element of C per thread.

6.2.2 Implementation Spaces Characteristics

We now study the characteristics of the implementation space of each kernel. For this, we first explain how we measure the size of implementation spaces and then present these sizes and the average number of decisions towards a fully specified implementation.

Implementation Spaces Size Estimation. Computing the exact size of the implementation spaces is intractable due to the large number of possible choices. To estimate their size, we turn to probabilistic methods which have been used to estimate the size of search trees in constraint solvers [52].

The precise method we use was described by Chen [53], which is a generalization of an earlier method by Knuth [54]. The Knuth algorithm starts at the root and performs a random descent until reaching a leaf. It then estimates the size of the tree by using the branching factor along the path. The Chen algorithm, called heuristic sampling, adds the concept of strata: classes of sub-trees estimated by the algorithm designer to be structurally similar. The algorithm maintains a queue containing strata, represented by a single sub-tree in the stratum, along with an estimate of its size. When a sub-tree is discovered, the corresponding stratum in the queue is updated with its count, and it can randomly be selected as the new representative for the stratum. The total size estimate is then the sum of estimates for the leaves encountered. A partial order on the strata which is strictly decreasing along the tree is required to ensure well-formedness. This algorithm was chosen for its balance between simplicity and performance. We use a lexicographic pair of the depth in the tree and number of remaining choices (assuming none gets forced through propagation—this provides a simple proxy for the sub-tree size to guide the algorithm) as a stratifier.

Kernel	Input Size	Search Space Size		Average Depth	
axpy	$n = 2^{26}$	1.1e11	±1.7e10	21.9	±0.1
matmul	$m = n = 256$, $k = 32$	1.83e21	±3.3e20	41.0	±0.1
matmul	$m = n = k = 1024$	3.5e21	±1.8e21	41.0	±0.1
strided matmul	$m = n = k = 1024$	6.0e20	±2.0e20	40.9	±0.1
batched matmul	$b = 512$, $m = n = 32$, $k = 64$	2.5e28	±1.3e28	57.4	±0.2
reuse matmul	$b = 512$, $m = n = 32$, $k = 64$	7.1e25	±4.5e25	56.2	±0.2

Table 6.1: Kernels Characteristics

Per Kernel Characteristics. Table 6.1 presents the input sizes we consider for each kernel, along with the size and depth of the resulting implementation spaces. As both the reported size and depth of the implementation spaces are derived from statistical estimations, we provide the numbers with their 95% confidence interval.

As the list of valid tiling factors depends on the dimensions size, both the size and the depth of the implementation space depend on the input size. For *matmul*, we consider two input sizes: $m = n = 256$ with $k = 32$ and $m = n = k = 1204$. In the rest of this dissertation, we denote them *matmul 256* and *matmul 1024* respectively.

For computing the size of implementation spaces, we ran 1000 iterations of the Chen method and $100,000$ iterations of the Knuth method, and selected the result that had the best confidence interval.

For some of the largest implementation spaces, we perform additional iterations to bring the confidence interval to the correct order of magnitude.

The reported depth is the average number of decisions to make to obtain a fully specified implementation when randomly selecting the outcomes of choices. Note that this is not the average depth of a uniformly selected implementation. Indeed, different decisions, each selected with the same probability, may lead to regions of implementation space with different sizes.

6.3 Empirical Evaluation

This section presents experiments, based on the kernels defined in Section 6.2, that study how the properties inherent to candidates benefit search algorithms. Section 6.3.1 first studies how the generated code compares to reference implementations. Then, Section 6.3.2 shows how candidates help the search algorithm by exposing actionable information early in the compilation process. Finally, Section 6.3.3 shows how the fact that decisions commute improves the search performance. All experiments were conducted on a Linux machine equipped with a 12-core Xeon E5-2620v2 processor, 64GB of RAM and a Quadro K4000 Kepler GPU, with the CUDA toolkit version 10.0.

6.3.1 Search Performance

We first show that our search algorithm based on candidate implementations generates code that is competitive with hand-tuned implementations supplied by GPU vendors. For this, we ran the search algorithm multiple times on the implementation space of each kernel. We then compare the execution time of the best implementation the algorithm found at each run to the reference implementation.

Kernel	Avg. Runtime		Reference	Speedup	
axpy	$7.06ms$	$\pm 0.02ms$	$9.16ms$	**1.30**	± 0.005
matmul 256	$29.4\mu s$	$\pm 2.22\mu s$	$77.4\mu s$	**2.63**	± 0.18
matmul 1024	$4.34ms$	$\pm 0.18ms$	$3.71ms$	**0.85**	± 0.03
strided matmul	$10.1ms$	$\pm 0.59ms$	$637ms$	**66.7**	± 3.9
batched matmul	$578\mu s$	$\pm 3.8\mu s$	$400\mu s$	**0.69**	± 0.04
reuse matmul	$557\mu s$	$\pm 155\mu s$	$396\mu s$	**0.71**	± 0.09

Table 6.2: Implementation Space Exploration Results

Methodology. We ran the search algorithm on the implementation space of each kernel for the minimum of 1 hour or 10,000 evaluations on the GPU. We reproduced each experiment 10 times. We measured the performance of the best implementation found by each run of the search algorithm and of the reference implementations of each kernel by evaluating them 40 times on the hardware. Each time, the 95% confidence interval was within $\pm 0.5\%$ and is thus omitted in the following results.

Table 6.2 summarizes the performances of the implementations found by the algorithm for each kernel. The execution time we report only include the execution on the GPU, without the transfers of data from and to the host CPU.

- The *Avg Runtime* column reports the average accross the 10 experiments of the runtime of the best implementation found by the search algorithm. It also reports the maximal variation between this average and the runtime obtained for each experiment.

- The *Reference* column reports the runtime of reference implementations.

- The *Speedup* column reports the average speedup accross the 10 experiments, as well as the maximal variation between this average and the speedup for each experiment.

Interpretation. Below are the main lessons we can derive from the experiments on each kernel.

- *axpy* and *matmul* 256 show that our approach is able to outperform hand-tuned implementations by finding better implementation decisions.
- *matmul* 1024, *batched matmul* and *reuse matmul* compare against implementations that almost reaches the peak performance of the GPU. We have little chance of beating them. Indeed, the literature shows that manual register allocation is needed to avoid register bank conflicts [45]. Still, we generate code that achieves 85% of the performance of Nvidia's implementation for *matmul* 1024, and around 70% of this performance for *batched matmul* and *reuse matmul*.
- *strided matmul* shows the benefits of our approach for kernels unavailable in hand tuned libraries, with a 66× speedup over a naive implementation. While not reaching the peak theoretical performance of the GPU, it is within a factor 2.7 of CuBLAS non-strided version (as shown by the ratio between the average runtime of *strided matmul* and the reference runtime of *matmul* 1024 in Table 6.2) and thus can be useful.

The variance on the execution time of the best implementation among explorations on the same search space shows that we still have room for better search algorithms. However, this is beyond the scope of this dissertation that focuses on how to expose the implementation space rather than on how to design search algorithms.

6.3.2 Discriminant Information Exposed by Candidates

We have already shown in Section 5.4 that our representation of the implementation space allows for extracting pertinent information early in the compilation process, when only a few decisions are set. We now confirm those results in the context of the search algorithm, and with more kernels considered.

Methodology. To measure the impact of the information that candidates expose early in the compilation process on the search performance, we examine at how the lower bound performance model prunes the search tree in its first levels. For this we expand the entire search tree up to depth 10. We then apply the performance model to prune these first 10 level only and estimate how many nodes and implementations remain in the tree.

To prune the search tree, we cut nodes whose lower bound was higher than the average runtime of the best implementation obtained from the experiments of Section 6.3.1.[1] We only pruned nodes up to depth 10, but pruning a node also removes all the candidates below it.

Table 6.3 reports the estimated number of implementations in the search tree, before and after pruning the first 10 levels. For this, it relies on the algorithm estimating the implementation space size from Section 6.2.2. Due to the large size of the implementation space of *batched matmul* and *reuse matmul*, we could not obtain a meaningful estimation of the size of the pruned space for these two kernels. The table also reports the number of nodes of depth 10 in the tree, both before and after pruning. Contrary to the number of implementations, we obtained these numbers through exhaustive enumeration up to depth 10.

Interpretation. These experiments show that we are able to eliminate large portions of search spaces early on. For most kernels, we reduce the size of the space by two or three orders of magnitudes in the first 10 levels. The implementation space of *reuse matmul* is more complex than the implementation space of other kernels and would require to prune deeper in the tree to have a higher proportion of pruned nodes. We still manage to cut about 40% of the nodes at depth 10.

6.3.3 Impact of the Decisions Order

Finally, we show how commutative decisions empower the search algorithm. For this, we ran the search algorithm with different priorities on the choices to specify first. The idea is that specifying the most

[1] Our performance model does not account for the overhead of the driver on the bounds it computes. As such, we used the runtime measured by the GPU performance counters – that does not account for the time spent by the driver before and after executing the kernel – to cut the nodes. The cut threshold is thus slightly lower than the value we report in Table 6.2 for the average runtime of the best implementation.

Kernel	Implementations in the Search Tree					Nodes at Depth 10		
	Before		After		**Ratio**	Before	After	**Ratio**
axpy	1.1e11	±1.7e10	2.9e7	±1.1e7	**3793**	86,587	121	**715**
matmul 256	1.83e21	±3.3e20	3.5e18	±2.2e18	**523**	134,060	20,126	**6.7**
matmul 1024	3.5e21	±1.8e21	9.5e17	±4.6e17	**3884**	161,980	6,738	**24**
strided matmul	6.0e20	±2.0e20	8.1e17	±4.5e17	**740**	142,780	9,512	**15**
batched matmul	2.5e28	±1.3e28				344,464	1,946	**177**
reuse matmul	7.1e25	±4.5e25				69,805	43,402	**1.6**

Table 6.3: Effects of Pruning the Search Tree Up to Depth 10

important decisions allows both the performance model and the MCTS to discriminate closer to the root of the tree and to examine fewer branches. Note that the order of choices only affects the structure of the search tree and not the implementations that compose the implementation space.

Methodology. For the experiments we present in Sections 6.3.1 and 6.3.2, the search algorithm specifies the decisions in the following order: first the memory layout, then `size`, `dim_kind`, `thread_mapping`, `memory_space`, `order` and finally `cache`. This order was obtained from manual experiments and experience, to prioritize the most discriminative decisions.

Here, we compare the performance of the search both with this order and its opposite, on the *matmul 1024* kernel. In both cases, we computed the ratio of nodes that the performance model can prune in the first T levels of the tree, assuming the cut threshold is the average runtime reported in Table 6.2. The number of candidates with a depth $\leq d$ varies greatly with the order of decisions. To have comparable results, we took, for each order, the first d such that the number of candidates of depth d is above 10^5. This resulted in $d = 10$ with 1.6×10^5 for the direct order and $d = 16$ with 1.1×10^5 nodes for the reverse.

Interpretation. With the direct order, the performance model reduces the tree size by a factor of 24. This factor decreases to 1.8 for the reverse order. This yields 13 times more branches to consider. We also tried running the search algorithm using the reverse order but it ran out of memory due to an explosion in the number of branches. While we illustrated the impact of the decision order with the performance model, it also applies to other algorithms: picking the most discriminant decisions first helps focusing the search.

6.4 Discussion

We now compare our algorithm to existing search algorithms, both in the context of code optimization and in the general context of optimizing an objective function. We first discuss genetic algorithms and then explain how our algorithm relates to reinforcement learning. This section suggests direction for future research that we further develop in Section 7.2.3.

Genetic Algorithms. A common approach to find a good implementation of a program is to use genetic algorithms to apply rewrite rules [55]. Genetic algorithms maintain a set of potential implementations. At each step of the algorithm, they select a part of the set (the selection phase) on which they apply random transformations (the mutation phase) to generate the content of the set at the next iteration. The algorithm may also apply rules (crossover rules) to combine multiple implementations together to create and new ones for the next step.

Pouchet et al. proposed custom genetic operators for affine schedules, to search for dependence-preserving loop nest transformations [33, 34]. They operate on a vector of decisions – the coefficients of the affine schedule. However, their algorithm requires the implementation space to be convex. More recently, genetic algorithms were combined with a machine-learning based performance model to optimize tensor computations for performance [24]. The authors used the model on fully specified implementations to select the implementations for evaluation on the hardware after the mutation phase. An interesting idea would be to use our approach on top of their representation to explore schedules.

In contrast to what candidates offer in the early compilation steps, genetic algorithms lack a global understanding of what transformations are possible. Instead, they are limited to local transformations. In our case, genetic algorithms could help finding a local optimum after generating a kernel. Vasilache et al. recently showed that a similar approach yields good results [35]. In their approach, the position of the genetic algorithm in the decision process in inverted: a genetic algorithm selects high level strategies and parameters, followed by a gray box optimization based on integer linear programming.

Reinforcement Learning. The problem of choosing actions to perform (e.g. implementation decisions) in an environment (e.g. the decisions made so far or the domain of the current candidate) to maximize a value (e.g. the performance of the kernel) is not limited to optimizing compilation. The process of automatically finding the actions to perform based on the impact of previous actions is called *reinforcement learning*.

The statistical component of our algorithm is a variant of Threshold Ascent on Graph (TAG), which was previously applied to Spiral [50], a code generator for Fast Fourier Transforms. This work was itself based on a general solution of the multi-armed bandit problem [56], where an algorithm must choose between a fixed number of possible outcomes of a choice, to maximize the expected value of an objective function.

Several breakthroughs have since been achieved in the field of reinforcement learning. In particular, for algorithms playing games, such as the Alpha Go algorithm [57] that was able to beat human players at the game of Go solely based on the game runes. While they are similarities between such algorithms and our problem, reusing them is non trivial. Indeed, most of the research on this type of approach focuses on adversarial contexts, where other players also take actions. In particular, our search is totally deterministic while adversaries introduce a random component. Another difference is that these algorithms optimize for the average outcome of the actions while we only care about the best implementation found.

6.5 Summary

This chapter presented a simple search algorithm that combines a statistical approach with a performance model of partially specified implementations. The performance model prunes the implementation space and biases the search towards promising regions while the statistical component focuses the exploration based on actual evaluations. This search algorithm is only a first iteration, but it already shows promising results, generating code that is competitive with reference hand-tuned libraries, and outperforms them in some cases.

More importantly, this simple algorithm demonstrates how our approach of representing the implementation space with candidates helps search algorithms find more efficient implementations faster. We show that taking most discriminative decisions first has a considerable impact on the search performance. This is only possible because we expose possible decisions upfront and because they are commutative. We also show that candidates expose actionable information early in the compilation process, even when only few decisions have been made. This information is critical to reduce the size of the implementation space, pruning whole regions of the implementation space at once. As explained in Chapter 5, this is only possible because of the specificities of our representation.

So far, we limited ourselves to a relatively simple search algorithm with a few remaining hardwired decisions. More work is needed to fully exploit the benefits of our approach. For example, we currently use a fixed priority to select the choices to instantiate next. The algorithm could instead automatically search for the best order. We could also share information about the best implementations between the branches of the search tree when considering the same choices. Our representation makes such a sharing easy, as choices have fixed positions in the decision vector.

Chapter 7

Conclusion

We presented a novel approach to generate efficient code for hardware accelerators. Specifically, we focused on the problem of choosing amongst the many potential optimizations that may be applied to the initial program. For that, we proposed to expose the compilation process as the progressive refinement of a *candidate implementation*. A candidate is a partially specified implementation with some implementation choices left open while others are fixed. The originality of this approach is that all potential decisions are made explicit upfront and that they commute. This makes it possible to anticipate downstream decisions and to extract profitability information from the early compilation steps.

The concept of candidate is based on the Constraint Satisfaction Problem (CSP) formalism. A candidate has a *domain* that lists the possible decision for each choice. Constraints restrict the valid combinations of decisions to ensure they are coherent and respect the semantics of the kernel and the limitations of the hardware. An implementation is a candidate whose domain contains a single value for each choice and satisfies the constraints. A candidate the represents the whole set of implementations that can be obtained by restricting its domain. This allows for manipulating entire regions of the implementation space at once and to extract information about them without enumerating individual implementations.

We introduced a formalism to describe the choices and constraints that compose the implementation spaces independently of the kernel instance. It defines a generic *decision space* that can then be instantiated for each kernel. We propose a Domain Specific Language (DSL) implementing the formalism. It allows the programmer to define a decision space in a declarative form and to automatically generate code to represent, instantiate and manipulate the corresponding candidates.

We applied the DSL to build a candidate representation for linear algebra kernels on GPUs. The decision space supports the most common transformations for this domain of applications, with both thread-local and global implementation decisions, including compositions of strip-mining, loop fusion, loop interchange, unrolling, vectorization, mapping to multiple levels of parallelism, and orchestrating data movements across the memory hierarchy. This encoding serves both to illustrate our approach and to provide a new generator for high-performance linear algebra kernels on GPU.

The unique properties of candidates allow for extracting reliable information from the early steps of the compilation process. We presented a performance model that computes a lower bound for the execution time of any implementation that derives from a partially specified candidate. Our experiments showed that the model allows for pruning candidates after only a few decisions are made, and that it allows for reducing the implementation space size by several orders of magnitude.

Finally, we evaluated our approach in the context of a search algorithm. For this, we developed an algorithm that combines the performance model with actual evaluation and statistical search. This algorithm is only a first step towards building a search strategy that fully exploits the benefits of candidates. We applied our search algorithm on implementation spaces for linear algebra kernels. We showed that it generates code that is competitive with hand-tuned libraries, and that the properties of candidates have a significant impact on the performance of the search. In particular, we showed that being able to make most discriminative decisions first is critical to drive the search algorithm towards the most efficient implementations.

Outline. The rest of the conclusion chapter is organized as follows. Section 7.1 first goes back to the main contributions of this dissertation. It explains their importance, discusses the current limitations of our work, and gives our short-term research directions to improve them. Section 7.2 then presents our long-term research perspectives towards new contributions.

7.1 Principal Contributions

Our main contributions fall into four categories that we develop in dedicated sections: the introduction and formalization of the concept of candidate in Section 7.1.1, the DSL to describe decision spaces in Section 7.1.2, the instantiation of our approach for linear algebra on GPUs in Section 7.1.3 and finally the lower bound performance model Section 7.1.4.

7.1.1 Concept of Candidate Implementation

Our first contribution is the concept of candidates, which allows for the formal definition, representation and exploration of spaces of possible implementations. The representation is compact as it does not list individual implementations. Instead, it exposes potential decisions as the domain of a constraint satisfaction problem. This domain defines the possible values for each implementation choice.

Benefits for Search Algorithms. Candidates help search algorithms find more efficient implementations faster. First, a candidate decision vector defines a well-behaved set of actions the algorithm can take to generate implementations. In particular, decisions commute and the set of potential decisions decreases after each action. This means that the search algorithm is aware of all potential actions upfront and can make the most performance-impacting decisions first. Our experiments showed that the order of choices had a significant impact on the search performance.

Second, our approach is not tied to a particular framework to express the objective function and to drive the search. Instead, it allows to interact with the intermediate steps of the decision process and to mix actual evaluations on the hardware, performance models, heuristics and statistical approaches. This is in opposition to black-box solvers, such as ILP or SMT solvers, optimizing for a single objective function. Such solvers cannot fully encode the complexity of the target architecture.

Finally, candidates allow for the extraction of actionable information early in the compilation process. Indeed, the decision vector offers a Cartesian over-approximation of all possible combinations of decisions. As a result, we can define global heuristics aware of which optimizations may feature in the final implementations. As a proof, we built a model that computes a lower bound of the execution time of possible implementations of a candidate. Our experiments showed that the model provided pertinent information, even after taking only a few decisions. Such heuristics could also be a probability distribution on the actions to take next or a probability that an implementation is better than another.

Candidates Formalism. We provided a formal definition of candidates and of the implementation space they represent. This formalism exposes the objects that make up the kernel representation and the choice and constraints that define the decision space. Importantly, the definition of a decision space is independent of a particular kernel instance. This enables defining the decision space once and instantiating it for different kernels.

Even though candidates can expose all possible decisions upfront, they can also purposefully hide some decisions until a condition is satisfied. This is useful when lowering high level constructs. For example, it can be lowering a point-to-point communication between two loop nests into a store and a load instruction. In that case, it makes sense to delay decisions about the load and store instruction until after the decision to lower the point-to-point communication into memory instructions. This avoids cluttering the decision space with auxiliary decisions that may have no impact on the final code. Such decisions can still be pre-computed so that heuristics working on partially specified candidates are still aware of all potential choices.

7.1.2 Decision Space Description Language

Our second contribution is the decision space description language and its associated compiler. The DSL, allows programmers to provide a declarative specification of the properties that constitute the

kernel representation, and of the choices, the constraints and the lowering conditions that constitute the decision space from which the compiler automatically generates code to instantiate, represent and manipulate candidates.

The declarative nature of the DSL makes it concise and simple to use. A programmer can add an outcome to a choice or add a new constraint with just a few lines of code. The DSL compiler then generates the code for the corresponding candidate representation. This simplicity played a central role in the design of the decision space for linear algebra kernels on GPUs, enabling us to try out different decision spaces faster. It should also enable new candidate representations for other domains of applications or for other hardware targets.

The DSL describes the decision space independently of the kernel instance. For this, it relies on sets of objects exposed by the kernel representation. It defines the same choices, constraints and triggers for every combination of objects in the same sets. Retrospectively, we could have designed a simpler DSL by matching more closely the candidates formalism. While sets accurately match unary properties of the candidate formalism, properties with a higher arity are cumbersome to encode. The reason is that the formalism was only finalized after the DSL. We intend to fix this limitation in future work.

7.1.3 Candidates Representation for Linear Algebra on GPUs

Our third contribution is the application of our approach for linear algebra kernels for GPUs, providing a new way to automatically generate code that is competitive with hand-tuned libraries, outperforming them in some cases. Our decision space exposes complex decisions that fundamentally change the structure of the generated code, including compositions of strip-mining, loop fusion, loop interchange, unrolling, vectorization, mapping to multiple levels parallelism, and orchestrating data movements across the memory hierarchy.

The first limitation of our framework for the generation of efficient implementations of linear algebra kernels for GPUs is that it lacks an input language. Programmers must currently build the kernel representation using API calls. The reason for this is that we preferred to focus on the implementation space encoding rather than the kernel representation. In the future, we intend to write a compiler that translates an existing language into our representation.

The second limitation is that our representation only supports a limited class of kernels. In particular, it only supports loop bounds that are function of the kernel parameters and it only supports point-to-point communication between loop nests. We intend to address this problem using a polyhedral representation of communications and loop structures. Indeed, while polyhedral representations are usually used in conjunction with ILP solvers, we can also express them as constraint satisfaction problems. In that case, the variables of the CSP are the coefficients of the polyhedral encoding. We believe that such a representation would also improve the efficiency of our representation, as the `order` choice currently has a quadratic memory footprint and its transitivity constraints have cubic time complexity.

7.1.4 Lower Bound Performance Model of Partially Specified Candidates

Our last principal contribution is the lower bound performance model. We provided both a generic model and its instantiation for linear algebra kernels on GPU. This model demonstrates two properties of candidates. First, that it is possible to compute a correct lower bound for the execution time of an implementation and second that it is possible to do so even when some choices are left open, thus allowing for information to be derived early in the compilation process.

The generic model makes minimal assumptions about the hardware and the decision space. It only supposes a hierarchy of parallelism levels and that the structure of the code is specified with the `order` choice. It handles unspecified decisions by making local optimistic assumptions, independently for each hardware bottleneck.

The instantiation of the performance model for GPUs allows for pruning the implementation space, reducing its size by several orders of magnitude. The main limitation when building the model was the lack of public information about GPUs micro architecture, such as the internals of the memory system. We could certainly build a more accurate model with access to more information.

7.2 Long Term Perspectives

This final section presents our long-term research perspectives, including new contributions concerning candidates and their application in search algorithms. Section 7.2.1 presents an idea to encode complex control flow structures without relying on a fixed control flow graph. Section 7.2.2 details how we could enrich the lower bound of the performance model to help pick choices to instantiate. Finally, Section 7.2.3 lays out our perspective towards building better search algorithms.

7.2.1 Predicate-Based Control Flow Encoding

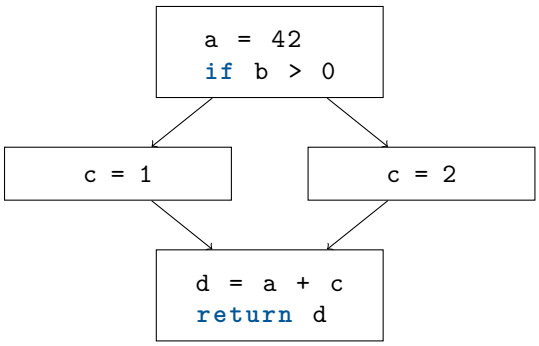
The decision space for GPU implementations of linear algebra kernels is limited to counted loops with no data-dependent bounds. Section 7.1.3 already proposed to use the polyhedral representation of loop nests to support a broader class of programs. This section proposes an orthogonal approach to represent partially specified control flows. This new approach is compatible with the polyhedral model as it encodes the local structure of the control flow, while the polyhedral model encodes the global structure of loop nests.

We first present Control Flow Graphs (CFGs), the usual approach to represent control flow in optimizing compilers. We then introduce our idea to represent partially specified control flows. Finally, we discuss the current limitations of this approach and the work we intend to do to improve it.

Control Flow Graphs. The traditional approach to represent the control flow of programs is to rely on CFGs. The nodes of a CFG represent basic blocks, that is blocks of straight line code with a single entry and exit point, while the edges represent jumps in the control flow. Figure 7.1 shows an example of CFG for an imperative code.

```
a = 42
if b > 0:
  c = 1
else:
  c = 2
d = a + c
return d
```

(a) Original Imperative Code



(b) Corresponding Control Flow Graph

Figure 7.1: Example of Control Flow Graph

The problem of control flow graphs is that they impose a fixed structure on the code. We do not see how we could define a partially specified control flow graph. The sea-of-nodes approach tries to solve this problem by relaxing the relation between the CFG and the instructions [58]. Instead of fixing the order of instructions within basic blocks, it represents them in a separate data-flow graph, with control dependencies from the CFG to the instructions. However the structure of the control flow is still fixed.

Predicate-Based Control Flow. We propose to represent the control flow of a loop-free code by predicating each instruction with a predicate with a boolean variable (or its negation) computed earlier in the code. We then place the instructions in a graph that represents dependencies. Figure 7.2 shows the example program of Figure 7.1 represented with our approach.

Alongside the dependency graph, we maintain a list of relations between predicates of the form $p = \bigsqcup_{i<k} p_i$. Such a relation indicates that:

$$p \iff \exists i < k.\ p_i \qquad\qquad \forall i, j < k.\ \neg p_i \vee \neg p_j \tag{7.1}$$

We combine these relations to answer queries about the disjointedness of two predicates (that is if they can be simultaneously true or not), and the implication of a predicate by a disjunction of others. In our

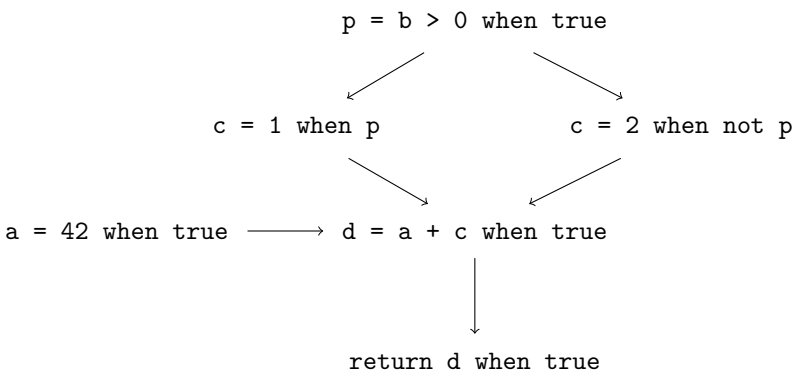


Figure 7.2: Example of Predicate-Based Control Flow Representation

case, the only relation we have is the tautology $\top = p \sqcup \neg p$, but relations can get more complex with more predicates.

Note that these relations perfectly encode the information we could obtain from the CFG of the original code. Indeed, the relation p and $(p_i)_{i<k}$ can be interpreted by the fact that p is the predicate of a basic block, and that they $(p_i)_{i<k}$ are the predicates of the edges starting from (or ending in) the basic block. We are still working on finding the best data structure to represent these constraints and query them.

Control Flow Graph Reconstruction. Our representation offers flexibility to reorder instructions and move instructions in and out of if-conditions. Any order that satisfies data dependencies is valid. A decision space based on this representation could expose a choice `predicate` that selects the predicate of each instruction and a choice `order` that specifies the pairwise ordering of instructions, similar to the `order` choice presented in Chapter 4.

We have developed an algorithm that builds a control flow graph given the predicate of each instruction and a total order of instructions. The strength of this approach is that can encode the structure of any loop-free control flow graph with the right instructions, predicates and order. In particular, it can represent irregular control flows, that cannot be expressed with nested if-then-else conditions.

Limitations and Future Work. This work builds on a long-standing research effort to unify data and control flow into a single data-flow intermediate representation. The Program Dependence Graph [59] tried combining data-flow and control by introducing predicates in a program representation. However, it cannot handle all irregular control flow graphs. More recently, Gated SSA extend a data-flow representation to explicitly choose between two possible versions of a variable where the control flow merges [60, 61]. While it simplifies code analysis, Gated SSA fails to regenerate efficient imperative code as it relies on lazy evaluation for its semantics. To solve this problem, the Program Dependence Web adds control information where the control flow splits. However, it suffers from the same issues with irregular control flow graphs as the Program Dependency Graph [61].

While our representation supports arbitrary acyclic control flow graphs, further work is required to understand how to support loops. One idea would be to specify the loop structure using the polyhedral model and the local structure of the code with the predicates approach. The difficulty is to represent the control flow at the border of iterations without limiting ourselves to hierarchical blocks of code with a single entry point and a single exit point.

7.2.2 Performance Model Gradient

Chapter 5 presented a performance model that computes a lower bound for the execution time of every implementations that derive from a candidate. We propose to modify this model so that it also indicates which decision is likely to increase the bound the most. The idea is that such decisions are more discriminative and that search algorithms should thus instantiate the corresponding choices first.

Finding the exact decision that would increase the bound the most is infeasible in practice. We can, however, find a decision that is likely to increase the bound more than most others. To this end, we annotate every intermediate variable x that the performance model computes with a decision d_x that would increase it and a lower bound δ_x of the impact of d_x on x. If x directly depends on a set decisions, we separately instantiate each of these decisions to find how they impact x. Otherwise, we can write $x = f(x_0, \ldots, x_{n-1})$ were $(x_i)_{i<n}$ are previously computed variables. In that case, we independently compute for each $i < n$ the effect of d_{x_i} on x based on the value of δ_{x_i}. Note that d_{x_i} might also have an impact on the $(x_j)_{j \neq i}$ that we do not model.

Let us now list the different components of the model from which the decision with the maximal impact may originate.

- When computing `latency`, the decision with the most impact on the bound can either originate from the individual latency and usage of hardware resources of statements, the fusion of statements (in equations (5.13) and (5.14)), the size of dimensions (in equation (5.15)) or from the nesting order of statements (in equations (5.15) and (5.17)).

- When computing the lower bound B_i on the execution time of parallelism level i, the decision can originate from the amount of hardware parallelism μ_i available at level i, from the amount of software parallelism available at level i (η_i^{max} and η_i^{min}) or from the bound B_{i-1} at level $i-1$, B_{i-1}.

- When computing the bound B_0 on the execution time of a single thread, the model computes longest paths in graphs. The same strategy as proposed above applies to find the decision that is likely to have the most impact on the longest path: it applies to each intermediate step of the longest path algorithm. The decision can originate from a modification of the weight of an existing edge between program points or from an additional edge. In the first case, the decision may originate from `latency` or from the individual latency of instructions while in the second case, it originates directly from the `order` choice.

7.2.3 Search Algorithms

Finally, we discuss our ideas to build a better search algorithm that fully exploits the benefits of candidates. We first mention how genetic algorithms could help us find local optimum and how a statistical model could predict the performance of a partially specified candidate. We then discuss how we could improve the tree search algorithm and, finally, how we could automatically adapt the order of decisions to the context, to make most discriminative decisions first.

Neighbors of an Implementation and Genetic Algorithms. A first idea to improve the performance of the generated code is to leverage genetic algorithms to find local optimum. While our lower bound performance model is able to detect the most discriminative choices early, it is unable to differentiate between decisions that only have a small, incremental, effect on the performance. Small variations of the bound are dwarfed by the inaccuracy of the bound.

The problem for applying genetic algorithm to candidates is to define the concept of *neighbor* of an implementation. That is, the implementations that can be reached by making incremental changes to another implementation. Mutating the outcome of a single choice would most likely lead to an invalid candidate.

A first idea would be to iteratively reset the domain of choices that appear in constraint invalidated by the mutation until all constraints are respected, and then to randomly specify choices again. The order in which to reset domains could be randomized, so that domains have an equal opportunity to be reset when multiple resets may satisfy a constraint.

Predictive Performance Modeling. A second idea would be to develop a machine learning based performance model, that models the expected value of the execution time rather than a lower bound. This model could operate either on fully specified implementations, to avoid evaluating slow implementations on the hardware, or on partially specified implementations to guide the search.

The development of such a model raises several questions. First *what features would it relies on?*. A solution would be to directly use the domains of choices as input of the statistical model. However,

these attributes have no structure, making it hard to derive a bound. Another solution would be to rely on a combination of attributes from the intermediate results of the lower bound performance model, the domains, and the structure of the code.

The second question is *how to deal with open choices?*. Indeed, existing statistical models, such as Ithemal [49] for the performance of single basic blocks, tend to follow the structure of the code. But with open choices, this structure might only be partially specified. A solution might be to do what the performance model of Chapter 5 already does: only require the code structure to be locally consistent and consider groups of dimensions (loop levels following our terminology) to account for unordered loop nests.

The last question is *how to develop a performance model that is independent of the kernel to implement?*. Indeed, a model that depends on the kernel would need to be learned at each run of the search algorithm, thus slowing down the search. A solution might be to rely on domain adaptation techniques to transfer the knowledge from one kernel to the others.

Reinforcement Learning. The problem of choosing actions to perform (e.g. implementation decisions) in an environment (e.g. the decisions made so far or the domain of the current candidate) to maximize a value (e.g. the performance of the kernel) is not limited to optimizing compilation. The process of automatically finding the actions to perform based on the impact of previous actions is called *reinforcement learning*.

A third direction of research would be to exploit the recent breakthrough in the field of reinforcement learning to drive the search towards the most efficient implementations. The first difficulty is that most of the research in the area focuses on adversarial contexts, where other players also take actions. In particular, our search is totally deterministic while adversaries introduce a random component. Another difference is that these algorithms optimize for the average outcome of the actions while we only care about the best implementation found.

Decisions Ordering. Finally, an algorithm could better exploit the commutativity of decisions. Section 6.3.3 already showed that the order of decisions has a significant impact on the performance of the search. Unfortunately, the order currently used is fixed upfront and thus cannot adapt to the kernel or to the previous decisions. Moreover, it operates at the granularity of generic choices instead of individual decisions.

Ideally, the search algorithm should automatically detect the most discriminative decisions and instantiate them first. For this, it could measure the variance relative to each decision on previous evaluations of implementations. Alternatively, it could rely on the performance model gradient, as explained in Section 7.2.2.

Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*. 2012.

[2] NVIDIA cuDNN GPU accelerated deep learning. `https://developer.nvidia.com/cudnn`.

[3] Ulysse Beaugnon, Alexey Kravets, Sven van Haastregt, Riyadh Baghdadi, David Tweed, Javed Absar, and Anton Lokhmotov. Vobla: A vehicle for optimized basic linear algebra. In *ACM SIGPLAN Notices*, volume 49. ACM, 2014.

[4] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. SPIRAL: Code generation for DSP transforms. *IEEE*, 93(2), 2005.

[5] Daniele G Spampinato and Markus Püschel. A basic linear algebra compiler. In *Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. ACM, 2014.

[6] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017.

[7] Henry Massalin. Superoptimizer: a look at the smallest program. In *ACM SIGARCH Computer Architecture News*, volume 15, pages 122–126. IEEE Computer Society Press, 1987.

[8] Rajeev Joshi, Greg Nelson, and Keith Randall. *Denali: a goal-directed superoptimizer*, volume 37. ACM, 2002.

[9] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 297–310. ACM, 2016.

[10] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[11] Guido Tack. *Constraint propagation - models, techniques, implementation*. PhD thesis, Saarland University, Germany, 2009.

[12] G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R.M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, January 2005.

[13] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[14] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*, pages 359–371. Elsevier, 2011.

[15] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. Skelcl-a portable skeleton library for high-level gpu programming. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176–1182. IEEE, 2011.

[16] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):134, 2014.

[17] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. *ACM SIGPLAN Notices*, 46(8):47–56, 2011.

[18] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[19] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. Nova: A functional language for data parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, page 8. ACM, 2014.

[20] Cliff Click and Keith D Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(2):181–196, 1995.

[21] Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, and Albert Cohen. Optimization space pruning without regrets. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017. ACM, 2017.

[22] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *ACM SIGPLAN Notices*, volume 44, pages 264–276. ACM, 2009.

[23] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. 2012.

[24] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, 2018.

[25] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical report, 1998.

[26] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[27] Paul Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992.

[28] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International journal of parallel programming*, 21(6):389–420, 1992.

[29] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *ACM SIGPLAN Notices*, volume 49, pages 396–407. ACM, 2014.

[30] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM Sigplan Notices*, 41(11):404–415, 2006.

[31] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Acm Sigplan Notices*, volume 43, pages 101–113. ACM, 2008.

[32] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In *Proceedings of the 27th International Conference on Compiler Construction (CC)*, Vienna, Austria, 2018.

[33] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 90–100, 2008.

[34] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, Jagannathan Ramanujam, Ponnuswamy Sadayappan, and Nicolas Vasilache. Loop transformations: convexity, pruning and optimization. In *ACM SIGPLAN Notices*, volume 46, pages 549–562. ACM, 2011.

[35] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[36] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, March 2008.

[37] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[38] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[39] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[40] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.

[41] Yinan Li, Jack Dongarra, and Stanimire Tomov. A note on auto-tuning GEMM for GPUs. In *International Conference on Computational Science, ICCS'09*. Springer, May 25-27 2009.

[42] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.

[43] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. Diesel: Dsl for linear algebra and neural net computations on gpus. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 42–51, New York, NY, USA, 2018. ACM.

[44] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 2009.

[45] Junjie Lai and André Seznec. Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE Computer Society, 2013.

[46] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[47] Sara S Baghsorkhi, Matthieu Delahaye, Sanjay J Patel, William D Gropp, and Wen-mei Hwu. An adaptive performance modeling tool for GPU architectures. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 45, pages 105–114. ACM, 2010.

[48] Mehrzad Samadi, Amir Hormati, Mojtaba Mehrara, Janghaeng Lee, and Scott Mahlke. Adaptive input-aware compilation for graphics engines. *ACM SIGPLAN Notices*, 47(6):13–22, 2012.

[49] Charith Mendis, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. *arXiv preprint arXiv:1808.07412*, 2018.

[50] Frédéric de Mesmay, Arpad Rimmel, Yevgen Voronenko, and Markus Puschel. Bandit-based optimization on graphs with application to library performance tuning. ICML '09. ACM, 2009.

[51] NVIDIA cuBLAS GPU accelerated linear algebra. `https://developer.nvidia.com/cublas`.

[52] Philip Kilby, John Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2*, AAAI'06, pages 1014–1019. AAAI Press, 2006.

[53] Pang C Chen. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, 21(2):295–315, 1992.

[54] Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[55] Keith D Cooper, Philip J Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *ACM SIGPLAN Notices*, volume 34, pages 1–9. ACM, 1999.

[56] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[57] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[58] Cliff Click and Keith D Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17, 1995.

[59] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9, 1987.

[60] Robert Cartwright and Mattias Felleisen. The semantics of program dependence. In *Conference on Programming Language Design and Implementation*, PLDI '89, 1989.

[61] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Conference on Programming Language Design and Implementation*, PLDI '90, 1990.

Appendix A

Proofs of the Performance Model

This appendix provides the proofs of the properties and theorems we make in Section 5.2, when presenting the presenting the performance model of a single thread. We first present some notations and some demonstrate basic lemmas in Section A.1. We then give the proofs relative to the program order in Section A.2 and the proofs of the properties of execution graphs in Section A.3. Finally, we show that any path G_c has a path of same weight that only visit concrete execution points and that the performance model algorithm indeed computes the weight of a path in the execution graph in Section A.5.

A.1 Notations and Basic Properties

In the following sections, we consider c a candidate and c' a fixed candidate that derives from c. We denote $\mathcal{I}'$ and $\mathcal{I}'$ the instructions, $\mathscr{D}$ and $\mathscr{D}'$ the dimensions, $\mathscr{S}$ and $\mathscr{S}'$ the statements, $\mathbb{L}$ and $\mathbb{L}'$ the levels, $\mathbb{P}$ and $\mathbb{P}'$ the program points, $\mathbb{I}$ and $\mathbb{I}'$ the indexes, $\mathbb{E}$ and $\mathbb{E}'$ the execution points and $\preceq$ and $\preceq'$ the program orders of c and c' respectively. We also denote $\texttt{order}'$ the domain of the `order` choice for c'.

Before jumping to the proofs, Section A.1.1 points out some useful results about the relation between c and c'. Then, Section A.1.2 discuss the properties we can expect from `order`.

A.1.1 Relation Between c and c'

Candidates follow the formalism of Chapter 3. In this formalism, specifying decisions can only restrict the domain of choices. Restricting the domain of a choice can trigger the lowering of a high level construct, but this can only add new objects to the set representing each property that the kernel exposes. We thus have the following remark.

Remark A.1. *Specifying decisions cannot suppress statements, instructions or dimensions. Thus $\mathcal{I} \subseteq \mathcal{I}'$, $\mathscr{D} \subseteq \mathscr{D}'$ and $\mathscr{S} \subseteq \mathscr{S}'$. Moreover, decisions may only restrict domains. Thus, for any $s_1, s_2 \in \mathscr{S}$ distinct, $\texttt{order}(s_1, s_2) \subseteq \texttt{order}'(s_1, s_2)$.*

The first corollary of this, is that the sets of levels increases. Indeed,

$$\begin{aligned}\mathbb{L} &= \{L \subseteq \mathscr{D} \mid \forall d_1, d_2 \in L.\ d_1 = d_2 \vee \texttt{order}(d_1, d_2) \subseteq \{\texttt{MERGED}, \texttt{IN}, \texttt{OUT}\}\} && \text{(A.1)}\\ &\subseteq \{L \subseteq \mathscr{D}' \mid \forall d_1, d_2 \in L.\ d_1 = d_2 \vee \texttt{order}'(d_1, d_2) \subseteq \{\texttt{MERGED}, \texttt{IN}, \texttt{OUT}\}\} && \text{(A.2)}\\ &= \mathbb{L}' && \text{(A.3)}\end{aligned}$$

By definition of program points, this gives us the following lemma.

Lemma A.1. *The sets of levels and program points increases when specifying decisions. That is, $\mathbb{L} \subseteq \mathbb{L}'$ and $\mathbb{P} \subseteq \mathbb{P}'$.*

A.1.2 Properties of the `order` Choice

Section 4.3.2 imposes constraints on the `order` choice. While propagators may not fully enforce constraints on partially specified choices, they are always exact on fixed domains. In particular, in c is a fixed candidate, then the following lemma holds.

Lemma A.2. *Let $s_1, s_2, s_3 \in \mathscr{S}$ be three distinct statements. If the domain of* `order` *is fixed, then it respects the following implications.*

$$\left.\begin{array}{r} \mathit{order}(s_1, s_2) \subseteq \{\texttt{BEFORE}, \texttt{IN}, \texttt{MERGED}\} \\ \mathit{order}(s_2, s_3) \subseteq \{\texttt{BEFORE}\} \end{array}\right\} \implies \mathit{order}(s_1, s_3) \subseteq \{\texttt{BEFORE}\} \tag{A.4}$$

$$\left.\begin{array}{r} \mathit{order}(s_1, s_2) \subseteq \{\texttt{BEFORE}\} \\ \mathit{order}(s_2, s_3) \subseteq \{\texttt{BEFORE}, \texttt{IN}, \texttt{MERGED}\} \end{array}\right\} \implies \mathit{order}(s_1, s_3) \subseteq \{\texttt{BEFORE}, \texttt{IN}\} \tag{A.5}$$

$$\left.\begin{array}{r} \mathit{order}(s_1, s_2) \subseteq \{\texttt{BEFORE}, \texttt{IN}, \texttt{OUT}, \texttt{MERGED}\} \\ \mathit{order}(s_2, s_3) \subseteq \{\texttt{BEFORE}\} \end{array}\right\} \implies \mathit{order}(s_1, s_3) \subseteq \{\texttt{BEFORE}, \texttt{OUT}\} \tag{A.6}$$

$$\left.\begin{array}{r} \mathit{order}(s_1, s_2) \subseteq \{\texttt{BEFORE}, \texttt{IN}, \texttt{OUT}, \texttt{MERGED}\} \\ \mathit{order}(s_2, s_3) \subseteq \{\texttt{BEFORE}, \texttt{IN}, \texttt{OUT}, \texttt{MERGED}\} \end{array}\right\} \implies \mathit{order}(s_1, s_3) \subseteq \{\texttt{BEFORE}, \texttt{IN}, \texttt{OUT}, \texttt{MERGED}\} \tag{A.7}$$

$$\left.\begin{array}{r} \mathit{order}(s_1, s_2) \subseteq \{\texttt{BEFORE}, \texttt{OUT}, \texttt{MERGED}\} \\ \mathit{order}(s_2, s_3) \subseteq \{\texttt{BEFORE}, \texttt{OUT}\} \end{array}\right\} \implies \mathit{order}(s_1, s_3) \subseteq \{\texttt{BEFORE}, \texttt{OUT}\} \tag{A.8}$$

$$\left.\begin{array}{r} \mathit{order}(s_1, s_2) \subseteq \{\texttt{BEFORE}\} \\ \mathit{order}(s_2, s_3) \subseteq \{\texttt{BEFORE}, \texttt{OUT}, \texttt{MERGED}\} \end{array}\right\} \implies \mathit{order}(s_1, s_3) \subseteq \{\texttt{BEFORE}\} \tag{A.9}$$

$$\left.\begin{array}{r} \mathit{order}(s_1, s_2) \subseteq \{\texttt{BEFORE}, \texttt{OUT}, \texttt{MERGED}\} \\ \mathit{order}(s_2, s_3) \subseteq \{\texttt{BEFORE}, \texttt{OUT}, \texttt{MERGED}\} \end{array}\right\} \implies \mathit{order}(s_1, s_3) \subseteq \{\texttt{BEFORE}, \texttt{OUT}, \texttt{MERGED}\} \tag{A.10}$$

$$\left.\begin{array}{r} \mathit{order}(s_1, s_2) \subseteq \{\texttt{OUT}, \texttt{MERGED}\} \\ \mathit{order}(s_2, s_3) \subseteq \{\texttt{OUT}\} \end{array}\right\} \implies \mathit{order}(s_1, s_3) \subseteq \{\texttt{OUT}\} \tag{A.11}$$

These properties are obtained by enumerating all possible assignments of $\texttt{order}(s_1, s_2)$, $\texttt{order}(s_2, s_3)$ and $\texttt{order}(s_1, s_3)$ and filtering out the ones that do not respects constraints (4.3), (4.4) and (4.7). When applying constraints, we use the fact that `order` is antisymmetric to filter out more assignments.

A.2 Program Order

This section gives the proofs for Section 5.2.1. Section A.2.1 first shows $\preceq$ is indeed a partial order. Section A.2.2 then shows that it is a total order on concrete execution points. Finally, Section A.2.3 shows that the $\preceq$ relation only imposes ordering constraints that are valid for any implementation $c' \in [\![c]\!]$ that derives from c.

A.2.1 The Program Order is a Partial Order

We first show that $\preceq$ is a partial order. That is, that it is a reflexive, transitive and antisymmetric relation. The reflexivity derives directly from the definition of $\preceq$. We prove below that it is transitive and antisymmetric.

Transitive. Let $p_1, p_2, p_3 \in \mathbb{P}$ such that $p_1 \preceq p_2$ and $p_2 \preceq p_3$. Let us show that $p_1 \preceq p_3$. For that, we need to show that it respects the five conditions of equation (5.32). The first four conditions applied to $p_1 \preceq p_2$ and $p_2 \preceq p_3$ gives us the first four conditions for $p_1 \preceq p_3$:

$$\texttt{before}(p_1) \subseteq \texttt{before}(p_2) \subseteq \texttt{before}(p_3) \tag{A.12a}$$

$$\texttt{before}(p_1) \cup \texttt{current}(p_1) \subseteq \texttt{before}(p_2) \cup \texttt{current}(p_2) \subseteq \texttt{before}(p_3) \cup \texttt{current}(p_3) \tag{A.12b}$$

$$\texttt{after}(p_1) \supseteq \texttt{after}(p_2) \supseteq \texttt{after}(p_3) \tag{A.12c}$$

$$\texttt{after}(p_1) \cup \texttt{current}(p_1) \supseteq \texttt{after}(p_2) \cup \texttt{current}(p_2) \supseteq \texttt{after}(p_3) \cup \texttt{current}(p_3) \tag{A.12d}$$

For the last condition, we reason by contradiction and assume it is violated. Lets assume $p_1 = X_\emptyset$ and $p_3 = E_\emptyset$. This gives us $\texttt{before}(p_1) = \mathscr{S}$ and $\texttt{before}(p_3) = \emptyset$. But then, according to (A.12a), $\texttt{before}(p_1) \subseteq \texttt{before}(p_3)$. Thus $\mathscr{S} = \emptyset$ and $\mathbb{P} = \{E_\emptyset, X_\emptyset\}$. Thus $p_2 \in \{p_1, p_2\}$. This is a contradiction as $p_1 \preceq p_2$ and $p_2 \preceq p_3$, but $p_1 \not\preceq p_3$ as (p_1, p_3) does not respect the third condition.

Antisymmetric. Let $p_1, p_2 \in \mathbb{P}$ such that $p_1 \preceq p_2$ and $p_2 \preceq p_1$. Let us show that $p_1 = p_2$. For this, we first show how we can uniquely identify a program point $p \in \mathbb{P}$ from $\texttt{before}(p)$, $\texttt{current}(p)$ and $\texttt{after}(p)$.

- If p in an instruction, then $\texttt{current}(p) \cap \mathcal{I} = \{i\}$, otherwise $\texttt{current}(p) \cap \mathcal{I} = \emptyset$. Thus, $p = i \in \mathcal{I}$ if and only if $\texttt{current}(p) \cap \mathcal{I} = \{i\}$.

- If p is the entry point E_L of a loop level $L \in \mathbb{L}$, then $\texttt{after}(p) \cap \texttt{current}(p) = L$. Otherwise $\texttt{after}(p) \cap \texttt{current}(p) = \emptyset$. Thus, for $L \neq \emptyset$, $p = E_L$ if and only if $\texttt{after}(p) \cap \texttt{current}(p) = L$.

- Similarly, if p is the exit point X_L of a loop level $L \in \mathbb{L}$, then $\texttt{before}(p) \cap \texttt{current}(p) = L$. Otherwise $\texttt{before}(p) \cap \texttt{current}(p) = \emptyset$. Thus, for $L \neq \emptyset$, $p = X_L$ if and only if $\texttt{after}(p) \cap \texttt{current}(p) = L$.

It is then a matter of applying these identification techniques to p_1 and p_2. By definition of $\preceq$, we have the following equalities.

$$\texttt{before}(p_1) = \texttt{before}(p_2) \tag{A.13}$$
$$\texttt{after}(p_1) = \texttt{after}(p_2) \tag{A.14}$$
$$\texttt{before}(p_1) \cup \texttt{current}(p_1) = \texttt{before}(p_2) \cup \texttt{current}(p_2) \tag{A.15}$$
$$\texttt{after}(p_1) \cup \texttt{current}(p_1) = \texttt{after}(p_2) \cup \texttt{current}(p_2) \tag{A.16}$$

By combining equations (A.15) and (A.16) with the fact that $\forall p \in \mathbb{P}.\ \texttt{before}(p) \cap \texttt{after}(p) = \emptyset$, we obtain the following equations:

$$(\texttt{before}(p_1) \cup \texttt{current}(p_1)) \cap (\texttt{after}(p_1) \cup \texttt{current}(p_1)) = \texttt{current}(p_1) \tag{A.17}$$
$$= (\texttt{before}(p_2) \cup \texttt{current}(p_2)) \cap (\texttt{after}(p_2) \cup \texttt{current}(p_2)) = \texttt{current}(p_2) \tag{A.18}$$

We then distinguish three cases depending on the value of p_1.

- If $p_1 = i \in \mathcal{I}$, then $\texttt{current}(p_2) \cap \mathcal{I} = \texttt{current}(p_1) \cap \mathcal{I} = \{i\}$. Thus $p_2 = i = p_1$.

- If there exists $L \in \mathbb{L}$ such that $p_1 = E_L$, then $\texttt{after}(p_2) \cap \texttt{current}(p_2) = \texttt{after}(p_1) \cap \texttt{current}(p_1) = L$. Thus, if $L \neq \emptyset$, then $p_2 = E_L = p_1$. Otherwise, $p_2 = E_\emptyset$ or $p_2 = X_\emptyset$. Because $p_2 \preceq p_1$, condition $(5.32e)$ in the definition of $\preceq$ ensures that $p_2 \neq X_\emptyset$. Thus, $p_2 = E_\emptyset = p_1$.

- Similarly, if there exists $L \in \mathbb{L}$ such that $p_1 = X_L$, then $p_1 = p_2$.

□

A.2.2 The Program Order is Total on Concrete Execution Points

Let us assume for this section that c is a fixed candidate. We want to prove that $\preceq$ restricted to concrete execution points is a total order that follows the structure of the generated code. For that, we consider $p_1, p_2 \in \mathbb{P}$ two concrete program points. We show that p_1 and p_2 are ordered by $\preceq$, and that this order matches their order in the generated code. We first treat the special cases where p_1 or p_2 are the entry point $E_\emptyset$ or the exit point $X_\emptyset$ of the entire kernel before handling the general case.

Entry and Exit Points. If p_1 is the entry point $E_\emptyset$ of the entire kernel, then $p_1 \preceq p_2$. Indeed, in that case, $\texttt{before}(p_1) = \texttt{current}(p_1) = \emptyset$ and $\texttt{after}(p_1) = \mathscr{S}$. Similarly, if p_1 is exit point of the entire kernel, then $p_2 \preceq p_1$ as $\texttt{before}(p_1) = \mathscr{S}$ and $\texttt{current}(p_1) = \texttt{after}(p_1) = \mathscr{S}$hus $p_2 \preceq p_1$. The same results apply if $p_2 = E_\emptyset$ or $p_2 = X_\emptyset$.

Merged Dimensions. We now assume p_1 and p_2 distinct from $E_\emptyset$ and $X_\emptyset$. Since they are both concrete execution points, they both relate to a single class of merged statements. We denote $S_1, S_2 \subseteq \mathscr{S}$ these classes of statements. We now consider five cases depending on the relative order of S_1 and S_2.

1. If $S_1 = S_2$, then either a) $p_1 = p_2$, b) $S_1 \in \mathbb{L}$, $p_1 = E_{S_1}$ and $p_2 = X_{S_1}$ or c) $S_1 \in \mathbb{L}$, $p_1 = X_{S_1}$ and $p_2 = E_{S_1}$. Case a) is trivial and case c) is the symmetric of b). We thus focus on b) and suppose

$p_1 = E_{S_1}$ and $p_x = X_{S_1}$. We then have $\texttt{before}(p_1) \subseteq \texttt{before}(p_2)$ as shown below.

$$\texttt{before}(p_1) = \left\{ s \in \mathscr{S} \;\middle|\; s \notin S_1 \wedge \bigcup_{d \in S_1} \texttt{order}(s,d) = \{\texttt{BEFORE}\} \right\} \tag{A.19}$$

$$\subseteq \left\{ s \in \mathscr{S} \;\middle|\; s \in S_1 \vee \bigcup_{d \in S_1} \texttt{order}(s,d) \subseteq \{\texttt{BEFORE}, \texttt{IN}\} \right\} \tag{A.20}$$

$$\subseteq \texttt{before}(p_2) \tag{A.21}$$

By symmetry, we also have $\texttt{after}(p_1) \supseteq \texttt{after}(p_2)$. Moreover $\texttt{current}(p_1) = \texttt{current}(p_2)$. Thus the entry point of the loop level S_1 is correctly ordered before its exit point.

2. If S_1 is ordered before S_2, we show that $\texttt{before}(p_1) \subseteq \texttt{before}(p_2)$ and $\texttt{before}(p_1) \cup \texttt{current}(p_1) \subseteq \texttt{before}(p_2) \cup \texttt{current}(p_2)$. By symmetry, we will also have the conditions on `after` and `current`, showing that $p_1 \preceq p_2$. Since S_1 is before S_2, we have:

$$\forall s_1 \in S_1, s_2 \in S_2.\ \texttt{order}(s_1, s_2) = \{\texttt{BEFORE}\} \tag{A.22}$$

Let $s \in \texttt{before}(p_1)$. Then, either:

- $s \in S_1$, in which case, we can directly apply (A.22) to obtain $s \in \texttt{before}(p_2)$, or
- for every $s_1 \in S_1$, $\texttt{order}(s, s_1) \subseteq \{\texttt{BEFORE}, \texttt{IN}\}$. In particular, since $S_1 \neq \emptyset$, there exists such an s_1. Combined with (A.22) and (A.4), this gives us:

  $$\forall s_2 \in S_2.\ \texttt{order}(s, s_2) = \{\texttt{BEFORE}\} \tag{A.23}$$

  Thus, $s \in \texttt{before}(p_2)$.

Thus $\texttt{before}(p_1) \subseteq \texttt{before}(p_2)$.

Similarly, let $s \in \texttt{current}(p_1)$. Then either,

- $s \in S_1$, in which case, we can directly apply (A.22) to obtain $s \in \texttt{before}(p_2)$, or
- for every $s_1 \in S_1$, $\texttt{order}(s, s_1) \subseteq \{\texttt{OUT}, \texttt{MERGED}\}$. In particular, since $S_1 \neq \emptyset$, there exists such an s_1. Combined with (A.22) and (A.8), this gives us:

  $$\forall s_2 \in S_2.\ \texttt{order}(s, s_2) = \{\texttt{OUT}\} \vee \texttt{order}(s, s_2) = \{\texttt{BEFORE}\} \tag{A.24}$$

  Thus, $s \in \texttt{before}(p_2) \cup \texttt{current}(p_2)$.

Thus $\texttt{before}(p_1) \cup \texttt{current}(p_1) \subseteq \texttt{before}(p_2) \cup \texttt{current}(p_2)$. Thus $p_1 \preceq p_2$.

3. If S_1 is ordered after S_2, this is the symmetric of case 2.

4. If S_1 is nested inside S_2, then $S_2 \in \mathbb{L}$ and either $p_2 = E_{S_2}$ or $p_2 = X_{S_2}$. We only treat the case $p_2 = E_{S_2}$ as the case $p_2 = X_{S_2}$ is similar. We show that in this case, we correctly have $p_2 \preceq p_1$. For that, we show that $\texttt{before}(p_2) \subseteq \texttt{before}(p_1)$ and $\texttt{before}(p_2) \cup \texttt{after}(p_2) \subseteq \texttt{before}(p_1) \cup \texttt{current}(p_1)$. By symmetry, we will also have the conditions on `after` and `current` showing that $p_2 \preceq p_1$. Since S_1 is nested inside S_2, we have:

   $$\forall s_1 \in S_1, s_2 \in S_2.\ \texttt{order}(s_2, s_1) = \{\texttt{OUT}\} \tag{A.25}$$

   Let $s \in \texttt{before}(p_2)$. Then, either:

   - $s \in S_2$, in which case we can directly apply (A.22) to obtain $s \in \texttt{before}(p_1)$, or
   - for every $s_2 \in S_2$, $\texttt{order}(s, s_2) = \{\texttt{BEFORE}\}$. In particular, since $S_2 \neq \emptyset$, there exists such an s_2. Combined with (A.25) and (A.9), this gives us:

     $$\forall s_1 \in S_1.\ \texttt{order}(s, s_1) = \{\texttt{BEFORE}\} \tag{A.26}$$

     Thus, $s \in \texttt{before}(p_1)$.

Thus $\texttt{before}(p_1) \subseteq \texttt{before}(p_2)$.

Similarly, let $s \in \texttt{current}(p_2)$. Then either,

- $s \in S_2$, in which case we can directly apply (A.22) to obtain $s \in \texttt{before}(p_1)$, or
- for all $s_2 \in S_2$, $\texttt{order}(s, s_2) \subseteq \{\texttt{OUT}, \texttt{MERGED}\}$. In particular, since $S_2 \neq \emptyset$, there exists such an s_2. Combined with (A.22) and (A.11), this gives us:

$$\forall s_1 \in S_1.\ \texttt{order}(s, s_1) = \{\texttt{OUT}\} \tag{A.27}$$

  Thus, $s \in \texttt{before}(p_1) \cup \texttt{current}(p_1)$.

Thus $\texttt{before}(p_2) \cup \texttt{current}(p_2) \subseteq \texttt{before}(p_1) \cup \texttt{current}(p_1)$. Thus $p_2 \preceq p_1$.

5. If S_1 is ordered outside of S_2, this is the symmetric of case 4

□

A.2.3 The Program Order Remains of c Remains Valid for c'

Finally, we show that $\preceq$ respects the program order of any implementation $c' \in [\![c]\!]$ that derives from c. Specifically, we want to prove that

$$\forall p_1, p_2 \in \mathbb{P}.\ p_1 \preceq p_2 \implies p_1 \preceq' p_2. \tag{A.28}$$

Let $p_1, p_2 \in \mathbb{P}$ such that $p_1, \preceq p_2$. We already showed in Lemma A.1 that $\mathbb{P} \subseteq \mathbb{P}'$. Thus $\preceq'$ is well defined for p_1 and p_2. We now show that p_1 and p_2 respect the five conditions of Definition 5.3.

We prove conditions (5.32a) and (5.32b). We omit the proof for (5.32c) and (5.32d) as they are similar to the two first. Condition (5.32e) is the same for c and c' and thus already enforced. To summarize, we need to show the following equations.

$$\texttt{before}'(p_1) \subseteq \texttt{before}'(p_2) \tag{A.29}$$

$$\texttt{before}'(p_1) \cup \texttt{current}'(p_1) \subseteq \texttt{before}'(p_2) \cup \texttt{current}'(p_2) \tag{A.30}$$

If $p_1 = E_\emptyset$, then $\texttt{before}'(p_1) = \texttt{current}'(p_1) = \emptyset$, and both properties are trivial. If $p_1 = X_\emptyset$, then $\texttt{before}(p_1) = \mathscr{S}$ and $\texttt{current}(p_1) = \texttt{after}(p_1) = \emptyset$. Thus, $p_2 \preceq p_1$. But $\preceq$ is antisymmetric, thus $p_1 = p_2$ and both properties are also trivial. We now suppose that $p_1 \neq E_\emptyset$ and $p_1 \neq X_\emptyset$. We distinguish three cases depending on the values of p_2.

- If there exists $i \in \mathcal{I}$ such that $p_2 = i$, then:

$$p_2 = i \in \texttt{current}(p_2) \subseteq \texttt{current}(p_2) \cup \texttt{after}(p_2) \subseteq \texttt{current}(p_1) \cup \texttt{after}(p_1) \tag{A.31}$$

  Then either $p_1 = i$, in which case the conditions are trivial, or $i \in \texttt{after}(p_1)$. We distinguish three additional cases depending on the value of p_1.

  - If $p_1 \in \mathcal{I}$, then $\texttt{order}'(p_1, p_2) = \{\texttt{BEFORE}\}$. The conditions then follow from the properties (A.4) and (A.8) of Lemma A.2.
  - If there exists $L \in \mathbb{L}$ such that $p_1 = E_L$, then $\forall d \in L.\ \texttt{order}'(d, p_2) \subseteq \{\texttt{BEFORE}, \texttt{OUT}\}$. In particular, there exists such d since $L \neq \emptyset$. The conditions then follow from the properties (A.9) and (A.10) of Lemma A.2.
  - If there exists $L \in \mathbb{L}$ such that $p_1 = X_L$, then $\forall d \in L.\ \texttt{order}'(d, p_2) = \{\texttt{BEFORE}\}$. In particular, there exists such d since $L \neq \emptyset$. The conditions then follow from the properties (A.4) and (A.8) of Lemma A.2.

- If there exists $L \in \mathbb{L}$ such that $p_2 = E_L$, then:

$$L \subseteq \texttt{after}(p_2) \subseteq \texttt{after}(p_1) \tag{A.32}$$

  We then distinguish three additional cases depending on the value of p_1.

- If $p_1 = i$, then $\forall d \in L.\ \texttt{order}'(p_1, p_2) = \{\texttt{BEFORE}\}$. The conditions then follow from the properties (A.4) and (A.8) of Lemma A.2.
- If there exists $L' \in \mathbb{L}$ such that $p_1 = E_{L'}$, then $\forall d \in L.\ d \in L' \vee \forall d' \in L'.\ \texttt{order}'(d', d) \subseteq \{\texttt{BEFORE}, \texttt{OUT}\}$. In particular, since L' is not empty, $\forall d \in L.\ d \in L' \vee \exists d' \in L'.\ \texttt{order}'(d', d) \subseteq \{\texttt{BEFORE}, \texttt{OUT}\}$. The conditions then follow from the properties (A.9) and (A.10) of Lemma A.2.
- If there exists $L' \in \mathbb{L}$ such that $p_1 = X_{L'}$, then $\forall d \in L, d' \in L'.\ d \neq d' \wedge \texttt{order}'(d', d) = \{\texttt{BEFORE}\}$. In particular, since L' is not empty, there exists such a d'. The conditions then follow from the properties (A.9) and (A.6) of Lemma A.2.

- If there exists $L \in \mathbb{L}$ such that $p_2 = X_L$, then:

$$L \subseteq \texttt{current}(p_2) \subseteq \texttt{current}(p_2) \cup \texttt{after}(p_2) \subseteq \texttt{current}(p_1) \cup \texttt{after}(p_1) \tag{A.33}$$

We then distinguish additional three cases depending on the value of p_1.

- If $p_1 \in \mathcal{I}$, then $\forall d \in L.\ \texttt{order}'(p_1, d) \subseteq \{\texttt{IN}, \texttt{BEFORE}\}$. The conditions then follow from the properties (A.5) and (A.7) of Lemma A.2.
- If there exists $L' \in \mathbb{L}$ such that $p_1 = E_{L'}$, then $\forall d \in L.\ d \in L' \vee \forall d' \in L'.\ \texttt{order}'(d', d) \subseteq \{\texttt{BEFORE}, \texttt{OUT}, \texttt{MERGED}\}$. In particular, since L' is not empty, $\forall d \in L.\ d \in L' \vee \exists d' \in L'.\ \texttt{order}'(d', d) \subseteq \{\texttt{BEFORE}, \texttt{OUT}, \texttt{MERGED}\}$. The conditions then follow from the properties (A.9) and (A.10) of Lemma A.2.
- If there exists $L' \in \mathbb{L}$ such that $p_1 = X_{L'}$, then $\forall d \in L.\ d \in L' \vee \forall d' \in L'; \texttt{order}'(d', d) \subseteq \{\texttt{BEFORE}, \texttt{IN}, \texttt{MERGED}\}$. In particular, since L' is not empty, $\forall d \in L.\ d \in L' \vee \exists d' \in L'.\ \texttt{order}'(d', d) \subseteq \{\texttt{BEFORE}, \texttt{IN}, \texttt{MERGED}\}$. The conditions then follow from the properties (A.5) and (**??**) of Lemma A.2.

The conditions are thus respected for any value of p_1 and p_2. □

A.3 Execution Graph Properties

This section provides the proof relative to the properties of execution graphs. Section A.3.1 proves Proposition 5.2, and Section A.3.2 proves Proposition 5.5.

A.3.1 The Execution Graph is Acyclic

We want to prove that G_c is acyclic. For that, we first define a partial order $<$ on execution points and then show that every edges $e_1 \xrightarrow{w} e_2$ in G_c goes from an execution point e_1 to a greater execution point e_2.

Execution Order. We define the *execution order* $<$ on execution points to match the order in which they execute. For this, we first define an order on indexes.

Let $\leq$ be an arbitrary total order on $\mathbb{L}$ such that for all $L_1, L_2 \in \mathbb{L}$, $L_1 \leq L_2 \implies E_{L_1} \preceq E_{L_2}$. Such an order exists as $\preceq$ is a partial order. We then define the partial order $\trianglelefteq$ on indexes as the lexicographical order on the $(I(L))_{L \in \mathbb{L}}$ ordered by $\leq$. Thus $\trianglelefteq$ is the lexicographical order on the tuples $(I(L_1), \ldots, I(L_n)) \in \mathbb{L}^n$, where $\forall i < n.\ L_i < L_{i+1}$ and $n = |\mathbb{L}|$.

We then define $\leq$ on execution points $(p, I) \in \mathbb{E}$ as the lexicographical order on first the index I and second the program point p. That is, for $(p_1, I_1), (p_2, I_2) \in \mathbb{E}$:

$$(p_1, I_1) \leq (p_2, I_2) \iff I_1 \triangleleft I_2 \vee (I_1 = I_2 \wedge p_1 \preceq p_2) \tag{A.34}$$

Edges Follow the Execution Order. Let $(p_1, I_1) \xrightarrow{w} (p_2, I_2)$ be an edge in G_c. We want to show that $(p_1, I_1) < (p_2, I_2)$. For that, we consider the different possible definitions of an edge in the definition of G_c. Each case bellow corresponds to the case with the matching number in Definition 5.6.

1. In the first case, $p_1 \prec p_2$ and $I_1 = I_2$. Then, by definition of $<$, $(p_1, I_1) < (p_2, I_2)$.

2. In the second case, $L \in \mathbb{L}$ such that $p_1 = E_L$ and $p_2 = X_L$ and $I_1 \leq I_2$. Thus, $p_1 \prec p_2$ and $I_1 \trianglelefteq I_2$ so $(p_1, I_1) < (p_2, I_2)$.

3. In the third case, p_1 and p_2 are instructions such that p_2 consumes the value that p_1 produces, and $I_1 \leq I_2$. Then $I_1 \trianglelefteq I_2$ and, since p_2 reads the value that p_1 produces, $p_1 \prec p_2$. Thus $(p_1, I_1) < (p_2, I_2)$.

4. In the fourth case, there exists $L \in \mathbb{L}$ such that $p_1 = X_L$, $p_2 = E_L$. Their also exists $\mathbb{L}_{\texttt{IN}} \subset \mathbb{L}$ a set of loop ordered within L such that I_2 respects the following definition.

$$\forall L' \in \mathbb{L}.\ I_2(L') := \begin{cases} I_1(L) + 1 & \text{if } L = L' \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I_1(L') & \text{otherwise} \end{cases} \tag{A.35}$$

   Let $L' \in \mathbb{L}$ such that $I_1(L') > I_2(L')$. Then, $L' \in \mathbb{L}_{\texttt{IN}}$ per the definition of I_2. But then $E_L \prec E_{L'}$ since $\mathbb{L}_{\texttt{IN}}$ contains loop levels nested inside L. Thus $\forall L' \in \mathbb{L}.\ I_1(L') \leq I_2(L') \vee L < L'$. Moreover $I_1(L) < I_2(L)$. Thus, per the definition of $\trianglelefteq$, $I_1 \triangleleft I_2$, and since p_2 reads the value p_1 produces, $p_1 \preceq p_2$. Thus $(p_1, I_1) < (p_2, I_2)$.

5. In the fifth case, $p_1 = p_2$ and there exists $L \in \mathbb{L}$ and $\mathbb{L}_{\texttt{IN}} \subset \mathbb{L}$ a set of loop levels ordered within L such that I_2 respects the following definition.

$$\forall L' \in \mathbb{L}.\ I_2(L') := \begin{cases} I_1(L) + 1 & \text{if } L = L' \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I_1(L') & \text{otherwise} \end{cases} \tag{A.36}$$

   Then for the same reasons than case 5, $(p_1, I_1) < (p_2, I_2)$.

6. The last case is the same than 5, but with $\mathbb{L}_{\texttt{IN}} = \emptyset$.

A.3.2 The Execution Graph Increases when Specifying Decisions

This section proves that there exists a function $f : \mathbb{E} \to \mathbb{E}'$ such that preserves the edges from G_c to $G_{c'}$:

$$\forall e_1, e_2 \in \mathbb{E}.\ e_1 \xrightarrow{w} e_2 \in G_c \implies \exists w' \geq w.\ f(e_1) \xrightarrow{w'} f(e_2) \in G_{c'}. \tag{A.37}$$

For that, we define a function $g : \mathbb{I} \to \mathbb{I}'$ and define $f : \mathbb{E} \to \mathbb{E}'$ for $(p, I) \in \mathbb{E}$ as $f(p, I) = (p, g(I))$. We thus need to prove

$$(p_1, I_1) \xrightarrow{w} (p_2, I_2) \in G_c \implies \exists w' \geq w.\ (p_1, g(I_1)) \xrightarrow{w'} (p_2, g(I_2)) \in G_{c'}. \tag{A.38}$$

Note that $\mathbb{P} \subseteq \mathbb{P}'$, so that $(p_1, g(I_1))$ and $(p_2, g(I_2))$ are indeed included in $\mathbb{E}'$. Moreover, $I_1 < I_2 \implies g(I_1) < g(I_2)$.

For every $I \in \mathbb{I}$, we define $g(I)$ as follows.

$$\forall L \in \mathbb{I}'.\ g(L) = \begin{cases} I(L) & \text{if } L \in \mathbb{L} \\ 0 & \text{otherwise} \end{cases} \tag{A.39}$$

This definition ensures that $g(I) \in \mathbb{I}'$. Indeed, for every $L \in \mathbb{I}$, $I(L) < \texttt{size}_{min}(L) \leq \texttt{size}'_{min}(L)$. We now prove that f preserves the edges for each case of the definition of G_c. Let $(p_1, I_1), (p_2, I_2) \in \mathbb{E}$ such that G_c has an edge of weight w from (p_1, I_1) to (p_2, I_2). Each case below corresponds to the case with the matching number in Definition 5.6.

1. In the first case, $p_1 \prec p_2$, $I_1 = I_2$ and $w = 0$. Since $p_1 \prec p_2$, $G_{c'}$ has an edge $(p_1, g(I_1)) \xrightarrow{0=w} (p_2, g(I_1)) = (p_2, g(I_2))$. Thus the edge is preserved.

2. In the second case, $I_1 \leq I_2$ (thus $g(I_1) \leq g(I_2)$) and there exists $L \in \mathbb{L}$ such that $p_1 = E_L$, $p_2 = X_L$ and $w = \texttt{latency}(i, L)$ where $i < k$ is a parallelism level that produces synchronization instructions. Then $L \in \mathbb{L}'$ since $\mathbb{L} \subseteq \mathbb{L}'$. Thus, there exists an edge $(p_1, g(I_1)) \xrightarrow{\texttt{latency}'(i,L)} (p_2, g(I_2))$ in $G_{c'}$. Moreover, $\texttt{latency}'(i, L) \geq \texttt{latency}(i, L) = w$, thus the edge is preserved.

3. In the third case, $p_1, p_2 \in \mathcal{I}$, $I_1 \leq I_2$ and there exists two series of dimensions $(d_i)_{i<n}$ and $(d'_i)_{i<n}$ such that p_2 reads the value produced by p_1 with point to point communication from dimensions $(d_i)_{i<n}$ to dimensions $(d'_i)_{i<n}$. There also exists $(L_i)_{i<n}$ and $(L'_i)_{i<n}$ two series of loop levels such that for every $i < n$, $d_i \in L_i$, $d'_i \in L_i$ and L_i and L'_i only contain dimensions that are necessarily merged with each other and for every $i < n$, $I_1(L_i) = I_2(L'_i)$ and either $I_1(L'_i) = I_1(L_i)$ or $I_1(L'_i) = 0$.

   Then $g(I_1) \leq g(I_2)$ and for every $i < n$, $g(I_1)(L_i) = g(I_2)(L'_i)$ and either $g(I_1)(L'_i) = g(I_2)(L'_i)$ or L'_i is exclusive in I_1. Thus $G_{c'}$ contains an edge $(p_1, g(I_1))$ to $(p_2, g(I_2))$ of weight $\texttt{inst\_latency}'(p_1) \geq \texttt{inst\_latency}(p_1)$ and the edge is preserved.

4. In the fourth case, $w = 0$ and there exists $L \in \mathbb{L}$ such that $p_1 = X_L$, $p_2 = E_L$ and L is exclusive in I_1. Their also exists a set of loop levels $\mathbb{L}_{\texttt{IN}}$ that are all nested inside L such that I_2 is defined as follows.

$$\forall L' \in \mathbb{L}.\ I_2(L') := \begin{cases} I_1(L) + 1 & \text{if } L = L' \\ 0 & \text{if } \exists\ i < n.\ L' = L_i \\ I_1(L') & \text{otherwise} \end{cases} \tag{A.40}$$

   By definition of g, this equation also holds if we replace I_1 by $g(I_1)$ and I_2 by $g(I_2)$ and L is also exclusive in g. Thus, there exists an edge $(p_1, g(I_1)) \xrightarrow{0} (p_2, g(I_2))$ in $G_{c'}$ and the edge is preserved.

5. In the fifth case, $p_1 = p_2 \in \mathcal{I}$, $w = \texttt{inst\_latency}(p1)$ and there exists $L \in \mathbb{L}$ such that p_1 reads the value it produces at the previous iteration of L and L is exclusive in I_1. Their also exists a set of loop levels $\mathbb{L}_{\texttt{IN}}$ that are all nested inside L such that I_2 is defined as follows.

$$\forall L' \in \mathbb{L}.\ I_2(L') := \begin{cases} I_1(L) + 1 & \text{if } L = L' \\ 0 & \text{if } \exists\ i < n.\ L' = L_i \\ I_1(L') & \text{otherwise} \end{cases} \tag{A.41}$$

   By definition of g, this equation also holds if we replace I_1 by $g(I_1)$ and I_2 by $g(I_2)$, and L is also exclusive in $g(I_1)$. Thus, there exists an edge $(p_1, g(I_1)) \xrightarrow{0} (p_2, g(I_2))$ in $G_{c'}$ and the edge is preserved.

6. In the last case, $p_1 = p_2$, $w = 0$ and there exists $L \in \mathbb{L}$ such that $I_2 = I_1[L/I_1(L) + 1]$. Thus there is an edge $(p_1, g(I_1)) \xrightarrow{w=0} (p_1, g(I_1)[L/g(I_1)(L) + 1] = (p_2, g(I_2))$ in $G_{c'}$ and the edge is preserved.

A.4 Virtual and Concrete Paths

This section proves Proposition 5.4. Lets suppose c is a fixed candidate and let $\mathbb{H}$ be the loop levels that represent classes of merged dimensions in c. We want to show that if P is a path in G_c, then there exists a path P', still in G_c, that only visits concrete execution points. For that, we first show that we can put P in a cannonical form. We then show how to transform a cannonical path P into a path P' that only iterates on loop levels of $\mathbb{H}$ and finally how to transform this path P' into a path that only visits concrete execution points.

Definition A.1 (Canonical Path)**.** Let P be a path in G_c. We say that P is canonical if every execution point (p, I) and every edge from (p_1, I_1) to (p_2, I_2) it visits respect the following conditions.

1. p is exclusive in I.
2. I is only positive on loop levels $L \in \mathbb{L}$ that start before p: $I(L) > 0 \implies E_L \preceq p$.
3. If the edge was defined from the second case of the definition of G_c, then $I_1 = I_2$,
4. If the edge was defined from the third case of the definition of G_c, then I_2 is minimal. Concretly, if $(L_i)_{i<n}$ and $(L'_i)_{i<n}$ are the set of merged dimensions along which p_1 and p_2 communicate, then I_2 is defined as follows.

$$\forall L \in \mathbb{L}.\ I_2(L) = \begin{cases} I_1(L_i) & \text{if } \exists i < n.\ L = L'_i \\ I_1(L) & \text{otherwise} \end{cases} \tag{A.42}$$

5. The edge cannot be defined from the last case of the definition of G_c.

Proposition A.1. *Let P be a path in G_c. Then there exists a cannonical path P' of same weight in G_c, that visists the same program points. Moreover, if for every execution point (p, I) visited by P, I is only positive on loop levels of $\mathbb{H}$, then this property is preserved in P'.*

The following proposition shows that it is possible to translate the iterations along arbitrary loop levels into iterations along loop levels of $\mathbb{H}$.

Proposition A.2. *Let P be a canonical path in G_c. Then there exists a path P' of same weight that only visits concrete execution points and for every execution point (p, I) visited by P', I is only positive on loop levels of $\mathbb{H}$.*

Finally, the last proposition shows that if the path P obtained in Proposition A.2 is canonicalized, it is possible modify P so that it only visits concrete execution points, thus proving the property that any path in G_c has a path of same weight in G_c that only visits concrete execution points.

Proposition A.3. *Let P be a canonical path in G_c that only visits concrete execution points and for every execution point (p, I) it visits, I is only positive on loop levels of $\mathbb{H}$. Then, there exists a path P' of same weight in G_c that only visits concrete execution points.*

We now prove each proposition in a dedicated section: Proposition A.1 in Section A.4.1, Proposition A.2 in Section A.4.2, and Proposition A.3 in Section A.4.3

A.4.1 Path Canonicalization

Let P be a path of weight w in G_c. We show, by induction on the length n on P, how to generate a canonical path P' of same weight in G_c, that visits the same program points than P. Moreover, we prove that if for every execution point (p, I) visited by P, I is only positive on loop levels of $\mathbb{H}$, then this property is preserved in P'. Finally, to help with the induction, we add the constraint that if I is the final index of P and I' the final index of P', then $I \leq I'$.

The case where $n = 0$ is trivial. Indeed, if $P = (p, I)$, then $P' = (p, I_0)$ where $I_0 \in \mathbb{I}$ is defined such that $\forall L \in \mathbb{L}.\ I_0(L) = 0$ is a solution. We now focus on the case where $n > 0$ and assume the property true for any path of length $< n$. Let $(p_1, I_1) \xrightarrow{w_1} (p_2, I_2)$ be the last edge of P. The induction hypothesis gives us a canonical path of weight $w - w_1$ that end in (p_1, I_1') such that $I_1' \leq I_1$. We now consider the different cases of the definition of the edge from (p_1, I_1) to (p_2, I_2) in the definition of G_c. Each case below corresponds to the same case in Definition 5.6.

First Case: Program Order Constraints. In that case, $I_1 = I_2$, $p_1 \prec p_2$ and $w_1 = 0$. Thus, there exists an edge of same weight from (p_1, I_1') to (p_2, I_1'). Moreover, I_1' respect the conditions of canonical paths according to the induction hypothesis. Thus, the path P' extended with (p_2, I_1') is a solution.

Second Case: Loop Level Latency. In that case, there exists $L \in \mathbb{L}$ and $i \in [\![0, k]\!]$ a parallelism level that produce synchronization operations such that $p_1 = E_L$, $p_2 = X_L$, $I_1 \leq I_2$ and $w_1 = \texttt{latency}(i, L)$. But then, there exists an edge from (p_1, I_1') to (p_2, I_1') of same weight. Moreover, I_1' respect the conditions of canonical paths according to the induction hypothesis. Thus, the path P' extended with (p_2, I_1') is a solution.

Third Case: Data Dependencies. In that case, $p_1, p_2 \in \mathcal{I}$ are instructions such that p_2 reads the value that p_1 produces, with point to point communication from dimensions $(d_i)_{i<n}$ to dimensions $(d_i')_{i<n}$ and $I_1 \leq I_2$. There also exists two series of loop levels $(L_i)_{i<n}$ and $(L_i')_{i<n}$ such that for every $i < n$, $d_i \in L_i$, $d_i' \in L_i'$, both L_i and L_i' only contains dimensions merged with each other, $I_1(L_i) = I_2(L_i')$ and either $I_1(L_i') = I_2(L_i')$ or L_i' is exclusive in I_1. Let I_2' be defined as follows.

$$\forall L \in \mathbb{L}.\ I_2'(L) = \begin{cases} I_1'(L_i) & \text{if } \exists i < n.\ L = L_i' \\ I_1'(L) & \text{otherwise} \end{cases} \tag{A.43}$$

With this definition, $I_2' \leq I_2$ and for every $i < n$, $I_1'(L_i) = I_2'(L_i')$. We now prove that $I_1' \leq I_2'$ and for every $i < n$ either $I_1'(L_i') = I_2'(L_i')$ or L_i' is exclusive in I_1'. We already now that for every loop level

that is not an L'_i, $I'_1(L) = I'_2(L)$. We now consider $i < n$ and distinguish two cases depending on if L'_i is exclusive or not in I_1 to prove that $I'_1(L'_i) \leq I'_2(L'_i)$ and either $I'_1(L'_i) = I'_2(L'_i)$ or L'_i is exclusive in I'_1. For that, we distinguish two cases depending on if L'_i is exclusive in I_1.

- If L'_i is exclusive in I_1, then L'_i also exclusive in I'_1 since $I'_1 \leq I_1$. Moreover, two dimensions with point to point communication are either merged or sequentially ordered. We thus have the following cases.
  - If $L_i = L'_i$, then $I'_1(L'_i) = I'_1(L_i) = I'_2(L'_i)$.
  - If L_i is merged with L'_i, then $I_1(L_i) = 0$ since L'_i is exclusive, thus $I_2(L'_i) = I_1(L_i) = 0$ and $I_1(L'_i) \leq I_2(L'_i) = 0$. Thus $I'_1(L'_i) \leq I_1(L'_i) = 0$. Thus $I'_1(L_i) = I'_2(L_i)$.
  - Otherwise, L'_i is after L_i. Thus $E_{L'_i} \not\prec p_1$ and $I'_1(L'_i) = 0$ according to the induction hypothesis. Thus $I'_1(L'_i) \leq I'_2(L'_i)$.
- Otherwise, $I_1(L'_i) = I_2(L'_i)$ and the induction hypothesis ensures that $I_1(L'_i) = 0$. But $I'_1 \leq I_1$ and $I'_2 \leq I_2$, thus $I'_1(L'_i) = 0 = I_2(L'_i)$ which proves both properties.

This proves that there is an edge from (p_1, I_1) to (p_2, I_2) of weight $w = \texttt{latency}(p_1)$ in G_c. Moreover, the index only increase along loop levels $(L'_i)_{i<n}$, and only if they are exclusive in I_1. Thus, the loop levels $L \in \mathbb{L}$ remain exclusive in I_2 whenever $I_2(L) > 0$. Finally, p_2 is nested into each L'_i. Thus, for every $i < n$, $E_{L'_i} \prec p_2$. Thus, the path P' extended with (p_n, I'_{m-1}) is a solution.

Fourth Case: Loop Iteration. In that case, there exists $L \in \mathbb{L}$ exclusive in I_1, such that $p_1 = X_L$, $p_2 = E_L$, $w_1 = 0$ and There exists a set of loop levels $\mathbb{L}_{\texttt{IN}}$ such that:

$$\forall L' \in \mathbb{L}.\ I_2(L') = \begin{cases} I_1(L') + 1 & \text{if } L' = L \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I_1(L') & \text{otherwise} \end{cases} \tag{A.44}$$

Let $I'_2 \in \mathbb{I}$ be defined as follows.

$$\forall L' \in \mathbb{L}.\ I'_2(L') = \begin{cases} I'_1(L') + 1 & \text{if } L' = L \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I'_1(L') & \text{otherwise} \end{cases} \tag{A.45}$$

Then, there exists an edge from (p'_1, I'_1) to (p'_2, I'_2) of weight 0 in G_c. Moreover, the index only increases along L, which is exclusive and verifies $E_L \preceq E_L = p_2$, and $I'_2 \leq I_2$. Thus, the path P' extended with (p_2, I'_2) is a solution.

Fifth Case: Loop-Carried Data Dependencies. In that case, $p_1 = p_2 \in \mathcal{I}$ is an instruction that reads the value it produces at the last iteration of a loop level $L \in \mathbb{L}$, exclusive in I_1. There exists a set of loop levels $\mathbb{L}_{\texttt{IN}}$ such that:

$$\forall L' \in \mathbb{L}.\ I_2(L') = \begin{cases} I_1(L') + 1 & \text{if } L' = L \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I_1(L') & \text{otherwise} \end{cases} \tag{A.46}$$

Let $I'_2 \in \mathbb{I}$ be defined as follows.

$$\forall L' \in \mathbb{L}.\ I'_2(L') = \begin{cases} I'_1(L') + 1 & \text{if } L' = L \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I'_1(L') & \text{otherwise} \end{cases} \tag{A.47}$$

Then there is also an edge from (p_1, I'_1) to (p_2, I'_2) of same weight. Moreover, the index only increases along L, which is exclusive and verifies $E_L \preceq p_2$, and $I'_2 \leq I_2$. Thus, the path P' extended with (p_2, I'_2) is a solution.

Thus, the path P' extended with (p_2, I'_2) is a solution.

Sixth Case: Iterations Order. In that case, there exists $L \in \mathbb{L}$ such that $p_1 = p_2$ and $I_2 = I_1[L/I_1(L)+1]$. In that case, the path P' is already a solution. □

A.4.2 Path Translation

Let $P = (p_0, I_0) \xrightarrow{w_0} \dots \xrightarrow{w_{n-1}} (p_n, I_n)$ be a canonical path in G_c. We define two functions $f : \mathbb{P} \to \mathbb{P}$ and $g : \mathbb{I} \to \mathbb{I}$ such that for every $i \leq n$, there exists a path P_i of of weight w from $(g(p_i), f(I_i))$ to $(g(p_{i+1}), f(I_{i+1}))$ in G_c and that for every program point (p, I) along P_i:

1. p is a concrete program point, and
2. for every $L \in \mathbb{L}$, $I_i(L) > 0 \implies L \in \mathbb{H}$.

We first give the definition of f and g and then prove that the path P_i exists for every $i \leq n$. This proves Proposition A.2, that any canonical path P can be transformed into a path P' of same weight that only visits concrete execution points and for every execution point (p, I) visited by P', I is only positive on loop levels of $\mathbb{H}$.

Definition of f and g

Let (p, I) be a program point in P. We define $f(p)$ as follows.

- If $p \in \mathcal{I}$, $p = E_\emptyset$ or $p = X_\emptyset$, then $f(p) = p$.
- If there exists $L \in \mathbb{L}$ such that $p = E_L$ and $L \neq \emptyset$, let $d \in L$ be the innermost loop of p and L_d be the loop level that corresponds to the class of merged dimensions of d. Then $f(p) = E_{L_d}$.
- Similarly, if there exists $L \in \mathbb{L}$ such that $p = X_L$ and $L \neq \emptyset$, then $f(p) = X_{L_d}$.

We define $g(I)$ such that $g(I)(L) = 0$ for every loop level $L \notin \mathbb{H}$. For $L \in \mathbb{H}$, there exists at most a single loop level $L' \in \mathbb{L}$ such that $I(L') > 0$ and $L \cap L' \neq \emptyset$. Indeed, if $I(L') > 0$ implies that L' is exclusive in I since P is canonical. If there is no such loop level L', $g(I)(L) = 0$. Otherwise, let $(H_i)_{i<n}$ be the classes of merged dimensions represented in L', sorted by nesting order starting from the innermost. Let $i_0 < n$ be the index of L in $(H_i)_{i<n}$. We define $g(I)(L)$ as follows.

$$g(I)(L) = \left\lceil \frac{I(L')}{\prod_{i<i_0} \texttt{size}(H_i)} \right\rceil \mod \texttt{size}(H_{i_0}) \tag{A.48}$$

This corresponds to the index along the i_0-th dimension of the loop nest represented by L'.

Existence of P_i

We now consider an edge from $(p_1, I_1) \xrightarrow{w} (p_2, I_2)$ in the path P. We show that there exists a path P' from $(f(p_1), g(I_1))$ to $(f(p_2), g(I_2))$ of same weight in G_c, such that for any execution point (p, I) along P'.

1. p is a concrete program point, and
2. for every $L \in \mathbb{L}$, $I_i(L) > 0 \implies L \in \mathbb{H}$.

From the definition of f and g, these properties are already respected for $(f(p_1), g(I_1))$ and for $(f(p_2), g(I_2))$. We thus only need to prove that there exists a path and that its internal execution point respect the properties. For that, we consider the different cases of the definition of the edge from (p_1, I_1) to (p_2, I_2) in the definition of G_c. Each case below corresponds to the same case in Definition 5.6. The sixth case does not occurs since P is canonical.

First Case: Program Order Constraints. In that case, $I_1 = I_2$, $p_1 \prec p_2$ and $w = 0$. Then, $f(p_1) \preceq f(p_2)$. To prove that, we consider different cases depending on the value of p_1 and p_2.

- If $p_1 = E_\emptyset$ or $p_2 = X_\emptyset$, the property is trivial. If $p_1, p_2 \in \mathcal{I}$, then $f(p_1) = p_1 \preceq p_2 = f(p_2)$ and the property is also trivial.
- If there exists $L \in \mathbb{L}$ such that $p_1 = E_L$ and $p_2 \in \mathcal{I}$, let $H \in \mathbb{H}$ be the innermost class of merged dimensions represented in L. Then $p_1 = E_H$. Since $E_L \preceq p_2$, p_2 is either nested inside or after H. Thus $f(p_1) = E_H \preceq p_2$. Similarly, if $p_1 \in \mathcal{I}$ and there exists $L \in \mathbb{L}$ such that $p_2 = X_L$, then $f(p_1) \preceq f(p_2)$.
- If $p_1 \in \mathcal{I}$ and there exists $L \in \mathbb{L}$ such that $p_2 = E_L$, let $H \in \mathbb{H}$ be the innermost class of merged dimensions represented in L. Then $p_2 = E_H$ and $E_L \preceq E_H$. Thus $p_1 \preceq E_L \preceq E_H$. Similarly, if there exists $L \in \mathbb{L}$ such that $p_1 = X_L$, and $p_2 \in \mathcal{I}$, then $f(p_1) \preceq f(p_2)$.
- If there exists $L, L' \in \mathbb{L}$ such that $p_1 = E_L$ and $p_2 = E_{L'}$, then either L, L' have the same innermost class of merged dimensions, in which case $f(p_1) = f(p_2)$ or, since $p_1 \prec p_2$, the innermost class $H \in \mathbb{H}$ of L is before or nested outside the innermost class $H' \in \mathbb{H}$ of L'. Thus $f(p_1) = E_H \prec E_{L_1} = f(p_2)$. Similarly, if there exists $L_1, L_2 \in \mathbb{L}$ such that $p_1 = X_{L_1}$ and $p_2 = X_{L_2}$, then $f(p_1) \preceq f(p_2)$.
- If there exists $L, L' \in \mathbb{L}$ such that $p_1 = X_L$ and $p_2 = E_{L'}$, let H be the innermost class of merged dimensions of L and H' the class of innermost merged dimensions of L'. Then, $f(p_1) = E_H \preceq E_L \prec X_{L'} \preceq E_{H'} = f(p_2)$.
- If there exists $L, L' \in \mathbb{L}$ such that $p_1 = E_L$ and $p_2 = X_{L'}$, let H be the innermost class of merged dimensions of L and H' the class of innermost merged dimensions of L'. Then, either $H = H'$, H and H' are nested in each other, or H is before H'. In all cases $f(p_1) = E_H \prec X_{H'} = f(p_2)$.

Since $f(p_1) \prec f(p_2)$ and $g(I_1) = g(I_2)$, there exists an edge of weight $w = 0$ from $(f(p_1), g(I_1))$ to $(f(p_2), g(p_2))$ in G_c. Thus P' exists and has no internal execution points.

Second Case: Loop Level Latency. In that case, there exists $L \in \mathbb{L}$ and $i \in [\![0, k]\!]$ a parallelism level that produce synchronization operations such that $p_1 = E_L$, $p_2 = X_L$, $I_1 \leq I_2$ and $w_1 = \texttt{latency}(i, L)$. The statements nested in the innermost class of merged dimensions $H \in \mathbb{H}$ of L are the same than the statements nested in L. Thus, $\texttt{latency}(i, H) = \texttt{latency}(i, L)$. Moreover $I_1 = I_2$ since P is canonical. Thus there is an edge from $(f(p_1), g(I_1)) = (E_H, g(I_1))$ to $(X_H, g(I_2)) = (f(p_2), g(I_2))$ of same weight in G_c. Thus P' exists and has no internal execution points.

Third Case: Data Dependencies. In that case, $p_1, p_2 \in \mathcal{I}$ are instructions such that p_2 reads the value that p_1 produces, with point to point communication from dimensions $(d_i)_{i<n}$ to dimensions $(d'_i)_{i<n}$ and $I_1 \leq I_2$. There also exists two series of loop levels $(L_i)_{i<n}$ and $(L'_i)_{i<n}$ such that for every $i < n$, $d_i \in L_i$, $d'_i \in L'_i$, both L_i and L'_i only contains dimensions merged with each other and either $I_1(L'_i) = I_2(L'_i)$ or L'_i is exclusive in I_1. Moreover, since P is canonical, I_2 satisfies the following definition.

$$\forall L \in \mathbb{L}.\ I_2(L) = \begin{cases} I_1(L_i) & \text{if } \exists i < n.\ L = L'_i \\ I_1(L) & \text{otherwise} \end{cases} \tag{A.49}$$

For each $i < n$, let $H_i, H'_i \in \mathbb{H}$ the classes of merged dimensions of d_i and d'_i respectively. Since $g(I_1)$ is only positive on classes of merged dimensions, H'_i is exclusive in $g(I_1)$. We now show that $g(I_1) \leq g(I_2)$ and that $\forall i < n.\ g(I_1)(H_i) = g(I_2)(H'_i)$, thus proving that there exists an edge of weight w from $(f(p_1), g(I_1))$ to $(f(p_2), g(p_2))$ in G_c and that P' exists with no internal execution points.

We first prove that $I_1 \leq I_2$. Let $L \in \mathbb{L}$. If $L \notin \mathbb{H}$, $g(I_1)(L) = 0 = g(I_2)(L)$. Otherwise, $L \in \mathbb{H}$. Let $L' \in \mathbb{L}$ such that $I_1(L') > 0$ and $L \cap L' \neq \emptyset$. If there is no such L', then $g(I_1)(L) = 0 \leq g(I_2)(L)$. $g(I_1)(L)$ only depends on the value of $I_1(L')$. We distinguish two cases to prove that $g(I_1)(L) \leq g(I_2)(L)$.

- If there exists $i < n$ such that $L' = L'_i$, then $g(I_1)(L) = I_1(L') \leq I_2(L')$ since L'_i only contains representatives of a single class of merged dimensions. Moreover $I_1(L') > 0$ thus $I_2(L') > 0$ thus L' is exclusive in L_2 (since P is canonical) and $g(I_2)(L) = I_2(L')$. Thus $g(I_1)(L) \leq g(I_2)(L)$.

- Otherwise $I_2(L') = I_1(L') > 0$. But then L' is exclusive in I_2 (since P is canonical). Thus $g(I_2)(L)$ depends on $I_2(L')$ in the same way than $g(I_1)(L)$ depends on $I_1(L')$. Thus $g(I_1)(L) = g(I_2)(L)$.

Let $i < n$. We now prove that $g(I_1)(H_i) = g(I_2)(H'_i)$. If $g(I_2)(H'_i) = 0$, then the property is trivial since $g(I_1) \leq g(I_2)$. Then there exists a unique loop level $L \in \mathbb{L}$ such that $I_2(L) > 0$. We distinguish two cases depending on the value of L.

- If $L = L'_i$, then $g(I_2)(H'_i) = I_2(L'_i) = I_1(L_i)$. Moreover $g(I_2)(H'_i) > 0$ thus $I_1(L_i) > 0$ and L_i is exclusive in I_1. Thus $g(I_1)(L_i) = I_1(L_i) = I_2(L'_i) = g(I_2)(H'_i)$.
- Otherwise L is also distinct from every other L'_j with $j < n$ and $I_1(L) = I_2(L)$. Indeed, the dimensions $(d'_j)_{j<n}$ are all part of the iteration space of p_2 and thus cannot be merged. Then, since P is canonical and $I_2(L) > 0$, L is exclusive in I_1.

  According to the constraints on point to point communication, H_i and H'_i are either merged or H'_i after H_i. But H'_i cannot be after H_i thus $H_i = H'_i$. To prove that, we reason by the absurd and suppose H'_i after H_i. Then H'_i is also after p_1. Thus $E_L \not\prec p_1$ since L contains a representative of H'_i. Since P is canonical, we thus have $I_1(L) = 0$ which is absurd since $I_1(L) = I_2(L) > 0$.

  We thus have $H_i = H'_i$ Then $g(I_1)(H_i)$ and $g(I_2)(H_i)$ are both computed from $I_1(L) = I_2(L)$ using the same formula. Thus $g(I_1)(H_i) = g(I_2)(H'_i)$.

Fourth Case: Loop Iteration. In that case, there exists $L \in \mathbb{L}$ exclusive in I_1, such that $p_1 = X_L$, $p_2 = E_L$, $w_1 = 0$ and there exists a set of loop levels $\mathbb{L}_{\texttt{IN}}$ such that:

$$\forall L' \in \mathbb{L}.\ I_2(L') = \begin{cases} I_1(L') + 1 & \text{if } L' = L \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I_1(L') & \text{otherwise} \end{cases} \tag{A.50}$$

Let $(H_i)_{i<n}$ be the classes or merged dimensions represented in L sorted by nesting order with the inner most class of dimensions first. Given the definition of f, there exists a single $i_0 < n$ such that:

$$\forall i < n.\ g(I_2)(H_i) = \begin{cases} 0 & \text{if } i < i_0 \\ g(I_1)(H_i) + 1 & \text{if } i = i_0 \\ g(I_1)(H_i) & \text{if } i > i_0 \end{cases} \tag{A.51}$$

Then $f(p_1) = X_{H_0} \preceq X_{H_{i_0}}$ and $E_{X_{H_{i_0}}} \preceq f(p_2) = E_{H_0}$. Let $\mathbb{L}'_{\texttt{IN}}$ be the classes of merged dimensions represented inside $\mathbb{L}_{\texttt{IN}}$, and the classes of merged dimensions the decomposition of L that are nested inside H_{i_0}.

$$\mathbb{L}'_{\texttt{IN}} := \{H_i \mid i < i_0\} \cup \{H \in \mathbb{H} \mid \exists L' \in \mathbb{L}_{\texttt{IN}}\ H \cap L' \neq \emptyset\} \tag{A.52}$$

Then $g(I_1)$ and $g(I_2)$ satisfy the following equation, which shows there exists and edge from $(X_{H_{i_0}}, g(I_1))$ to $(E_{H_{i_0}}, g(I_1))$ of weight 0 in G_c.

$$\forall L \in \mathbb{L}\ g(I_2)(L) = \begin{cases} 0 & \text{if } L \in \mathbb{L}'_{\texttt{IN}} \\ g(I_1)(L) + 1 & \text{if } L = L_{i_0} \\ g(I_1)(L) & \text{otherwise} \end{cases} \tag{A.53}$$

Thus G_c contains the path

$$P_i := (f(p_1), g(I_1)) \xrightarrow{0} (X_{H_{i_0}}, g(I_1)) \xrightarrow{0} (E_{H_{i_0}}, g(I_2)) \xrightarrow{0} (f(p_2), g(I_2)) \tag{A.54}$$

which satisfies the conditions. Indeed, $E_{H_{i_0}}$ and $X_{H_{i_0}}$ are concrete program points and the $g(I_1)$ and $g(I_2)$ already satisfy the conditions by definition of f.

Fifth Case: Loop-Carried Data Dependencies. In that case, $p_1 = p_2 \in \mathcal{I}$ is an instruction that reads the value it produces at the last iteration of a loop level $L \in \mathbb{L}$, exclusive in I_1. There exists a set of loop levels $\mathbb{L}_{\texttt{IN}}$ such that:

$$\forall L' \in \mathbb{L}.\ I_2(L') = \begin{cases} I_1(L') + 1 & \text{if } L' = L \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I_1(L') & \text{otherwise} \end{cases} \tag{A.55}$$

Then, similarly to the previous case, there exists an edge of same weight from $(f(p_1), g(I_1))$ to $(f(p_2), g(I_2)))$ in G_c.

□

A.4.3 Concrete Path Generation

Let P be a canonical path of weight w in G_c such that for every execution point $(p, I) \in \mathbb{E}$ that it visits, p is a concrete program points and $\forall L \in \mathbb{L}.\ I(L) > 0 \implies L \in \mathbb{H}$. We want to prove that there exists a path P' of same weight that only visits concrete execution points in G_c. Specifically, we need to show that for every point (p, I) that P' visits,

1. p is a concrete program point,
2. for every $L \in \mathbb{L}$, $I(L) > 0 \implies L \in \mathbb{H}$,
3. for every $L \in \mathbb{L}$, $p \prec E_L \implies I(L) = 0$, and
4. for every $L \in \mathbb{H}$, $X_L \prec p \implies I(L) = \texttt{size}(L) - 1$.

The three first points are already satisfied by P. We thus build a path P' that increases the indexes of the execution points visited by P. For that we reason be induction on P, starting from its tail. We build P' such that if (p_0, I_0) is the first execution point of P and (p'_0, I'_0) the first execution point of P', then $p_0 = p'_0$ and $I_0 \leq I'_0$. For the initialization case with a path P of length 0, $p'_0 = p_0$ and I'_0 defined as follows satisfies the conditions.

$$\forall L \in \mathbb{L}.\ I'_0(L) := \begin{cases} \texttt{size}(L) - 1 & \text{if } L \in \mathbb{H} \wedge X_L \prec p_0 \\ I_0(L) & \textit{otherwise} \end{cases} \tag{A.56}$$

Induction Case

We now focus on the induction case, where P has a strictly positive length. Let $(p_0, I_0) \xrightarrow{w_0} (p_1, I_1)$ be the first edge of P. Following the induction hypothesis, we can suppose there exists a path P'_1 of weight $w - w_0$ that starts at (p_1, I'_1) with $I'_1 \geq I_1$ and only visits concrete execution points. We define $I'_0 \in \mathbb{I}$ as follows.

$$\forall L \in \mathbb{L}.\ I'_0(L) := \begin{cases} \texttt{size}(L) - 1 & \text{if } X_L \prec p_0 \wedge L \in \mathbb{H} \\ I_0(L) & \text{otherwise} \end{cases} \tag{A.57}$$

Then (p_0, I'_0) is a concrete execution point and $I'_0 \geq I_0$. We now show that there exists a path P'_0 from $(p_0, I_0)'$ to (p_1, I'_1) of weight w_0 that only visits concrete execution points. Since we already showed that (p_0, I'_0) and (p_1, I'_1) are concrete, we only need to prove it for the internal execution points of P'_0. We consider separately the different cases of the definition of the edge from (p_1, I_1) to (p_2, I_2) in the definition of G_c. Each case below corresponds to the same case in Definition 5.6. The sixth case does not occurs since P is canonical.

1. In the first case, $I_0 = I_1$, $p_0 \prec p_1$ and $w_0 = 0$. Let $(H_i)_{i<n}$ be the list of loop levels of $\mathbb{H}$ such that $p_0 \prec X_{H_i} \prec p_1$. Let's suppose that $(H_i)_{i<n}$ is sorted such that $\forall i < n-1.\ X_{H_i} \prec X_{H_{i+1}}$. It is always possible to sort this list as the program order is total on concrete execution points. For each $i < n$, and j such that $I_0(H_i) \leq j \leq \texttt{size}(H_i) - 1$ we define $J_i^j \in \mathbb{I}$ as follows.

$$\forall L \in \mathbb{L}.\ J_i^j(L) := \begin{cases} \texttt{size}(L) - 1 & \text{if } L \in \mathbb{H} \wedge X_L \prec X_{H_i} \\ j & \text{if } \wedge L = H_i \\ I'_0(L) & \text{otherwise} \end{cases} \tag{A.58}$$

   Then for each $i < n$ and j such that $I_0(H_i) \leq j \leq \texttt{size}(H_i) - 1$, (X_{H_i}, J_i^j) is concrete and, if $j < \texttt{size}(H_i) - 1$, there is an edge of weight 0 from (X_{H_i}, J_i^j) to (X_{H_i}, J_i^{j+1}) in G_c. Moreover, for each $i < n-1$, $J_i^{\texttt{size}(H_i)-1} = J_{i+1}^{I_0(H_{i+1})}$ and $X_{H_i} \prec X_{H_{i+1}}$. Thus, there is an edge from of weight 0 from $(X_{H_i}, J_i^{\texttt{size}(H_i)-1})$ to $(X_{H_{i+1}}, J_{i+1}^{I_0(H_{i+1})})$ in G_c. The path P'_0 that starts in (p_0, I'_0) ends in (p_1, I'_1) and visits all the (X_{L_i}, I_i^j) along the way thus satisfies the conditions.

2. In the second case, there exists $L \in \mathbb{L}$ and $j \in [\![0, k]\!]$ a parallelism level that produce synchronization operations such that $p_0 = E_L$, $p_1 = X_L$, $I_0 \leq I_1$ and $w_0 = \texttt{latency}(j, L)$. Then, since $I'_0 \leq I'_1$, there is an edge of same weight from (p_0, I'_0) to (p_1, I'_1) in G_c.

3. in the third case, $p_0, p_1 \in \mathcal{I}$ such that p_1 reads the value that p_0 produces, with point to point communication from dimensions $(d_i)_{i<n}$ to dimensions $(d'_i)_{i<n}$ and $I_0 \leq I_1$ and $w_0 = \texttt{latency}(p_0)$. There also exists two series of loop levels $(L_i)_{i<n}$ and $(L'_i)_{i<n}$ such that for every $i < n$, $d_i \in L_i$, $d'_i \in L'_i$, both L_i and L'_i only contains dimensions merged with each other, $I_0(L_i) = I_1(L'_i)$ and either $I_0(L'_i) = I_0(L_i)$ or L'_i is exclusive in I_0. Finally, $I'_0 \leq I_1$.

   Then, for every $i < n$, since d_i and d'_i are in the iteration space of p_0 and p_1 respectively, $X_{L_i} \not\prec p_0$ and $X_{L'_i} \not\prec p_1$. Thus, $I'_0(L_i) \leq I'_1(L'_i)$. Moreover, since I'_0 is null for every loop level that is not a class of merged dimension, L'_i is exclusive in I'_0. Thus there exists an edge of weight w_0 from (p_0, I'_0) to (p_1, I'_1) in G_c.

4. In the fourth case, $w_0 = 0$ and there exists $L \in \mathbb{L}$ exclusive in I_0 such that $p_0 = X_L$ and $p_1 = E_L$. There also exists a set of loop levels $\mathbb{L}_{\texttt{IN}}$ ordered within L such that:

$$\forall L' \in \mathbb{L}.\ I_1(L') = \begin{cases} I_0(L') + 1 & \text{if } L' = L \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I_0(L') & \text{otherwise} \end{cases} \tag{A.59}$$

   Let $\mathbb{L}'_{\texttt{IN}}$ be the set of all dimensions ordered within L.

$$\mathbb{L}'_{\texttt{IN}} := \{L' \in \mathbb{L} \mid E_L \prec E_{L'} \prec X_{L'} \prec X_L\} \tag{A.60}$$

   Then the equation above is still satisfied when replacing I_0 with I'_0, I_1 with I'_1 and $\mathbb{L}_{\texttt{IN}}$ with $\mathbb{L}'_{\texttt{IN}}$. Thus there exists an edge of weight w_0 from (p_0, I'_0) to (p_1, I'_1) in G_c.

5. In the fifth case, $p_0 = p_1 \in \mathcal{I}$, $w_0 = \texttt{inst\_latency}(p_0)$ and their exists $L \in \mathbb{L}$ exclusive in I_0 such that p_n is an instruction that reads the value it produces at the previous iteration of L. There also exists a set of loop levels $\mathbb{L}_{\texttt{IN}}$ such that:

$$\forall L' \in \mathbb{L}.\ I_1(L') = \begin{cases} I_0(L') + 1 & \text{if } L' = L \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I_0(L') & \text{otherwise} \end{cases} \tag{A.61}$$

   Let $\mathbb{L}'_{\texttt{IN}}$ be the set of all loop levels ordered within L. Then the equation above is still satisfied when replacing I_0 with I'_0, I_1 with I'_1 and $\mathbb{L}_{\texttt{IN}}$ with $\mathbb{L}'_{\texttt{IN}}$. Thus there is an edge of weight w_0 from (p_0, I'_0) to (p_1, I'_1) in G_c.

□

A.5 Abstraction of the Execution Graph

This section proves Proposition 5.6. Let $\mathbb{L}_0$ be the set of loops level the lower bound algorithm considers and $\mathbb{P}'$ the set of program points restricted to only consider these loop levels. Let $D \subseteq \mathscr{D}$ and $p_1, p_2 \in \mathbb{P}'$ such that there exists a path from p_1 to p_2 in of weight w in G_c^D. Let $I_1, I_2 \in \mathbb{I}$ such that $I_1 \leq I_2$ and the following conditions are respected for every loop level $L \in \mathbb{L}$.

$$\forall i \in \{1, 2\}.\ L \subseteq D \wedge p_i \not\prec E_L \implies I_i(L) = \texttt{size}_{min}(L) - 1 \tag{A.62}$$

$$\forall i \in \{1, 2\}.\ L \cap D \neq \emptyset \wedge p_i \prec E_L \implies I_i(L) = 0 \tag{A.63}$$

$$L \cap D = \emptyset \implies I_1(L) = I_2(L) \tag{A.64}$$

We want to prove that there exists a path from (p_1, I_1) to (p_2, I_2) of weight w in G_c. For that, we reason by induction and suppose Proposition 5.6 is respected for every set of dimensions $D' \subset D$. We first show that proving Proposition 5.6 in the case where the path from p_1 to p_2 contains a single edge is enough to prove it for any path. We then prove the property for edges by considering each case of how an edge in G_c^D can be defined.

Restriction of the Problem to Edges. Let us suppose for a moment that Proposition 5.6 is proved for the case where the path from (p_1, I_1) to (p_2, I_2) contains a single edge. We now consider a path $P = p_0 \xrightarrow{w_0} \dots \xrightarrow{w_{n-1}} p_n$ of weight w in G_c^D and two indexes I_0, I_n that respect the conditions of Proposition 5.6. For each $0 < i < n$, we define $I_i \in \mathbb{I}$ as follows.

$$\forall L \in \mathbb{L}.\ I_i(L) = \begin{cases} \texttt{size}_{min}(L) - 1 & \text{if } L \in L \subseteq D \wedge E_L \preceq p_i \\ I_0(L) & \text{otherwise} \end{cases} \tag{A.65}$$

For every $i < n$, the path $p_i \xrightarrow{w_i} p_{i+1}$ and the indexes I_i and I_{i+1} respect the conditions of Proposition 5.6. Thus, since we assumed the property proved for paths containing a single edge, there exists a path of weight w_i from (p_i, I_i) to (p_{i+1}, I_{i+1}) in G_c. By concatenating the paths for each $i < n$, we obtain a path of weight w from (p_0, I_0) to (p_n, I_n) in G_c. We now prove the proposition for the case where P contains a single edge.

Proof for Edges. Let $p_1, p_2 \in \mathbb{P}'$ such that G_c^D contains an edge of weight w from p_1 to p_2. Let $I_1, I_2 \in \mathbb{I}$ be two indexes that respect the conditions of Proposition 5.6. We prove that there exists a path of weight w from (p_1, I_1) to (p_2, I_2) in G_c. For that we consider the different cases for the definition of an edge in the definition of G_c^D. Section A.5.1 gives the proof for the first case, that considers the ordering constraints and the pressure of loop levels. Section A.5.2 gives the proof for the second case, that considers data dependencies. Finally, Section A.5.3 gives the proof for the third case, that considers paths iterating along loops. We omit the proof for the fourth case, that considers loop-carried dependencies, as it is similar to the third case.

A.5.1 First Case: Edges Along a Fixed Index

In the first case, either $p_1 \preceq p_2$ and $w = 0$ or there exists $L \in \mathbb{L}$ and $i \in [\![0, k]\!]$ such that $p_1 = E_L$, $p_2 = X_L$ and $w = \texttt{latency}(i, L)$.

- In the first sub-case, $p_1 \preceq p_2$. Thus there exists an edge $(p_1, I_1) \xrightarrow{w=0} (p_2, I_1)$ in G_c, according to the item 1 of its definition.
- In the second sub-case, there is also an edge $(p_1, I_1) \xrightarrow{w=\texttt{latency}(i,L)} (p_2, I_1)$ in G_c, according to the item 2 of its definition.

Thus, in both case, there exists a path from (p_1, I_1) to (p_2, I_1) of weight w.

According to the hypothesis of Proposition 5.6, $I_1 \leq I_2$. Thus there exists a path from (p_2, I_1) to (p_2, I_2) of weight 0 in G_c, that follows the edges defined in the case 6 of the definition of G_c. By concatenating the path from (p_1, I_1) to (p_2, I_1) and the path from (p_2, I_1) to (p_2, I_2), we obtain a path of weight w from (p_1, I_1) to (p_2, I_2) in G_c.

A.5.2 Second Case: Data Dependencies

In the second case, p_1 and p_2 are instructions such that p_2 consumes the value that p_1 produces, $w = \texttt{inst\_latency}(p_1)$, and the dimensions $(d_i)_{i<m}$ and $(d'_i)_{i<m}$ along which p_1 and p_2 communicate are all included in D. Let $I'_1 \in \mathbb{I}$ be defined as follows.

$$\forall L \in \mathbb{L}.\ I'_1(L) = \begin{cases} I_1(\{d_i\}) & \text{if } \exists i < n.\ L = \{d'_i\} \\ I_1(L) & \text{otherwise} \end{cases} \tag{A.66}$$

We first show that $I_1 \leq I'_1$ and $I'_1 \leq I_2$, which proves that there exists two paths of weight 0, that follow the edges defined in the case 6 of the definition of G_c, from (p_1, I_1) to (p_1, I'_1) and from (p_2, I'_1) to (p_2, I_2). We then show that there exists an edge of weight w from (p_1, I'_1) to (p_2, I'_1) in G_c. The concatenation of the first path, the edge and the second path gives us a path of weight w from (p_1, I_1) to (p_2, I_2) in G_c.

Relative Order of I_1, I'_1 and $_2$. Let $L \in \mathbb{L}$. If there exists $i < n$ such that $L = \{d'_i\}$, then $I_2(L) = \texttt{size}_{min}(\{d'_i\}) - 1$. Indeed, p_2 is nested inside d'_i thus $E_{d_i} \prec p_2$. For the same reason, $I_1(\{d_i\}) = \texttt{size}_{min}(\{d_i\}) - 1$. Moreover, constraints force dimensions on which point to point communication occurs to have the same size. Thus, $\texttt{size}_{min}(\{d_i\}) = \texttt{size}_{min}(\{d'_i\})$. Thus,

$$I_1(\{d'_i\}) \leq \texttt{size}_{min}(\{d'_i\}) - 1 = \texttt{size}_{min}(\{d_i\}) - 1 = I_1(\{d_i\}) = I'_1(\{d'_i\}) \tag{A.67}$$
$$= I_2(\{d'_i\}) \tag{A.68}$$

Thus $I_1(L) \leq I'_1(L) = I'_2(L)$. Otherwise, $I'_1(L) = I_1(L) \leq I_2(L)$. Thus, in both cases, $I_1 \leq I'_1 \leq I_2$.

Edge From (p_1, I'_1) To (p_2, I'_1). I'_1 respects the conditions that $I'_1 \leq I'_1$, that for every $i < n$, $I'_1(\{d_i\}) = I'_1(\{d'_i\})$ and that $I'_1(\{d'_i\}) = I'_1(\{d_i\})$. Thus, according to the third case of the definition of the edges in the definition of G_c, there exists an edge from (p_1, I'_1) to (p_2, I'_2).

A.5.3 Third Case: Iteration Along a Loop

In the third case, there exists $L \in \mathbb{L}_0$ such that $L \cup D_L \subseteq D$, L only contains sequential dimensions, $\forall L' \in \mathbb{L}.\ L \cap L' \neq \emptyset \implies p_1 \prec E_{L'}$ and $p_2 = E_L$. Their also exists a path from p_1 to E_L of weight w_1 and a path from E_L to X_L of weight w_2 in $G_c^{D_L}$ such that $w = w_1 + (\texttt{size}_{min}(L) - 2).w_2$. For every $j < \texttt{size}_{min}(L)$, let $I_2^j, I_3^j \in \mathbb{I}$ be defined as follows for every loop level $L' \in \mathbb{L}$.

$$I_2^j(L') := \begin{cases} j & \text{if } L' = L \\ \texttt{size}_{min}(L') - 1 & \text{if } L' \subseteq D_L \wedge E_L \not\prec E_{L'} \\ 0 & \text{otherwise, if } L' \cap D \neq \emptyset \wedge p_1 \prec E_{L'} \\ I_1(L') & \text{otherwise} \end{cases} \tag{A.69}$$

$$I_3^j(L') := \begin{cases} j & \text{if } L' = L \\ \texttt{size}_{min}(L') - 1 & \text{if } L' \subseteq D_L \wedge X_L \not\prec E_{L'} \\ 0 & \text{otherwise, if } L' \cap D \neq \emptyset \wedge p_1 \prec E_{L'} \\ I_1(L') & \text{otherwise} \end{cases} \tag{A.70}$$

We prove that:

1. there exists a path of weight w_1 from (p_1, I_1) to (X_L, I_3^0),
2. for every $j < \texttt{size}_{min}(L) - 1$, there exists a path of weight 0 from (X_L, I_3^j) to (E_L, I_2^{j+1}) in G_c,
3. for every $j < \texttt{size}_{min}(L$, there exists a path of weight w_2 from (E_L, I_2^j) to (X_L, I_3^j) in G_c, and
4. for every $j < \texttt{size}_{min}(L)$, there exists a path of weight 0 from (E_L, I_2^j) to (p_2, I_2) in G_c.

By concatenating the first path, then the second for $j = 0$, then the third and the second for every $j < \texttt{size}_{min}(L) - 1$, then the fourth for $j = \texttt{size}_{min}(L) = 1$, we create a path of weight $w = w_1 + (\texttt{size}_{min}(L) - 2)).w_2$ from (p_1, I_1) to (p_2, I_2) in G_c, thus proving the property.

Existence of the Paths

We now give the proof that each path exists. The first and the third path come from the induction hypothesis. The second comes from the definition of edges in G_c. The last follows increasing indexes for a fixed program point. The following lemma helps applying the induction hypothesis to the first and the third path. The proof of this lemma derives directly from the definition of I_2^j, I_3^j and the constraints on I_1. It is thus omitted here.

Lemma A.3. *Let $(p, I) \in \mathbb{E}$ be an execution point equal to (p_1, I_1), (E_{L_i}, I_2^j) or (X_{L_i}, I_3^j) for $j < \texttt{size}_{min}(L_i)$. Then (p, I) respect the first two conditions to apply Proposition 5.6 to the graph $G_c^{D_L}$. Specifically, for every loop level $L' \in \mathbb{L}$, (p, I) satisfies the following conditions.*

$$L' \subseteq D_L \wedge p_1 \not\prec E_{L'} \implies I(L') = \texttt{size}_{min}(L') - 1 \tag{A.71}$$
$$L' \cap D_L \neq \emptyset \wedge p_1 \prec E_{L'} \implies I(L') = 0 \tag{A.72}$$

Existence of a Path from (p_1, I_1) to (X_L, I_3^0). We already know that there exists a path from p_1 to X_L of weight w_1 in $G_c^{D_L}$. We apply the induction hypothesis to obtain a path of same weight from (p_1, I_1) to (X_L, I_3^0). For that we need to prove that I_1 and I_3^0 respect the conditions (5.47–5.49) of Proposition 5.6. Conditions (5.48) and (5.47) are already provided by Lemma A.3. We thus prove that $\forall L' \in \mathbb{L}.\ L' \cap D_L = \emptyset \implies I_1(L') = I_3^0(L')$.

Let $L' \in \mathbb{L}$ such that $L' \cap D_L = \emptyset$. We distinguish four cases.

- If $L' \subseteq D_L$, then $L' = \emptyset$, thus $I_1(L') = 0 = I_3^j(L')$.
- If $L' = L$, then $I_3^0(L') = 0$. Moreover, $L' \subseteq D$ and $p_1 \prec E_{L'}$. Thus $I_2(L') = 0$. Thus $I_2(L') = I_3^0(L')$.
- Otherwise, if $L' \cap D \neq \emptyset$ and $p_1 \prec E_{L'}$, then $I_1(L') = 0 = I_3^0(L')$.
- Otherwise, by definition of I_3^0, $I_1(L') = I_3^0(L')$.

Existence of an Edge from (X_L, I_3^j) to (E_L, I_2^{j+1}). Let $j < \texttt{size}_{min}(L) - 1$. We show that (X_L, I_3^j) and (E_L, I_2^{j+1}) satisfy the conditions of the fourth case of the edges definition, in the defintion of G_c. For that, we define $\mathbb{L}_{\texttt{IN}} \subseteq \mathbb{L}$ as follows.

$$\mathbb{L}_{\texttt{IN}} := \{L' \in \mathbb{L} \mid E_L \prec E_{L'} \wedge X_{L'} \prec X_L\} \tag{A.73}$$

We the need to show that L is exclusive in in I_3^j and I_2^{j+1} respects the following definition.

$$\forall L' \in \mathbb{L}.\ I_2^{j+1}(L') := \begin{cases} I_3^j(L) + 1 & \text{if } L = L' \\ 0 & \text{if } L' \in \mathbb{L}_{\texttt{IN}} \\ I_3^j(L') & \text{otherwise} \end{cases} \tag{A.74}$$

We first show that L is exclusive in I_3^j. Let $L' \in \mathbb{L}$ be a loop level that contains a dimension in common with L, or a dimension that can be merged with a dimension of L. Then $p_1 \prec E_{L'}$, thus $I_1(L') = 0$. Moreover, $L' \not\subseteq D_L$ thus either $I_3^j(L') = 0$ or $I_3^j(L') = I_1(L') = 0$. Thus $I_3^j(L') = 0$. Thus L is exclusive in I_3^j.

We then show that equation (A.74) is satisfied. Let $L' \in \mathbb{L}$. We distinguish three cases depending on the case in (A.74) and in the definition of I_3^j.

- If $L' = L$, then $I_2^{j+1}(L') = I_3^j(L') + 1$. Thus the condition is satisfied.
- If $L' \in \mathbb{L}_{\texttt{IN}}$, then $E_L \prec E_{L'}$. Moreover, $X_{L'} \prec X_L$ thus $L' \subseteq D_L$. Since $E_L \prec E_{L'}$, $L' \neq \emptyset$, thus $L' \cap D_L \neq \emptyset$. Thus $I_2^{j+1} = 0$ and the condition is satisfied.
- If $L' \subseteq D_L$ and $E_L \prec E_{L'}$ then $L' \in \mathbb{L}_{\texttt{IN}}$ since $L' \subseteq D_L$ implies $X_{L'} \preceq X_L$. Thus this case is already covered.
- If $L' \subseteq D_L$ and $E_L \not\prec E_{L'}$, then $X_L \not\prec E_{L'}$ since $E_L \prec X_L$ and prec is transitive. Thus, $I_3^j(L') = \texttt{size}_{min}(L') - 1 = I_2^{j+1}$.
- Otherwise, if $L' \cap D \neq \emptyset \wedge p_1 \prec E_{L'}$, then $I_2^j(L') = 0 = I_3^{j+1}(L')$.
- Otherwise, $I_2^j(L') = I_1(L') = I_3^{j+1}(L')$.

Existence of a Path from (E_L, I_2^j) to (X_L, I_3^j). Let $j < \texttt{size}_{min}(L)$. We already know that there exists a path from E_L to X_L of weight w_2 in $G_c^{D_L}$. We apply the induction hypothesis to obtain a path of same weight from (E_L, I_2^j) to (X_L, I_3^j). For that we need to prove that I_2^j and I_3^j respect the conditions (5.47–5.49) of Proposition 5.6. Conditions (5.48) and (5.47) are already provided by Lemma A.3. We thus prove that $\forall L' \in \mathbb{L}.\ L' \cap D_L = \emptyset \implies I_2^j(L') = I_3^j(L')$.

Let $L' \in \mathbb{L}$ such that $L' \cap D_L = \emptyset$. We distinguish four cases.

- If $L' \subseteq D_L$, then $L' = \emptyset$, thus $I_2^j(L') = 0 = I_3^j(L')$.
- If $L' = L$, then $I_2^j(L') = j = I_3^j(L')$.
- Otherwise, if $L' \cap D \neq \emptyset$ and $p_1 \prec E_{L'}$, then $I_2^j(L') = 0 = I_3^j(L')$.
- Otherwise, $I_2^j(L') = I_1(L') = I_3^j(L')$.

Existence of a Path from (E_L, I_2^j) to (p_2, I_2). Let $j < \texttt{size}_{min}(L)$. We prove that for every loop level $\forall L \in \mathbb{L}$. $I_2^j(L) \leq I_2(L)$. This proves that there exists a path of weight 0 from $(p_2, I_2^j) = (E_{L_i}, I_2^j)$ to (p_2, I_2) in G_c, that follows the edges defined in item 6 of the definition of G_c. Let $L' \in \mathbb{L}$. We distinguish

- If $L' = L$, then $E_{L'} \preceq E_L$ thus $I_2(L') = \texttt{size}_{min}(L') - 1 \geq I_2^j(L')$.
- If $L' \subseteq D_L \wedge E_L \not\preceq E_{L'}$, then $L' \subseteq D$ thus $I_2(L') = \texttt{size}_{min}(L') - 1 \geq I_2^j(L')$.
- Otherwise, if $L' \cap D \neq \emptyset \wedge p_1 \prec E_{L'}$, then $I_2^j(L') = 0 \leq I_2(L')$.
- Otherwise, $I_2^j(L') = I_1(L') \leq I_2(L')$.

RÉSUMÉ

Les compilateurs cherchant à améliorer l'efficacité des programmes doivent déterminer quelles optimisations seront les plus bénéfiques. Ce problème est complexe, surtout lors des premières étapes de la compilation où chaque décision influence les choix disponibles aux étapes suivantes. Nous proposons de représenter la compilation comme le raffinement progressif d'une implémentation partiellement spécifiée. Les décisions possibles sont toutes connues dès le départ et commutent. Cela permet de prendre les décisions les plus importantes en premier et de construire un modèle de performance capable d'anticiper les potentielles optimisations. Nous appliquons cette approche pour générer du code d'algèbre linéaire ciblant des GPU et obtenons des performances comparables aux bibliothèques optimisées à la main.

MOTS CLÉS

Compilation, Optimisation de code, Modèle de Performance, GPU, Programation par Contraintes

ABSTRACT

Compilers looking for an efficient implementation of a function must find which optimizations are the most beneficial. This is a complex problem, especially in the early steps of the compilation process. Each decision may impact the transformations available in subsequent steps. We propose to represent the compilation process as the progressive refinement of a partially specified implementation. All potential decisions are exposed upfront and commute. This allows for making the most discriminative decisions first and for building a performance model aware of which optimizations may be applied in subsequent steps. We apply this approach to the generation of efficient GPU code for linear algebra and yield performance competitive with hand-tuned libraries.

KEYWORDS

Compilation, Code Optimization, Performance Model, GPU, Constraint Programing